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 compare and contrast several algorithms, for each function, in regards to their efficiency and accuracy.

Contents

1 Introduction

For many thousands of years all calculations that a person 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 $\frac{223}{71} < \pi < \frac{22}{7}$[1][2, p. 106], which while sufficient for their needs is not sufficient for ours in the present.

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[3]. Similarly relevant have been the periodic trigonometric functions due to their relation to triangles, and exponential functions due to their use in finance for example to find 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 17$^{\text{th}}$ century until the 20$^{\text{th}}$ century; who would be hired for that purpose.

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 is Charles Babbage's difference engine[4] which was a large mechanical calculator that would tabulate polynomial functions developed in the early 1800s.

Eventually 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. Today 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 being developed.

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[5] 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 some applications where we may desire even more precision than this. To solve this I will be implementing certain algorithms using the GNU Multiple Precision Arithmetic Library[6] (referred to as GMP) as well as GNU MPFR Library[7] (referred to as MPFR), 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 $\alpha \in [\frac{1}{2}, 1), \beta \in \mathbb{Z}$ such that $x = \alpha \cdot 2^\beta$; an operation that would usually be equivalent to calculating the non-trivial $\log_2(x)$. 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 moving forwards.

2.1 Methods

In this document we will look at various functions, such as root functions and 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[8, p. 84]. 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 2.1.1, 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^*.

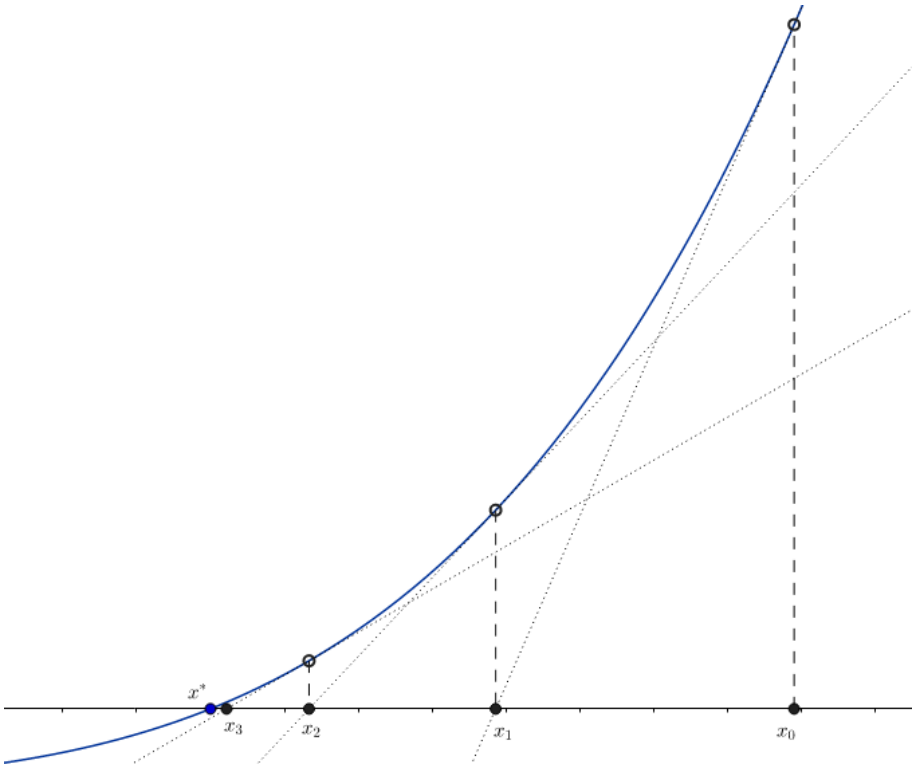
The specific definition of the Newton-Raphson method that I will be using in this document is below:

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

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

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.

Figure 2.1.1: Demonstration of Newton-Raphson Method



2.1.2 Taylor Series Expansion

The Taylor Series formulation was created by Brook Taylor in 1715[9], 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 interval $\mathcal{I}$ centred at $a \in \mathbb{R}$, we define the Taylor Series of f on $\mathcal{I}$ to be:

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

It was shown that on the open interval $\mathcal{I}$ 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 interval 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 Maclaurin series which is a Taylor series in an interval around $a = 0$. Thus a Maclaurin series has the form:

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

Some examples of simple Maclaurin Series are:

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n \qquad \forall\, x \in (-1,1) \tag{2.1.1}$$

$$(1+x)^k = \sum_{n=0}^{\infty} \binom{k}{n} x^n \qquad \forall\, x \in (-1,1),\ k \in \mathbb{N} \tag{2.1.2}$$

2.2 Errors

The error of an approximation $\tilde{v}$ for some v is a measure of how much $\tilde{v}$ differs from v. 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 need 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

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

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

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*

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

Proof. For $\implies$:
Suppose that $(x_n : n \in \mathbb{N})$ converges to x uniformly. Then

$$\forall\, \tau \in \mathbb{R}_0^+ \;\exists\, N \in \mathbb{N} \text{ s.t. } \;\; |x_n - x| < \tau \;\; \forall\, n \in [N, \infty) \cap \mathbb{Z}$$

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^**:*

$\mathbf{NR_1}$**:** $f'(x) \neq 0 \;\forall\, x \in I := [x^* - r, x^* + r],$ where $r \in [|x^* - x_0|, \infty)$

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

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

Proof. By Taylor's Theorem with Lagrange Remainders[9, p. 80] we have that

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

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

Then we get the following derivation:

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

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[10, p. 909].

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

$$\begin{aligned}
\lim_{n\to\infty} z_n &= \lim_{n\to\infty} (M\epsilon_0)^{2^n-1}\epsilon_0 \\
&= \epsilon_0 \lim_{n\to\infty} (M\epsilon_0)^{2^n-1} \\
&= \epsilon_0 \cdot 0 \qquad\qquad \text{because } M|\epsilon_0| < 1 \\
&= 0
\end{aligned}$$

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 however, 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[11] 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 consists of 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, which was 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

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

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 $\sqrt{N} = a_0 10^n + a_1 10^{n-1} + \dots$, 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

```
1   exactRootDigits(N ∈ ℝ₀⁺, d ∈ ℕ):
2       Digits_a := ∅
3       Digits_b := ∅
4       k := 0
5       n := ⌊½ log₁₀(N)⌋
6       while k < d:
7           a_k := max{t ∈ [0,9] ∩ ℤ : (20 Σ_{i=0}^{k-1} a_i 10^{k-i-1} + t) t 10^{2n-2k} ≤ N}
8           N ↦ N − (20 Σ_{i=0}^{k-1} a_i 10^{k-i-1} + a_k) a_k 10^{2n-2k}
9           if n − k < 0:
10              Digits_b ↦ Digits_b ∪ {a_k}
11          else:
12              Digits_a ↦ Digits_a ∪ {a_k}
13          k ↦ k + 1
14      if Digits_a = ∅:
15          Digits_a := {0}
16      if Digits_b = ∅:
```

```
17              Digits_b := {0}
18          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 $k \in [0, d] \cap \mathbb{Z}$. 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 ∈ ℝ₀⁺, d ∈ ℕ):
2       Digits_a := ∅
3       Digits_b := ∅
4       k := 0
5       n := ⌊½ log₁₀(N)⌋
6       P_0 := 0
7       while k < d:
8           a_k := max{t ∈ [0,9] ∩ ℤ : (20P_k + t) t 10^(2n−2k) ≤ N}
9           N ↦ 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 ↦ Digits_b ∪ {a_k}
13          else:
14              Digits_a ↦ Digits_a ∪ {a_k}
15          k ↦ k + 1
16      if Digits_a = ∅:
17          Digits_a := {0}
18      if Digits_b = ∅:
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 $\leq$ 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 $\mathcal{O}(1)$.

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

```
1   integerSquareRoot(N ∈ ℤ₀⁺):
2       M := N
```

```
3       m := max{m ∈ ℤ₀⁺ : 4^m ≤ M}
4       r := 0
5       while m ≥ 0:
6           if 2r(2^m) + 4^m ≤ M:
7               M ↦ M − 2r(2^m) + 4^m
8               r ↦ r + 2^m
9           m ↦ 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 .

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

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

        while (bit > 0)
        {
```

```
10                  if (res + bit <= num)
11                  {
12                          num = num - (res + bit);
13                          res = (res >> 1) + bit;
14                  }
15                  else
16                          res = res >> 1;
17
18                  bit = bit >> 2;
19          }
20
21          return res;
22  }
```

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

```
1   bisectionMethod(a ∈ ℝ, b ∈ (a, ∞), f ∈ C[a, b], τ ∈ ℝ⁺)
2       a_0 := a
3       b_0 := b
4       x_0 := (1/2)(a + b)
5           n := 0
6           while f(x_n) ≠ 0 AND b_n − a_n > τ:
7           if f(x_n) < 0:
8               a_{n+1} := x_n
9               b_{n+1} := b_n
10          else:
11              a_{n+1} := a_n
12              b_{n+1} := x_n
13          n ↦ n + 1
14          x_n := (1/2)(a_n + b_n)
15      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

```
1   bisectionSquareRoot(N ∈ ℝ₀⁺, τ ∈ ℝ⁺)
2       a_0 := 0
3       b_0 := max 1, N
4       x_0 := (1/2)(a_0 + b_0)
5       n := 0
6       while x_n^2 − N ≠ 0 AND b_n − a_n > τ:
7           if x_n^2 − N < 0:
8               a_{n+1} := x_n
9               b_{n+1} := b_n
10          else:
11              a_{n+1} := a_n
12              b_{n+1} := x_n
13          n ↦ n + 1
14          x_n := (1/2)(a_n + b_n)
15      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 $\sqrt{N} \in [a_0, b_0]$ 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 or implementing an integer power function, intPow, to use in place of x_n^2. This gives the following algorithm:

Algorithm 3.2.3: Bisection Method for General Roots

```
1   kRootBisectionMethod(N ∈ ℝ₀⁺, k ∈ [2,∞) ∩ ℤ, τ ∈ ℝ⁺)
2       a_0 := 0
3       b_0 := max 1, N
4       x_0 := ½(a_0 + b_0)
5       n := 0
6       while intPow(x_n, k) − N ≠ 0 AND b_n − a_n > τ:
7           if intPow(x_n, k) − N < 0:
8               a_{n+1} := x_n
9               b_{n+1} := b_n
10          else:
11              a_{n+1} := a_n
12              b_{n+1} := x_n
13          n ↦ n + 1
14          x_n := ½(a_n + b_n)
15      return x_n
```

The proof that method converges to the correct root is very similar to the proof of convergence for algorithm 3.2.2; and as such will be ommitted here.

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 $\sqrt{k}N$. 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 $\epsilon_n := |x_n - \sqrt{N}| \leq \frac{1}{2}(b_n - a_n)\ \forall\ n \in \mathbb{N}$. 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 $\epsilon_n \leq 10^{-d}$, to achieve this we must find $n \in \mathbb{N}$ such that $2^n \geq 10^d$. For example the following table indicates the minimum 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 is 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

```
1   bisectionSquareRoot(N ∈ ℝ₀⁺, τ ∈ ℝ⁺):
2       Let (r, e) ∈ [½, 1) : N = r · 2^e
3       if 2 ∤ e:
4           r ↦ r/2
5           e ↦ e + 1
6       a_0 := 0
7       b_0 := 1
8       x_0 := ½(a_0 + b_0)
9       n := 0
10      while x_n^2 − N ≠ 0 AND b_n − a_n > τ:
11          if x_n^2 − N < 0:
12              a_{n+1} := x_n
13              b_{n+1} := b_n
14          else:
15              a_{n+1} := a_n
16              b_{n+1} := x_n
17          n ↦ n + 1
18          x_n := ½(a_n + b_n)
19      return x_n · 2^(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 $f(x^*) = 0$ 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

```
1   NewtonSquareRoot(N ∈ ℝ, x_0 ∈ ℝ, τ ∈ (0, 1)):
2       n := 0
3       loop:
4           x_{n+1} := ½(x_n + N/x_n)
5           δ_n := |x_{n+1} − x_n|
6           if δ_n ≤ τ:
7               return x_{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 cases 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 version 1

```
1   NewtonSquareRoot_v1 ( N ∈ ℝ₀⁺, τ ∈ (0,1) ) :
2       if  N ∈ {0,1} :
3           return  N
4       if  N > 1 :
5           x_0 := N
6       else :
7           x_0 := 1
8       n := 0
9       loop :
10          x_{n+1} := ½(x_n + N/x_n)
11          δ_n := |x_{n+1} − x_n|
12          if  δ_n ≤ τ :
13              return  x_{n+1}
14          n ↦ 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. □

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 $I \not\subset \mathbb{R}^+$ 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 $M := \sup \left\{ \left| \frac{f''(x)}{f'(x)} \right| : x \in I \right\}$ 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:

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

Algorithm 3.3.3: Basic Newton Method for Square Root version 2

```
1   NewtonSquareRoot_v2(N ∈ ℝ₀⁺, τ ∈ (0,1)):
2       if N ∈ {0,1}:
3           return N
4       if N < 1:
5           x_0 := 1
6       else:
7           if N ≤ 4:
8               x_0 := N
9           else:
10              x_0 := IntSqrt(N)
11      n := 0
12      loop:
13          x_{n+1} := (1/2)(x_n + N/x_n)
14          δ_n := |x_{n+1} − x_n|
15          if δ_n ≤ τ:
16              return x_{n+1}
17          n ↦ 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

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

Where $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ is the nearest integer function. We can then implement the above observations in the following algorithm:

Algorithm 3.3.4: Newton Method for Square Root version 3

```
1   NewtonSquareRoot_v3(N ∈ ℝ₀⁺, τ ∈ (0,1)):
2       Let (a,b) :∈ [1/2, 1) × ℤ s.t. N = a · 2^b
3       x_0 := 1
```

```
4        if  b ≡ 0 mod 2:
5            α := 1
6        else:
7            if  b > 0:
8                α := √2
9            else:
10               α := 1/√2
11       n := 0
12       loop:
13           x_{n+1} := ½(x_n + a/x_n)
14           δ_n := |x_{n+1} − x_n|
15           if  δ_n ≤ τ:
16               return  α · x_{n+1} · 2^⌊b/2⌋
17           n ↦ 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 is the case that 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 $x_0 = 1$, we can then 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:

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

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 $n \geq log_2(log_{2\sqrt{2}}(2 \cdot 10^{10}(\sqrt{2}-1)))$. Calculating this value we find 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 thusly 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 acceptable, 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 in mind 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.

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

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

```
1   NewtonInvSquareRoot( N ∈ ℝ₀⁺, τ ∈ (0,1) ):
2       Let (a,b) :∈ [1/2,1) × ℤ s.t. N = a·2^b
3       x_0 := 1
4       if b ≡ 0 mod 2:
5           α := 1
6       else:
7           if b > 0:
8               α := 1/√2
9           else:
10              α := √2
11      n := 0
12      loop:
13          x_{n+1} := x_n(3/2 + (a/2)x_n^2)
14          δ_n := |x_{n+1} − x_n|
15          if δ_n ≤ τ:
16              return N·α·x_{n+1}·2^⌊−b/2⌉
17          n ↦ 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

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

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 $\sqrt{N}^{-1}$. □

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. For now we will merely observe that it is an inefficient method that 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 the methods discussed work by the same mechanism of successive approximations, and have similar complexity for each iteration; then we will compare the efficiency of these methods by their computation time. 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:

	Total time:	Average time:	Minimum time:	Maximum time:
mpfr_newton_sqrt_v1	10.507s	0.010s	0.003s	0.016s
mpfr_newton_sqrt_v2	12.707s	0.012s	0.004s	0.021s
mpfr_newton_sqrt_v3	8.188s	0.008s	0.005s	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	bisectSquareRoot	NewtonSquareRoot	NewtonInvSquareRoot
0	**0.**500000000000000	**1.**000000000000000	**0.**750000000000000
1	**0.**750000000000000	**0.8**75000000000000	**0.8**43750000000000
2	**0.8**75000000000000	**0.8660**71428571428	**0.86**5173339843750
3	**0.8**12500000000000	**0.86602540**5007363	**0.86602**4146705512
4	**0.8**43750000000000	**0.866025403784438**	**0.8660254037**81701
5	**0.8**59375000000000	**0.866025403784438**	**0.866025403784438**
6	**0.86**7187500000000	**0.866025403784438**	**0.866025403784438**

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:

	Total time:	Average time:	Minimum time:	Maximum time:
`root_digits_precise`	227.620s	0.227s	0.160s	0.429s
`bisect_sqrt`	2.520s	0.002s	0.002s	0.004s
`newton_sqrt`	1.028s	0.001s	0.000s	0.004s
`newton_inv_sqrt`	0.646s	0.000s	0.000s	0.001s
`builtin_sqrt`	0.072s	0.000s	0.000s	0.000s

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 3.4.1 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 of 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. Presently 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 lay out such identities that will prove useful in this document. As with Section 2 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.

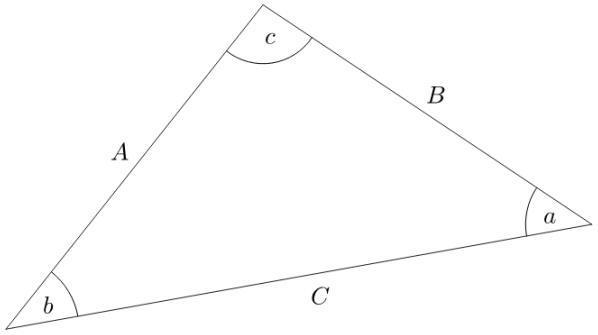
The first identities to consider are the basic ones taught in secondary school such as $\sin^2 x + \cos^2 x = 1$. In particular we are interested in the shifts, reflections and periods of $\sin$ and $\cos$. Some of the relevant functions are included below:

$$\begin{aligned} \sin(-x) &= -\sin(x) & \cos(-x) &= \cos(x) \\ \sin(x+\tfrac{\pi}{2}) &= \cos(x) & \cos(x+\tfrac{\pi}{2}) &= -\sin(x) \\ \sin(x+\pi) &= -\sin(x) & \cos(x+\pi) &= -\cos(x) \\ \sin(x+2\pi) &= \sin(x) & \cos(x+2\pi) &= \cos(x) \end{aligned}$$

Another useful formula to remember is the sine rule, which is detailed in figure 4.1.1 as well as the combined angle formulas

$$\begin{aligned} \sin(x) &= \sin(x)\cos(y) \pm \sin(y)\cos(x) \\ \cos(x \pm y) &= \cos(x)\cos(y) \mp \sin(x)\sin(y) \\ \tan(x) &= \frac{\tan(x) \pm \tan(y)}{1 \mp \tan(x)\tan(y)} \end{aligned}$$

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

$$\tfrac{d}{dx}\sin(x) = \cos(x) \qquad \tfrac{d}{dx}\cos(x) = -\sin(x) \qquad \tfrac{d}{dx}\tan(x) = \sec^2(x)$$

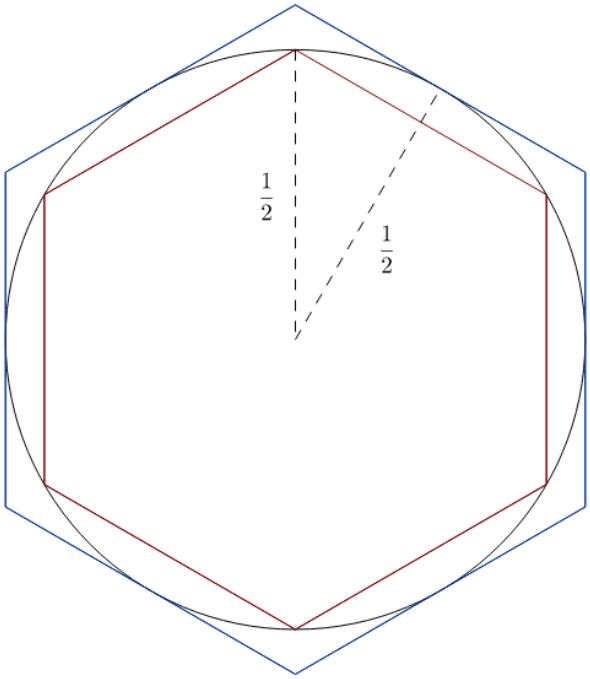
will be useful later on in the development of methods.

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[1][2, p. 106]. 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 $P_n = n\tan(\frac{\pi}{n})$.

Figure 4.2.1: Ancient method of calculating π



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 $p_{2n} = \sqrt{p_nP_{2n}}$[12], 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[13]:

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

or the Chudnovsky algorithm[14]:

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

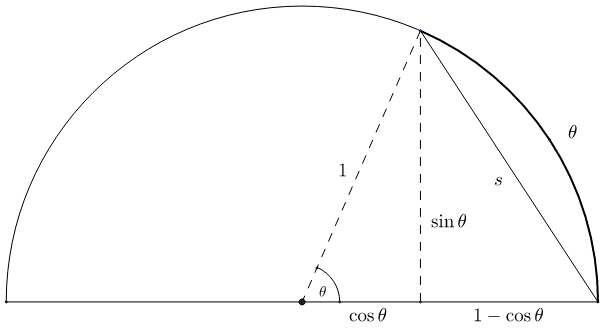
This final series is extremely rapidly convergent to the value of $\frac{1}{\pi}$, for example just the first term gives π accurate to 13 decimal places while we can get π accurate to 1000 decimal places with summing just 71 terms. Compared to Equation 4.2.1 which takes the summation of 500 terms to 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 Exploratorium.edu [15]

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 figure 4.3.1, which shows a unit circle.

Figure 4.3.1: Diagram showing angles to be dealt with



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:

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

We will now consider a figure 4.3.2 which will allow us to calculate an approximate value of 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 $s^2 = h^2(4-h^2)$. We now have the basis for a method that will allow us to calculate $\cos\theta$.

To complete our method we will introduce a new line that is to h what h is to s as shown in figure 4.3.3.
We then 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

Figure 4.3.2: Diagram detailing how to calculate s

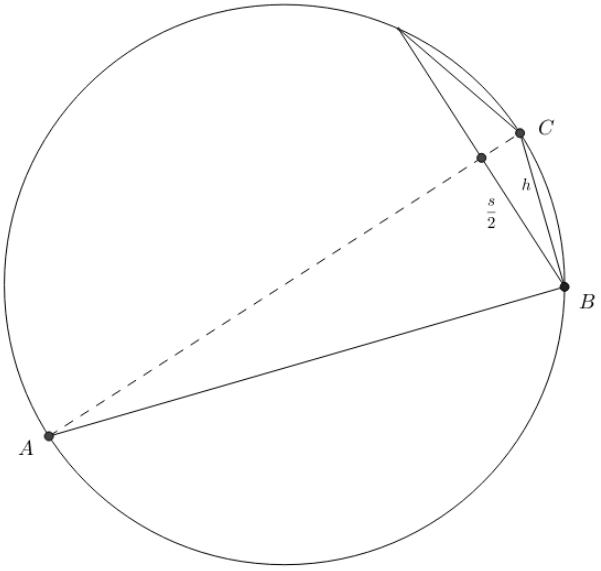
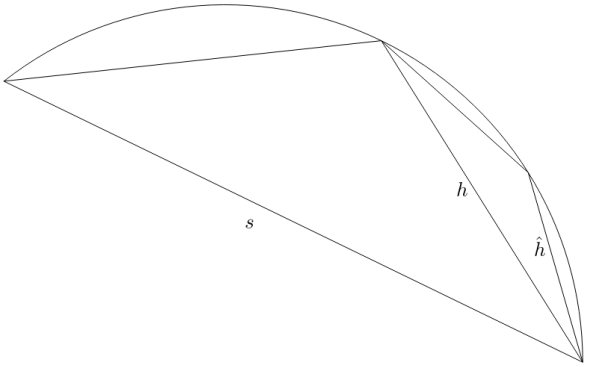


Figure 4.3.3: Detailing the recursive steps



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 $\cos$

```
1  geometric_cos(θ ∈ [0, π/2], k ∈ ℕ)
2      h_0 := θ2^{-k}
3      n := 0
4      while n < K:
5          h_{n+1}^2 := h_n^2 · (4 − h_n^2)
6          n ↦ n + 1
7      return 1 − (1/2)h_k^2
```

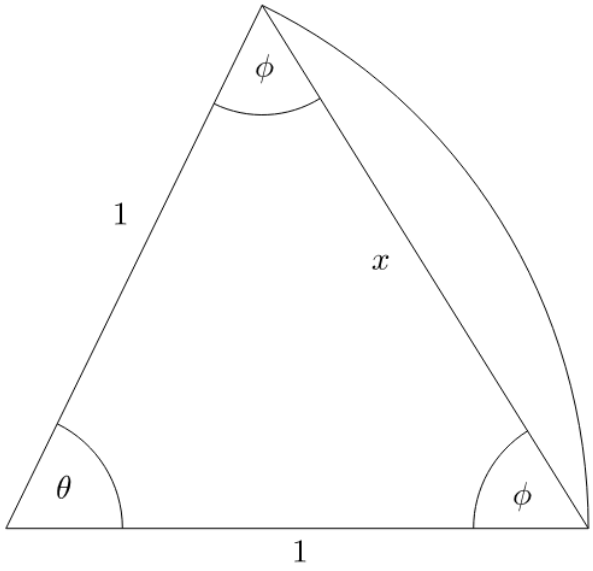
Now we can use the above pseudo-code to calculate any trigonometric function value by using various trigonometric identities. First we suppose $\theta \in \mathbb{R}$, then we can repeatedly apply the identity $\cos\theta = \cos(\theta \pm 2\pi)$ to either add or subtract 2π until we have a value $\theta'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

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

Now we can recall the trigonometric identities 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 $n \in [0, k] \cap \mathbb{Z}$.

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

$\mathbf{H}(0)$**:**

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

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

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

□

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.

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:

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

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 $\epsilon_{\mathcal{C}} = |\mathcal{C} - \cos(\theta)|$:

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

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:

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

It is not immediately obvious that $2\theta - 2^{k+2}\sin(\theta 2^{-k-1})$ is a useful upper bound for $\epsilon_{\mathcal{C}}$. However if we 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 this 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 ∈ [0,1], k ∈ ℕ)
2      h_k := 2 − 2x
3      n := k − 1
4      while n ≥ 0:
5          h_n^2 := 2 − √(4 − h_{n+1}^2)
6          n ↦ 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 3, 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 Maclaurin Series from Section 2.1.2, 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. Using the ratio test for series[**a**]nd quation 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 $N \in \mathbb{N}$:

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

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 $\sin$

```
1   taylor_cos (x ∈ ℝ, N ∈ ℕ)
2       C := 0
3       n := 0
4       while n < N:
5           C ↦ C + (−1)^n · 1/(2n)! x^{2n}
6           n ↦ n + 1
7       return C
8
9   taylor_sin (x ∈ ℝ, N ∈ ℕ)
10      S := 0
11      n := 0
12      while n < N:
13          S ↦ S + (−1)^n · 1/(2n+1)! x^{2n+1}
14          n ↦ n + 1
15      return 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 is $x^{2(n-1)} \cdot x^2$ 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

```
1   taylor_cos (x ∈ ℝ, N ∈ ℕ)
2       C := 1
```

```
3        x_2 := x^2
4        a := 1
5        b := 1
6        n := 1
7        while  n < N :
8            a ↦ a · 1/((2n−1)(2n))
9            b ↦ b · x_2
10           C ↦ C − a · b
11           a ↦ a · 1/((2n+1)(2n+2))
12           b ↦ b · x_2
13           C ↦ C + a · b
14           n ↦ n + 2
15       return  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 $\text{taylor\_cos}(x, N)$ and divide the resulting value, or we can calculate $\tan x$ directly using a Taylor expansion.

In calculating the Maclaurin 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 Maclaurin series only contains non-zero coefficients for odd powers of x[16]; therefore we get that $\tan x = \sum_{n=0}^{\infty} a_{2n+1} x^{2n+1} =$

$a_1x + a_3x^3 + a_5x^5 + \cdots$.

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

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

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 coefficient 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 4.4.2, 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 Maclaurin 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:

$$\begin{aligned}\tan^{-1} x &= \int \frac{1}{1+x^2}dx \\ &= \int (1 - (-x^2))^{-1}dx \\ &= \int \sum_{n=0}^{\infty}(-x^2)^n dx \qquad \text{by Equation 2.1.2} \\ &= \int \sum_{n=0}^{\infty}(-1)^n x^{2n} dx \\ &= c + \sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}x^{2n+1}\end{aligned}$$

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 2.1.2 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 Maclaurin series above. This gives the following method

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

```
1   taylor_aTan (x ∈ [0,1], N ∈ ℕ)
2       𝒯 := 0
3       x₂ := x²
4       y := x
5       n := 0
6       while n < N:
7           𝒯 ↦ 𝒯 + 1/(2n+1) y
8           y ↦ y · x₂
9           𝒯 ↦ 𝒯 − 1/(2n+3) y
10          y ↦ y · x₂
11          n ↦ n + 2
12      return 𝒯
```

Similar to Algorithm 4.4.2 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 $\sin^{-1}$

```
1   taylor_aSin(x ∈ [−1, 1], N ∈ ℕ)
2       if x = 1:
3           return π/2
4       if x = −1:
5           return −π/2
6       S := x
7       x_2 := x^2
8       y := x
9       a := 1
10      b := 1
11      c := 1
12      n := 1
13      while n < N:
14          a ↦ 2n · (2n − 1) · a
15          b ↦ n^2 · b
16          c ↦ 4 · c
17          y ↦ x_2 · y
18          S ↦ S + a/(b·c·(2n+1)) · y
19          n ↦ n + 1
```

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 **C**omputer[2, p. 138] 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 $\theta \in (-\frac{\pi}{2}, \frac{\pi}{2})$, to

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

To do this we repeatedly add and subtract our values for γ from θ 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 4.5.1, 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.

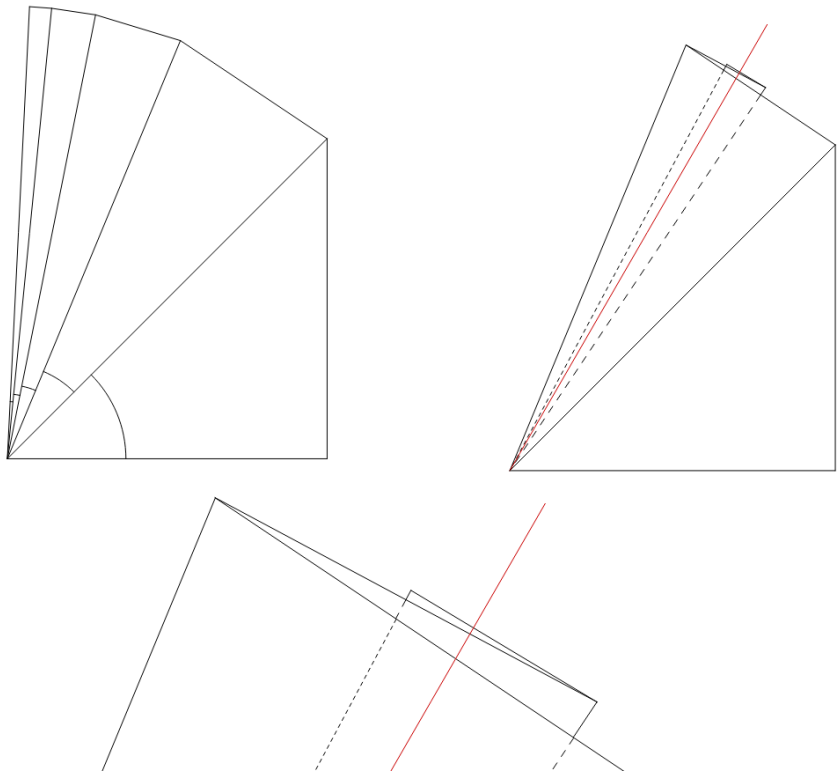
While there are many possible choices for γ_k we wish to consider $(\gamma_k : k \in [0, n] \cap \mathbb{Z})$ such that $\tan\gamma_k = 2^{-k}\ \forall\, k \in [0, n] \cap \mathbb{Z}$. We can note that the powers of 2 have a useful property, in that if $m > n \in \mathbb{N}$ we see that $\sum_{k=n}^{m-1} 2^k = 2^m - 2^n$. 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[17] 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 $n \in \mathbb{N}$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$ 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 $\begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$ 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 $\theta_n := \theta - \sum_{k=0}^{n-1} c_k\gamma_k$. At step n we then have $\theta_{n+1} = \theta_n - \gamma_n$ if $\theta_{n+1} \geq 0$, and $\theta_{n+1} = \theta_n + \gamma_n$ otherwise. This leads us to the general implementation of CORDIC for Trigonometric Functions:

Algorithm 4.5.1: General Cordic

```
1   CORDIC(θ ∈ (−π/2, π/2), n ∈ ℕ):
2       x := 1
3       y := 0
4       k := 0
5       while k < n:
6           if θ ≥ 0:
7               t := x
8               x ↦ 1/√(1+2^{−2k}) (x − 2^{−k} y)
9               y ↦ 1/√(1+2^{−2k}) (y + 2^{−k} t)
10              θ ↦ θ − tan^{−1}(2^{−k})
11          else:
12              t := x
13              x ↦ 1/√(1+2^{−2k}) (x + 2^{−k} y)
14              y ↦ 1/√(1+2^{−2k}) (y − 2^{−k} t)
15              θ ↦ θ + tan^{−1}(2^{−k})
16          k ↦ k + 1
17      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.

We first consider the representation of our values in the program, and while the previous algorithms used a floating point `double` value, as described in Section 1.1, we will instead use a fixed point representation for CORDIC. 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 processes of addition, subtraction, and multiplication & 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	$07f5_{16}$
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:

```
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 $\mathcal{O}(1)$ algo-

rithm, 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 A.3 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.**707106781186547572737 31
2	**0.**948683298050513768018 27
3	**0.8**436614877321074734695 1
4	**0.**903737838893538758533 45
5	**0.87**527458786899225984257
6	**0.8**899534681193336238536 0
...	...
19	**0.87758**301847694786257392
20	**0.877582**10404530012649360
21	**0.87758256**126152311971111
22	**0.877582**789869335244681 28
...	...
53	**0.87758256189037275**873943
54	**0.8775825618903726**4771712
55	**0.87758256189037275**873943
56	**0.87758256189037275**873943
...	...
63	**0.87758256189037275**873943

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 table 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 53-56 section of the table becomes:

n	Output with bold accurate digits
53	**0.877582561890372**73965747
54	**0.877582561890372**68653156
55	**0.8775825618903727**1298609
56	**0.877582561890372**72621336

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:

```
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 prior to the actual calculation, then we are never dividing by 0 and $z \in [-1, 1] \cap \mathbb{Z}$, 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, 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	$\text{geometric\_Cos}(0.5,\, n)$	$\text{taylor\_Cos}(0.5,\, n)$	$\text{CORDIC}(0.5,\, n)$
1	**0.87**6953125000000	**1.**000000000000000	**0.**707106781186547
2	**0.877**426177263259	**0.877**604166666666	**0.**948683298050513
3	**0.8775**43526076081	**0.877**604166666666	**0.8**43661487732107
4	**0.8775**72806699400	**0.87758256**2158978	**0.**903737838893538
5	**0.87758**0123327654	**0.87758256**2158978	**0.87**5274587868992
6	**0.87758**1952264380	**0.877582561890373**	**0.8**89953468119333
7	**0.877582**409484792	**0.877582561890373**	**0.8**82719918613777
8	**0.8775825**23789035	**0.877582561890372**	**0.87**9022003513595
9	**0.8775825**52365041	**0.877582561890372**	**0.877**152884812089
10	**0.8775825**59509040	**0.877582561890372**	**0.87**8089122532394

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:

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

These values show that the fastest algorithm that we have discussed is algorithm 4.4.2, 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[18], but it is implemented in a much lower level method that optimises the execution of the code.

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**51425307918200	**2.**270796326794896	**2.3**56194490192344
2	**2.34**7503635391542	**2.**327962993461563	**1.**892546881191538
3	**2.346**521397812842	**2.**340568243461563	**2.**137525544318402
4	**2.346**275724597314	**2.34**4244774711563	**2.**261880538865164
5	**2.346**214299177873	**2.34**5470795757570	**2.3**24299348861121
6	**2.34619**8942378459	**2.34**5913166442261	**2.3**55539182291389
7	**2.34619**5103149716	**2.346**081295659538	**2.3**39915453670913
8	**2.34619**4143336564	**2.3461**47594614218	**2.34**7727794731014
9	**2.346193**903386887	**2.3461**74467628018	**2.34**3821564599047
10	**2.3461938**43452078	**2.3461**85594784405	**2.34**5774687115525

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 $\mathrm{geometric\_aCos}$ is demonstrated in the following table:

n	geometric_aCos(0.5, n)
13	**2.34619382**2083380897
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	$\mathrm{geometric\_aCos}(0.5,\, n)$
13	**2.346193823**718087586
14	**2.3461938234**83759158
15	**2.3461938234**25177051
...	...
22	**2.34619382340560**50874
23	**2.34619382340564**9980
24	**2.34619382340564**757

With this we see that Algorithm 4.3.2 continues in the same pattern as before and is actually correct. So we may time our functions again to compare their efficiency. 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 $\mathrm{geometric\_aCos}$ when $n \geq 18$, $\mathrm{taylor\_aCos}$ when $n \geq 30$ and CORDIC when $n \geq 50$.

	Total time:	Average time:	Minimum time:	Maximum time:
`geometric_cos`	27.273s	0.027s	0.026s	0.033s
`taylor_cos`	14.358s	0.014s	0.014s	0.018s
`cordic_cos`	29.142s	0.029s	0.028s	0.032s
`builtin_cos`	2.143s	0.002s	0.001s	0.005s

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[19]; we therefore need to calculate x^y quickly and efficiently.

The first thing we consider is that x^y when $x \in \mathbb{R}^-$ 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:

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

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

```
1  naive_int_exp(x ∈ ℝ, a ∈ ℤ₀⁺):
2      n := 0
3      z := 1
4      while n < a:
5          z ↦ x · z
6          n ↦ n + 1
7      return z
```

This algorithm is very simple and has complexity of $\mathcal{O}(a)$, which is a reasonably low 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

```
1   exp_by_squaring(x ∈ ℝ, a ∈ ℤ₀⁺):
2       n := a
3       z := 1
4       x̂ := x
5       while n > 0:
6           if 2 ∤ n:
7               z ↦ x̂ × z
8               n ↦ n − 1
9           x̂ ↦ x̂²
10          n ↦ n/2
11      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:

	Total time:	Average time:	Minimum time:	Maximum time:
`naive_int_exp`	16.800s	0.016s	0.000s	0.037s
`squaring_int_exp`	2.593s	0.002s	0.000s	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 by 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 $e^c = \lim_{n\to\infty}(1+\frac{x}{n})^n$. This gives us our first basic method of how to calculate e^x:

Algorithm 5.3.1: Baisc Method for calculating e^x

```
1   basic_exp (x ∈ ℝ, n ∈ ℕ)
2       return (1 + 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:

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

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**48292488074831
10000	**2.225**4697161201278 50
100000	**2.2255**3380681087 3500
1000000	**2.225540**2163198643 58
10000000	**2.225540**8572751629 29
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 Maclaurin series of e^x. By the definition of a Maclaurin 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

```
1  taylor_exp(x ∈ ℝ, n ∈ ℤ₀⁺)
2      t = 1
3      z = 1
4      k = 1
5      while k < n:
6          t ↦ t·x/n
7          z ↦ z + t
8          k ↦ k + 1
9      return z
```

This allows us to calculate e^x more efficiently, and we can see that the error of the approximation is easy $\epsilon_n := |e^x - \sum_{k=0}^{n} \frac{x^k}{k!}| \leq \frac{|x|^{n+1}}{(n+1)!}$ 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 to use our method such that the error is at most $\tau_d := 10^{-d}$, then we need to find $n \in \mathbb{Z}_0^+ : \frac{1}{n!} < \tau_d$. 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

Therefore 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 Maclaurin 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 $x \in (0,2)$. □

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 1.1 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 $\ln(x)$

```
1   taylor_nat_log(x ∈ ℝ⁺, n ∈ ℕ):
2       Find (a, b) ∈ [1/2, 1) × ℤ such that x = a · 2^b
3       y := 1 − a
4       t := y
5       z := y
6       k := 2
7       while k < n:
8           t ↦ t · y
9           z ↦ z + t/k
10          k ↦ k + 1
11      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 note 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 $\log_x(y)$

```
1   taylor_log(x ∈ ℝ⁺, y ∈ ℝ⁺, n ∈ ℕ):
2       a := taylor_nat_log(y, n)
3       b := taylor_nat_log(x, n)
4       return a/b
5
6   taylor_pow(x ∈ ℝ⁺, y ∈ ℝ, n ∈ ℕ):
7       a := taylor_nat_log(x, n)
8       a ↦ y · a
```

```
9        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	**1.000**0000000	**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	**1.5180909816**
10	**7**193.4074522	**−9**04.7591286	**−**89.85299062	**1.5180909827**
...	...	...	...	...
20	**139**01.238666	**−11.**00988984	**−0.**092958315	**1.5180909827**
...	...	...	...	...
40	**13929.955484**	**0.0000717862**	**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	**−0.785**082036
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 taylor_log converges better than 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 taylor_exp. The value that had best convergence in the taylor_exp table 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:

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

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 $\ln$

```
1   hyperbolic_nat_log(x ∈ ℝ⁺, n ∈ ℤ₀⁺):
2       a := (x−1)/(x+1)
3       b := y²
4       c := a
5       k := 1
6       while k ≤ n:
7           a ↦ a · b
8           c ↦ c + a/(2k+1)
9           k ↦ k + 1
10      return 2 · 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 5.4.2, 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[20] 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[20, p. 266]:

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

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 the 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[21, Ch. 18], 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:

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

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:

$$\begin{aligned} e^x &= 1 + \cfrac{x}{1-\cfrac{\frac{1}{2}x}{1+\frac{1}{2}x-\cfrac{\frac{1}{3}x}{\ddots-\cfrac{\frac{1}{n-1}x}{1+\frac{1}{n-1}x-\frac{\frac{1}{n}x}{1+\frac{1}{n}x}}}}} \\ &= 1 + \mathbf{K}_{i=1}^{n}\frac{\alpha_i}{\beta_i} \end{aligned} \tag{5.6.4}$$

$$\text{where } \alpha_i := \begin{cases} x & : \ i = 1 \\ -\frac{1}{i}x & : \ i \in [2,n]\cap\mathbb{Z} \end{cases}$$

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

We want 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:*

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

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:

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

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:

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

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

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

This is a much simpler continued fraction, but evaluating it would still be computationally 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:

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

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 version 1

```
1   cont_frac_exp_v1 (x ∈ ℝ, n ∈ ℕ):
2       A_1 := x + 1
3       B_1 := 1
4       A_2 := x^2 + 2x + 2
5       B_2 := 2
6       a := −x
7       b := 2 + x
8       k := 2
9       while k <= n:
10          a ↦ a − x
11          b ↦ b + 1
12          A_{k+1} := bA_k + aA_{k−1}
13          B_{k+1} := bB_k + ab_{k−1}
14          k ↦ k + 1
15      return 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.

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 $|z - \frac{A_n}{B_n}| \leq \frac{1}{|B_{n+1}B_n|}$[20]. If we transform the continued fraction of e^x into this form by using Proposition 5.6.1, 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 $z = 2^a$ using algorithm 5.6.1. Then we can calculate z^{2^b} using algorithm 5.1.2, 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 its 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[22] 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

```
1   cont_frac_exp_v2 (x ∈ ℝ, n ∈ ℕ):
2       A_1 := 1
3       B_1 := 1
4       A_2 := 1
5       B_2 := 1 − x
6       a := 1
7       b := 1
8       k := 2
9       while k ≤ n:
10          a ↦ xa
11          b ↦ b + 1
12          A_{k+1} := bA_k + aA_{k−1}
13          B_{k+1} := bB_k + ab_{k−1}
14          k ↦ k + 1
15          b ↦ b + 1
16          A_{k+1} := bA_k − aA_{k−1}
17          B_{k+1} := bB_k − ab_{k−1}
18          k ↦ k + 1
19      return A_k / B_K
```

The fraction needed to analyse this method is again found by using proposition 5.6.1, 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 we get similar results to above, particularly 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[20, p. 86] 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:*

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

while the limit of the even convergent is:

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

Proof. Omitted □

If we apply proposition 5.6.1 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 5.6.1 one more time we reach the following continued fraction for e^x:

$$e^x = 1 + \cfrac{2x}{2 - x + \cfrac{x^2}{6 + \cfrac{x^2}{10 + \cfrac{x^2}{14 + \ddots}}}} \tag{5.6.9}$$

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

$$\beta_i := \begin{cases} 2 - x & : \quad i = 1 \\ 4i - 2 & : \quad i \in [2, \infty) \cap \mathbb{Z} \end{cases}$$

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

```
1   cont_frac_exp_v3 (x ∈ ℝ, n ∈ ℕ):
2       A_0 := 1
3       B_0 := 1
4       A_1 := 2 + x
5       B_1 := 2 − x
6       a := x^2
7       b := 2
8       k := 2
9       while k ≤ n:
10          b ↦ b + 4
```

11 $A_{k+1} := bA_k + aA_{k-1}$
12 $B_{k+1} := bB_k + ab_{k-1}$
13 $k \mapsto k + 1$
14 return $\frac{A_k}{B_K}$

As with the previous two continued fraction methods of approximating e^x we can apply proposition 5.6.1 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**21666666666	3.3333333333333	**2.013**2689987937
3	**2.01**21708333333	**2.0**769230769230	**2.01375**43842848
4	**2.013**5714166666	**2.0**054200542005	**2.0137527**042253
5	**2.0137**348180555	**2.013**2689987937	**2.013752707474**4
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	**2.0137527**069445	**2.0137527**042253	**2.0137527074704**
10	**2.013752707**4399	**2.0137527076**056	**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. Further, 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	**−0.3566**32500000	**−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, 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	**6.67887**34950163
4	**6.6**587865280763	**6.678873**557263
5	**6.67**48050386470	**6.6788735872**409
6	**6.678**0194099644	**6.67887358726**71
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**73534
8	**5.1**401274582517	**5.11826786**7291
9	**5.1**272612067887	**5.118267862**3951
10	**5.1**219353833296	**5.118267862**3708
...	...	...
20	**5.11826**84126550	**5.1182678623688**
...	...	...
30	**5.11826786**24756	**5.1182678623688**
...	...	...
40	**5.1182678623**689	**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
`builtin_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 we 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 its poor efficiency, though it has the interesting accuracy property of generating precisely one new digit each iteration. 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 to be accurate only to their integer part.

The root functions were successfully implemented in C, and when implemented with MPFR they 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 engaging 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 was found 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 its place. In particular CORDIC is still useful for simple systems that do not have the capability to handle 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 its use as being the most efficient method.

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

The Taylor methods 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 even methods other than those discussed here, the libraries utilising low level programming techniques to speed up computation, or a combination of the two.

References

[1] B. McKeeman. "The Computation of Pi by Archimedes". MathWorks File Exchange. Nov. 2010. URL: http://www.mathworks.com/matlabcentral/fileexchange/29504-the-computation-of-pi-by-archimedes/content/html/ComputationOfPiByArchimedes.html#37.

[2] G. Rising. *Inside Your Calculator. From Simple Programs to Significant Insights*. Wiley, 2007. ISBN: 978047011408.

[3] *Yale Babylonian Collection Images and Analysis*. URL: http://www.math.ubc.ca/~cass/Euclid/ybc/ybc.html.

[4] Computer History Museum. *The Babbage Engine*. URL: http://www.computerhistory.org/babbage/.

[5] Institue of Electrical and Electronics Engineers. *IEEE Standard for Floating-Point Arithmetic*. 2008. ISBN: 9780738157535. URL: https://standards.ieee.org/findstds/standard/754-2008.html.

[6] *The GNU Multiple Precision Artihmetic Library*. URL: https://gmplib.org/.

[7] *The GNU MPFR Library*. URL: http://www.mpfr.org/.

[8] A. Greenbaum and T. Chartier. *Numerical Methods. Design, analysis, and computer implementation of algorithms*. Princeton University Press, 2012. ISBN: 97806911511229.

[9] B. Taylor. *Methodus Incrementorum Directa et Inversa*. Direct and Reverse Methods of Incrementation. 1715.

[10] J. Stewart. *Multivariable Calculus*. 6th ed. Brooks Cole, June 2007. ISBN: 9780495011637.

[11] T. Henderson. "Cryptography and Complexity". 2012. URL: http://hackthology.com/pdfs/crypto-complexity.pdf.

[12] E. Howard. *An Introduction to the History of Mathematics*. 6th ed. Saunders College Publishing, 1992. ISBN: 9780880294188.

[13] S. Ramanujan. "Modular equations and approximations to pi". In: *Quart J Math: Oxford* (1914).

[14] D. Chudnovsky and G. Chudnovsky. "The Computation of Classical Constants". In: *PNAS* (Aug. 1989). URL: http://www.pnas.org/content/86/21/8178.full.pdf.

[15] Exploratorium.edu. *A million digits of Pi*. URL: http://www.exploratorium.edu/pi/pi_archive/Pi10-6.html.

[16] C. Stover. *Odd Function*. Wolfram Alpha. URL: http://mathworld.wolfram.com/OddFunction.html.

[17] F. Morgan. *Real Analysis*. AMS, 2005. ISBN: 0821836706.

[18] IBM. *Implementation of* COS *in* `math.h`. URL: https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;hb=HEAD#l281.

[19] L. Jordan. *Exponential Functions in the Real World*. URL: https://www.sophia.org/tutorials/exponential-functions-in-the-real-world--3.

[20] L. Lorentzen and H. Waadeland. *Continued Fractions. Volume 1: Convergence Theory*. Ed. by C. Chui. 2nd ed. Atlantis Press, 2008. ISBN: 9789078677079.

[21] L. Euler. *Introductio in analysin infinitorum*. 1748.

[22] W. Jones and W. Thron. *Continued Fractions. Analytic Theory and Applications*. Addison-Wesley Publishing Company, 1980.

[23] R. Parris. "Elementary Functions and Calculators". URL: http://math.exeter.edu/rparris/peanut/cordic.pdf.

[24] N. Artemiadis. *History of Mathematics. From a Mathematician's Vantage Point*. Trans. by N. Sofronidis. AMS, 2004. ISBN: 0821834037.

[25] C. Aliprantis and O. Burkinshaw. *Principles of Real Analysis*. 2nd ed. Academic Press Limited, 1990. ISBN: 0120502550.

A Code

In this appendix I list the entirety of the code which implement the algorithms discussed in the body of this document. The entirety of the codebase, as well as LaTeX files related to this document can be found on GitHub at https://github.com/Ybrad/Year-4-Project.

A.1 General Code

General Utilities File:

File : utilities.c

```
1 #include <gmp.h>
2 #include <mpfr.h>
```

```
3  #include "utilities.h"
4
5  const double ROOT_2     = 1.41421356237309504880168872420969807856967187537;
6  const double ROOT_2_INV = 0.70710678118654752440084436210484903928483593768;
7
8  inline unsigned int d(unsigned int D)
9  {
10   return D > 10 ? 10 : D;
11 }
12
13 void inline mpfr_digits_to_tolerance(unsigned int D, mpfr_t T)
14 {
15   mpfr_init_set_ui(T, 10, MPFR_RNDN);
16   mpfr_pow_ui(T, T, D, MPFR_RNDN);
17   mpfr_ui_div(T, 1, T, MPFR_RNDN);
18 }
```

Trigonometric Utilities File:

File : trig_utilities.c

```
1  #include <assert.h>
2  #include "trig_utilities.h"
3
4  TRIG_FIXED_TYPE double_to_fixed(double d)
5  {
6    assert(d < 2 && d >= -2);
7    return (TRIG_FIXED_TYPE) (d * (TRIG_FIXED_TYPE)CONVERSION_VALUE);
8  }
9
10 double fixed_to_double(TRIG_FIXED_TYPE t)
11 {
12   return (double)t / (TRIG_FIXED_TYPE)CONVERSION_VALUE;
13 }
```

Header Files for Utilities:

File : utilities.h

```
1  #ifndef UTILITIES_HEADER
2    #define UTILITIES_HEADER
3
4    #include <mpfr.h>
5
6    #define ROOT_2_INFILE "root_2_digits.txt"
7    #define ROOT_2_INV_INFILE "root_2_inv_digits.txt"
8
9    extern const double ROOT_2, ROOT_2_INV;
10   extern mpfr_t MPFR_ROOT_2, MPFR_ROOT_2,
11           MPFR_ONE, MPFR_HALF, MPFR_THREE_HALF, MPFR_TWO;
12
13   unsigned int d(unsigned int);
14   void mpfr_digits_to_tolerance(unsigned int, mpfr_t);
15 #endif
```

File : trig_utilities.h

```
1  #ifndef TRIG_UTILITIES
2    #define TRIG_UTILITIES
3
4    #include <mpfr.h>
```

```
  #include "trig_fixed.h"

  #define PI      3.14159265358979323846264338327950288419716939937510582
  #define HALF_PI 1.57079632679489661923132169163975144209858469968755291
  #define TWO_PI  6.28318530717958647692528676655900576839433879875021164
  #define PI_INFILE "pi_digits.txt"

  extern mpfr_t MPFR_PI, MPFR_HALF_PI, MPFR_TWO_PI;

  TRIG_FIXED_TYPE double_to_fixed(double);
  double fixed_to_double(TRIG_FIXED_TYPE);
#endif
```

File : log_exp_utilities.h

```
#ifndef LOG_EXP_UTILITIES_HEADER
#define LOG_EXP_UTILITIES_HEADER

  #include <mpfr.h>

  #define NAT_LOG_2 0.69314718055994530941723212145817656807550013436025
  #define E_CONST   2.71828182845904523536028747135266249775724709369995

  #define NAT_LOG_2_INFILE "nat_log_2_digits.txt"
  #define E_CONST_INFILE   "e_digits.txt"

  extern mpfr_t MPFR_NAT_LOG_2, MPFR_E_CONST;

#endif
```

Makefile for the project:

File : makefile

```
#Compiler and basic flags
CC=gcc
CFLAGS=-std=c11 -g

#Directories used
OBJDIR=obj
OUTDIR=out
LIBDIR=lib
TSTDIR=test

#Library linking options
MPFRLIB=-lgmp -lmpfr
MATHLIB=$(MPFRLIB) -lm
UTILLIB=-L$(LIBDIR) -lutil

#Used to compile a given file with the main program
EXE=-D COMPILE_MAIN

#The lists of files that are used or created
INFILES  =bisect_root.c exact_root.c newton_inv_sqrt.c newton_sqrt.c \
      geometric_trig.c geometric_inv_trig.c taylor_trig.c \
      taylor_inv_trig.c cordic_trig.c int_exp.c taylor_exp_log.c \
      hyperbolic_log.c cont_frac_exp.c
TESTFILES=test_newton_sqrt.c test_trig_methods.c test_inv_trig_methods.c \
      test_sqrt_methods.c test_int_exp_methods.c test_exp_methods.c \
      test_log_methods.c test_log_pow_methods.c
```

```
OUTFILES=$(addprefix $(OUTDIR)/, $(INFILES:.c=.out))
OBJECTS =$(addprefix $(OBJDIR)/, $(INFILES:.c=.o))
TESTS   =$(addprefix $(TSTDIR)/, $(TESTFILES:.c=.out))
LU=$(LIBDIR)/libutil.a

#Defualt option to build all the files
all: $(OBJECTS) $(LU) $(OUTFILES) $(TESTS)

#Cleans the workspace, best used for a fresh start
clean:
  rm $(OBJDIR)/*.o $(OUTDIR)/*.out $(LIBDIR)/*.a

#The following few are used to built the utilities library
$(OBJDIR)/util.o: utilities.c utilities.h
  $(CC) $(CFLAGS) -c $< -o $@

$(OBJDIR)/util_trig.o: trig_utilities.c trig_utilities.h
  $(CC) $(CFLAGS) -c $< -o $@

$(OBJDIR)/util_test.o: $(TSTDIR)/test_utilities.c $(TSTDIR)/test_utilities.h
  $(CC) $(CFLAGS) -c $< -o $@

$(LU): $(OBJDIR)/util.o $(OBJDIR)/util_trig.o $(OBJDIR)/util_test.o
  ar -cr $@ $(OBJDIR)/util*.o

#How to compile a c file to an object file
$(OBJECTS): $(notdir $(@:.o=.c)) $(notdir $(@:.o=.h)) utilities.h\
        trig_utilities.h
  $(CC) $(CFLAGS) -c $(notdir $(@:.o=.c)) -o $@

#The following are how to compile each of the OUTFILES
# The accompanying entry is a shorthand for the first

## SQUARE ROOT FILES ##
$(OUTDIR)/bisect_root.out: bisect_root.c bisect_root.h utilities.h $(LU)
  $(CC) $(CFLAGS) $(EXE) bisect_root.c \
    $(MATHLIB) $(UTILLIB) -o $@
bisect_root: $(OUTDIR)/bisect_root.out

$(OUTDIR)/exact_root.out: exact_root.c exact_root.h utilities.h $(LU)
  $(CC) $(CFLAGS) $(EXE) exact_root.c \
    $(MPFRLIB) $(UTILLIB) -o $@
exact_root: $(OUTDIR)/exact_root.out

$(OUTDIR)/newton_inv_sqrt.out: newton_inv_sqrt.c newton_inv_sqrt.h \
                    utilities.h $(LU)
  $(CC) $(CFLAGS) $(EXE) newton_inv_sqrt.c \
    $(MATHLIB) $(UTILLIB) -o $@
newton_inv_sqrt: $(OUTDIR)/newton_inv_sqrt.out

$(OUTDIR)/newton_sqrt.out: newton_sqrt.c newton_sqrt.h exact_root.h \
                utilities.h $(OBJDIR)/exact_root.o $(LU)
  $(CC) $(CFLAGS) $(EXE) newton_sqrt.c $(OBJDIR)/exact_root.o\
    $(MATHLIB) $(UTILLIB) -o $@
newton_sqrt: $(OUTDIR)/newton_sqrt.out

## TRIGONOMETRIC FILES ##
$(OUTDIR)/geometric_trig.out: geometric_trig.c trig_utilities.h \
```

```
                  geometric_trig.h $(LU)
  $(CC) $(CFLAGS) $(EXE) geometric_trig.c $(UTILLIB)\
    $(MATHLIB) -o $@
geometric_trig: $(OUTDIR)/geometric_trig.out

$(OUTDIR)/geometric_inv_trig.out: geometric_inv_trig.c trig_utilities.h \
                    geometric_inv_trig.h $(LU)
  $(CC) $(CFLAGS) $(EXE) geometric_inv_trig.c $(UTILLIB)\
    $(MATHLIB) -o $@
geometric_inv_trig: $(OUTDIR)/geometric_inv_trig.out

$(OUTDIR)/taylor_trig.out: taylor_trig.c trig_utilities.h \
                taylor_trig.h $(LU)
  $(CC) $(CFLAGS) $(EXE) taylor_trig.c $(UTILLIB)\
    $(MATHLIB) -o $@
geometric_trig: $(OUTDIR)/taylor_trig.out

$(OUTDIR)/taylor_inv_trig.out: taylor_inv_trig.c trig_utilities.h \
                  taylor_inv_trig.h $(LU)
  $(CC) $(CFLAGS) $(EXE) taylor_inv_trig.c $(UTILLIB)\
    $(MATHLIB) -o $@
geometric_trig: $(OUTDIR)/taylor_inv_trig.out

$(OUTDIR)/cordic_trig.out: cordic_trig.c trig_utilities.h trig_fixed.h \
                cordic_trig.h $(LU)
  $(CC) $(CFLAGS) $(EXE) cordic_trig.c $(UTILLIB) $(MATHLIB) -o $@
cordic_trig: $(OUTDIR)/cordic_trig.out

## LOGARITHM AND EXPONENTIAL FILES ##
$(OUTDIR)/int_exp.out: int_exp.c int_exp.h $(LU)
  $(CC) $(CFLAGS) $(EXE) int_exp.c $(UTILLIB) $(MPFRLIB) -o $@
int_exp: $(OUTDIR)/int_exp.out

$(OUTDIR)/taylor_exp_log.out: taylor_exp_log.c taylor_exp_log.h $(LU) \
                int_exp.h $(OBJDIR)/int_exp.o \
                log_exp_utilities.h
  $(CC) $(CFLAGS) $(EXE) taylor_exp_log.c $(OBJDIR)/int_exp.o \
    $(UTILLIB) $(MATHLIB) -o $@
taylor_exp_log: $(OUTDIR)/taylor_exp_log.out

$(OUTDIR)/hyperbolic_log.out: hyperbolic_log.c hyperbolic_log.h $(LU) \
                log_exp_utilities.h
  $(CC) $(CFLAGS) $(EXE) hyperbolic_log.c $(UTILLIB) $(MATHLIB) -o $@
hyperbolic_log: $(OUTDIR)/hyperbolic_log.out

$(OUTDIR)/cont_frac_exp.out: cont_frac_exp.c cont_frac_exp.h int_exp.h \
                hyperbolic_log.h $(LU)
  $(CC) $(CFLAGS) $(EXE) cont_frac_exp.c $(OBJDIR)/int_exp.o \
    $(OBJDIR)/hyperbolic_log.o $(UTILLIB) $(MATHLIB) -o $@
cont_frac_exp: $(OUTDIR)/cont_frac_exp.out

## TESTING FILES ##
$(TSTDIR)/test_newton_sqrt.out: $(TSTDIR)/test_newton_sqrt.c \
        $(TSTDIR)/test_utilities.h \
        $(OBJDIR)/newton_sqrt.o $(OBJDIR)/exact_root.o \
        $(LU)
  $(CC) $(CFLAGS) $< $(OBJDIR)/newton_sqrt.o $(OBJDIR)/exact_root.o \
    $(MPFRLIB) $(UTILLIB) -I. -o $@
```

```
143 test_newton_sqrt: $(TSTDIR)/test_newton_sqrt.out
144
145 $(TSTDIR)/test_trig_methods.out: $(TSTDIR)/test_trig_methods.c \
146         $(TSTDIR)/test_utilities.h \
147         $(OBJDIR)/geometric_trig.o $(OBJDIR)/taylor_trig.o \
148         $(OBJDIR)/cordic_trig.o $(LU)
149   $(CC) $(CFLAGS) $< $(OBJDIR)/geometric_trig.o $(OBJDIR)/taylor_trig.o \
150     $(OBJDIR)/cordic_trig.o $(MATHLIB) $(UTILLIB) -I. -o $(@)
151 test_trig_methods: $(TSTDIR)/test_trig_methods.out
152
153 $(TSTDIR)/test_inv_trig_methods.out: $(TSTDIR)/test_inv_trig_methods.c \
154         $(TSTDIR)/test_utilities.h \
155         $(OBJDIR)/geometric_inv_trig.o \
156         $(OBJDIR)/taylor_inv_trig.o \
157         $(OBJDIR)/cordic_trig.o $(LU)
158   $(CC) $(CFLAGS) $< $(OBJDIR)/geometric_inv_trig.o \
159     $(OBJDIR)/taylor_inv_trig.o \
160     $(OBJDIR)/cordic_trig.o $(MATHLIB) $(UTILLIB) -I. -o $(@)
161 test_inv_trig_methods: $(TSTDIR)/test_inv_trig_methods.out
162
163 $(TSTDIR)/test_sqrt_methods.out: $(TSTDIR)/test_sqrt_methods.c \
164         $(TSTDIR)/test_utilities.h \
165         $(OBJDIR)/exact_root.o $(OBJDIR)/bisect_root.o \
166         $(OBJDIR)/newton_sqrt.o $(OBJDIR)/newton_inv_sqrt.o $(LU)
167   $(CC) $(CFLAGS) $< $(OBJDIR)/exact_root.o $(OBJDIR)/bisect_root.o \
168     $(OBJDIR)/newton_sqrt.o $(OBJDIR)/newton_inv_sqrt.o\
169     $(MATHLIB) $(UTILLIB) -I. -o $(@)
170 test_sqrt_methods: $(TSTDIR)/test_sqrt_methods.out
171
172 $(TSTDIR)/test_int_exp_methods.out: $(TSTDIR)/test_int_exp_methods.c \
173         $(TSTDIR)/test_utilities.h $(OBJDIR)/int_exp.o $(LU)
174   $(CC) $(CFLAGS) $< $(OBJDIR)/int_exp.o $(MPFRLIB) $(UTILLIB) -I. -o $@
175 test_int_exp_methods: $(TSTDIR)/test_int_exp_methods.out;
176
177 $(TSTDIR)/test_exp_methods.out: $(TSTDIR)/test_exp_methods.c \
178                 $(TSTDIR)/test_utilities.h \
179                 $(OBJDIR)/taylor_exp_log.o \
180                 $(OBJDIR)/cont_frac_exp.o \
181                 $(OBJDIR)/int_exp.o $(LU)
182   $(CC) $(CFLAGS) $< $(OBJDIR)/taylor_exp_log.o \
183     $(OBJDIR)/cont_frac_exp.o $(OBJDIR)/int_exp.o \
184     $(OBJDIR)/hyperbolic_log.o $(MATHLIB) $(UTILLIB) -I. -o $@
185 test_exp_methods: $(TSTDIR)/test_exp_methods.out
186
187 $(TSTDIR)/test_log_methods.out: $(TSTDIR)/test_log_methods.c \
188                 $(TSTDIR)/test_utilities.h \
189                 $(OBJDIR)/taylor_exp_log.o \
190                 $(OBJDIR)/hyperbolic_log.o \
191                 $(OBJDIR)/int_exp.o $(LU)
192   $(CC) $(CFLAGS) $< $(OBJDIR)/taylor_exp_log.o $(OBJDIR)/int_exp.o\
193     $(OBJDIR)/hyperbolic_log.o $(MATHLIB) $(UTILLIB) -I. -o $@
194 test_log_methods: $(TSTDIR)/test_log_methods.out
195
196 $(TSTDIR)/test_log_pow_methods.out: $(TSTDIR)/test_log_pow_methods.c \
197         $(TSTDIR)/test_utilities.h $(OBJDIR)/taylor_exp_log.o \
198         $(OBJDIR)/hyperbolic_log.o $(OBJDIR)/int_exp.o \
199         $(OBJDIR)/cont_frac_exp.o $(LU)
200   $(CC) $(CFLAGS) $< $(OBJDIR)/taylor_exp_log.o $(OBJDIR)/int_exp.o\
```

```
    $(OBJDIR)/hyperbolic_log.o $(OBJDIR)/cont_frac_exp.o \
    $(MATHLIB) $(UTILLIB) -I. -o $@
test_log_pow_methods: $(TSTDIR)/test_log_pow_methods.out
```

A.2 Square Root Code

Code for Exact Square Root Metods:

File : exact_root.c

```c
#include <gmp.h>
#include <mpfr.h>
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>

#include "utilities.h"
#include "exact_root.h"

char *root_digits_precise(char *N, unsigned int D)
{
  //Counter variables
  unsigned int i, a;
  //The offset value used to set the correct character's value
  unsigned int o = 0;
  //Real and Integer types from GMP and MPFR used for precision
  mpfr_t Yr, Nr,  T, tmpr_0, tmpr_1;
  mpz_t  P,  tmpz, Yz;

  //Allocates memory for the number of digits requested plus 5 to be safe
  char *R = malloc((D+5) * sizeof(*R));

  //Sets Nr from the provided string representing N
  mpfr_init_set_str(Nr, N, 10, MPFR_RNDN);
  mpfr_init(Yr);
  mpfr_init(tmpr_0);
  mpfr_init(tmpr_1);

  //P will be used to keep track of the current partial solution
  mpz_init_set_ui(P, 0);
  mpz_init(Yz);
  mpz_init(tmpz);

  //T represents the power of the 10 that the current digit represents
  //T is initially floor(n/2) where N == K*10^n and K is in [0, 10)
  //T is of the form 10^t
  mpfr_init(T);

  //The mpfr_log10 function is used to help find the exponent (power 10)
  //  of Nr
  mpfr_log10(T, Nr, MPFR_RNDN);
  mpfr_div_ui(T, T, 2, MPFR_RNDN);
  mpfr_floor(T, T);
  //Similar to log, but for exponentiation
  mpfr_exp10(T, T, MPFR_RNDN);

  //This takes into account numbers of the form 0.x
  if(mpfr_cmp_ui(T, 1) < 0)
```

```
49      {
50        R[0] = '0';
51        R[1] = '.';
52        //Offset set to 2 to indicate there are 2 pre-assigned characters
53        o = 2;
54      }
55
56      //Main loop
57      for(i=0; i <= D; i++)
58      {
59        //Calculates 10^(2t) and 20*P
60        mpfr_mul(tmpr_0, T, T, MPFR_RNDN);
61        mpz_mul_ui(tmpz, P, 20);
62        //tmpr_1 is used to prevent re-calculation later on
63        mpfr_set_ui(tmpr_1, 0, MPFR_RNDN);
64
65        /*
66        This loop stops when any digit produces a Y too large or all
67          digits have been conisdered.
68        In both cases a will be one greater than required and thus
69          must be decremented afterwards
70        */
71        for(a=1; a <= 9; a++)
72        {
73          //Calculates N - (20*P + a)*a*10^(2t)
74          mpz_add_ui(Yz, tmpz, a);
75          mpz_mul_ui(Yz, Yz, a);
76          mpfr_mul_z(Yr, tmpr_0, Yz, MPFR_RNDN);
77
78          if(mpfr_cmp(Yr, Nr) > 0)
79            //The exit condition for the loop has been met
80            break;
81          else
82            //tmpr_1 updated to remove the need for re-calculation
83            mpfr_set(tmpr_1, Yr, MPFR_RNDN);
84        }
85
86        //Decrements a and adds the correct digit to the result string
87        R[i+o] = DIGITS[--a];
88
89        //Reduces Nr by the largest Yr found in the previous loop
90        mpfr_sub(Nr, Nr, tmpr_1, MPFR_RNDN);
91
92        //Break if an exact solution is found
93        //Note that due to the representation of floating point numbers it
94        //  is possible to have found an exact solution with a positive
95        //  remainder that is very close to zero. Unfortunately there is
96        //  no way to test for this without knowing, the exact precision
97        //  of the input beforehand.
98        if(mpfr_cmp_ui(Nr, 0) == 0)
99        {
100         //This loop adds 0s to a string where an exact solution has
101         //  been found but needs right padding with zeros.
102         while(mpfr_cmp_ui(T, 1) > 0)
103         {
104           R[++i + o] = '0';
105           mpfr_div_ui(T, T, 10, MPFR_RNDN);
106         }
```

```
107       break;
108     }
109
110     //Calculates P = 10*P + a
111     mpz_mul_ui(P, P, 10);
112     mpz_add_ui(P, P, a);
113     //Calculates T = T/10 => 10^t -> 10^(t-1)
114     mpfr_div_ui(T, T, 10, MPFR_RNDN);
115
116     //If we have dropped below 10^0 for the first time then add
117     //  a '.' to the result string and increase the offset to 1
118     //This case only occurs if no '.' is in the string already
119     if(o == 0 && mpfr_cmp_ui(T, 1) < 0)
120     {
121       o = 1;
122       R[i+o] = '.';
123     }
124   }
125
126   //Adds a null character to terminate the string
127   R[i+o+1] = '\0';
128   return R;
129 }
130
131 //The use of uintmax_t gives the largest number of unsigned integers
132 //  for which this function will work with.
133 uintmax_t uint_sqrt(uintmax_t num)
134 {
135   //Represents the value of 2r(2^m), where r is the
136   //  current known part of the integer root
137   uintmax_t res = 0;
138   //Represents the largest power of (2^m)^2 = 4^m, the initial value
139   //  is calculated as 011...11 XOR 0011...11 as the size
140   //  of uintmax_t is not known beforehand
141   uintmax_t bit = (UINTMAX_MAX >> 1) ^ (UINTMAX_MAX >> 2);
142
143   //Finds the largest power of 4 that is at most 'num' in value
144   while(bit > num)
145     bit >>= 2;
146
147   //while(bit) is equivalent to while(bit > 0) for unsigned integers
148   while(bit)
149   {
150     //Checks the two cases for updating 'res' and 'num'
151     if(num >= res + bit)
152     {
153       //'num' is used to keep track of the difference betweek
154       //  r and the original value, N, that was to be rooted.
155       num -= res + bit;
156       //This calculates 'res' -> 2(r + 2^m)*2^(m-1) using addition
157       //  and bitshifting
158       res = (res >> 1) + bit;
159     }
160     //In the other case 'res' -> 2r(2^m-1)
161     else
162       res >>= 1;
163
164     //Move on to the next lower power of 2
```

```
    bit >>= 2;
  }

  //Returns the integer part of the square root
  return res;
}

#ifdef COMPILE_MAIN
int main(int argc, char **argv)
{
  uintmax_t N;
  unsigned int p, d;
  char *R;

  if (argc == 1)
  {
    printf("Usage: %s [a/b] [arguments]", argv[0]);
    exit(1);
  }

  switch(argv[1][0])
  {
    case 'a':
      if(argc == 5 &&
        sscanf(argv[3], "%u", &d) == 1 &&
        sscanf(argv[4], "%u", &p) == 1)
      {
        mpfr_set_default_prec(p);
        printf("sqrt(%s) ~=\n\t%s", argv[2],
          root_digits_precise(argv[2], d));
      }
      else
        printf("Usage: %s a <N=Number to sqrt> "
             "<d=Number of significant digits> "
             "<p=bits of precision to use>\n", argv[0]);
      break;

    case 'b':
      if(argc == 3 &&
        sscanf(argv[2], "%" SCNuMAX, &N) == 1)
        printf("int_sqrt(%" PRIuMAX ") ~= %" PRIuMAX "\n",
            N, uint_sqrt(N));
      else
        printf("Usage: %s b <N=Positive integer to sqrt>\n",
            argv[0]);
      break;

    default:
      printf("Usage: %s [a/b] [arguments]", argv[0]);
  }
}
#endif
```

Code for the Bisection Methods:

File : bisect_root.c

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <gmp.h>
#include <mpfr.h>
#include <assert.h>
#include <math.h>

#include "bisect_root.h"
#include "utilities.h"

#define INIT_CONSTANTS mpfr_init_set_ui(MPFR_ONE, 1, MPFR_RNDN); \
                mpfr_init_set_d(MPFR_HALF, 0.5, MPFR_RNDN);

mpfr_t MPFR_ONE, MPFR_HALF;

double bisect_sqrt(double N, double T)
{
  assert(N >= 0);
  assert(T >= 0);

  int e;
  double a, b, x, f;

  //frexp finds a,b such that a*2^b = N and 1/2 <= a < 1
  N = frexp(N, &e);
  //Corrects for the case of odd exponents
  //  e%2 is true when e is odd
  if(e%2)
  {
    N /= 2;
    e += 1;
  }

  //Sets the initial values of a and b
  a = 0;
  b = 1;

  x = 0.5*(a + b);
  f = x*x - N;

  //fabs(f) > T is our approximation of
  //  f != 0, by using the given tolerance
  while(fabs(f) > T && b - a > T)
  {
    //Update of the bounds a and b
    if (f < 0)
      a = x;
    else
      b = x;

    //Update of x and f
    x = 0.5*(a + b);
    f = x*x - N;
  }

  return ldexp(x, e / 2);
}

double bisect_sqrt_it(double N, unsigned int I)
{
```

```
61    assert(N >= 0);
62
63    int e;
64    double a, b, x, f;
65
66    //frexp finds a,b such that a*2^b = N and 1/2 <= a < 1
67    N = frexp(N, &e);
68    //Corrects for the case of odd exponents
69    //  e%2 is true when e is odd
70    if(e%2)
71    {
72      N /= 2;
73      e += 1;
74    }
75
76    //Sets the initial values of a and b
77    a = 0;
78    b = 1;
79
80    x = 0.5*(a + b);
81    f = x*x - N;
82
83    //fabs(f) > T is our approximation of
84    //  f != 0, by using the given tolerance
85    for(int i = 0; i < I; ++i)
86    {
87      //Update of the bounds a and b
88      if (f < 0)
89        a = x;
90      else
91        b = x;
92
93      //Update of x and f
94      x = 0.5*(a + b);
95      f = x*x - N;
96    }
97
98    return ldexp(x, e / 2);
99  }
100
101 double iPow(double x, unsigned int n)
102 {
103   double r = 1;
104   while(n--)
105     r *= x;
106   return r;
107 }
108
109 double bisect_nRoot(double N, double T, unsigned int n)
110 {
111   assert(N >= 0);
112   assert(T >= 0);
113   //Ensures that none of the trivial cases are requested
114   assert(n >= 2);
115
116   //Runs the more optimal bisect_sqrt if n == 2
117   if(n == 2)
118     return bisect_sqrt(N, T);
```

```
119
120   double a, b, x, f;
121
122   //Sets the initial values of a and b
123   a = 0;
124   //This statement is equivalent to
125   //  if(N<1) b=1; else b=N;
126   b = N < 1 ? 1 : N;
127
128   x = 0.5*(a + b);
129   f = iPow(x, n) - N;
130
131   //fabs(f) > T is our approximation of
132   //  f != 0, by using the given tolerance
133   while(fabs(f) > T && b - a > T)
134   {
135     //Update of the bounds a and b
136     if (f < 0)
137       a = x;
138     else
139       b = x;
140
141     //Update of x and f
142     x = 0.5*(a + b);
143     f = iPow(x, n) - N;
144   }
145
146   return x;
147 }
148
149 void mpfr_bisect_sqrt(mpfr_t R, mpfr_t N, mpfr_t T)
150 {
151   if(mpfr_cmp_ui(N, 0) < 0)
152   {
153     fprintf(stderr, "The value to square root must be non-negative\n");
154     exit(-1);
155   }
156   if(mpfr_cmp_ui(T, 0) < 0)
157   {
158     fprintf(stderr, "The tolerance must be non-negative\n");
159     exit(-1);
160   }
161
162   mpfr_exp_t e;
163   mpfr_t a, b, x, f, d, fab, n;
164
165   mpfr_init(n);
166   mpfr_frexp(&e, n, N, MPFR_RNDN);
167   if(e%2)
168   {
169     mpfr_div_ui(n, n, 2, MPFR_RNDN);
170     e += 1;
171   }
172
173   //Set a == 0
174   mpfr_init_set_ui(a, 0, MPFR_RNDN);
175
176   //Set b == 1
```

```
177     mpfr_init_set_ui(b, 1, MPFR_RNDN);
178
179     //Set x = (a + b)/2
180     mpfr_init(x);
181     mpfr_add(x, a, b, MPFR_RNDN);
182     mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);
183
184     //Set f = x^2 - N and fab = |f|
185     mpfr_init(f);
186     mpfr_init(fab);
187     mpfr_mul(f, x, x, MPFR_RNDN);
188     mpfr_sub(f, f, N, MPFR_RNDN);
189     mpfr_abs(fab, f, MPFR_RNDN);
190
191     //Set d = b - a
192     mpfr_init(d);
193     mpfr_sub(d, b, a, MPFR_RNDN);
194
195     while(mpfr_cmp(fab, T) > 0 && mpfr_cmp(d, T) > 0)
196     {
197       //Update the bounds, a and b
198       if(mpfr_cmp_ui(f, 0) < 0)
199         mpfr_set(a, x, MPFR_RNDN);
200       else
201         mpfr_set(b, x, MPFR_RNDN);
202
203       //Update x
204       mpfr_add(x, a, b, MPFR_RNDN);
205       mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);
206
207       //Update f and fab
208       mpfr_mul(f, x, x, MPFR_RNDN);
209       mpfr_sub(f, f, n, MPFR_RNDN);
210       mpfr_abs(fab, f, MPFR_RNDN);
211     }
212
213     printf("beep");
214     mpfr_mul_2si(R, x, e/2, MPFR_RNDN);
215 }
216
217 void mpfr_bisect_nRoot(mpfr_t R, mpfr_t N, mpfr_t T, unsigned int n)
218 {
219     if(mpfr_cmp_ui(N, 0) < 0)
220     {
221       fprintf(stderr, "The value to square root must be non-negative\n");
222       exit(-1);
223     }
224     if(mpfr_cmp_ui(T, 0) < 0)
225     {
226       fprintf(stderr, "The tolerance must be non-negative\n");
227       exit(-1);
228     }
229     assert(n >= 2);
230
231     mpfr_t a, b, x, f, d, fab;
232
233     //Set a == 0
234     mpfr_init_set_ui(a, 0, MPFR_RNDN);
```

```
235
236     //Set b = max{1, N}
237     mpfr_init(b);
238     mpfr_max(b, MPFR_ONE, N, MPFR_RNDN);
239
240     //Set x = (a + b)/2
241     mpfr_init(x);
242     mpfr_add(x, a, b, MPFR_RNDN);
243     mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);
244
245     //Set f = x^2 - N
246     mpfr_init(f);
247     mpfr_init(fab);
248     mpfr_pow_ui(f, x, n, MPFR_RNDN);
249     mpfr_sub(f, f, N, MPFR_RNDN);
250     mpfr_abs(fab, f, MPFR_RNDN);
251
252     //Set d = b - a
253     mpfr_init(d);
254     mpfr_sub(d, b, a, MPFR_RNDN);
255
256     while(mpfr_cmp(fab, T) > 0 && mpfr_cmp(d, T) > 0)
257     {
258       //Update the bounds, a and b
259       if(mpfr_cmp_ui(f, 0) < 0)
260         mpfr_set(a, x, MPFR_RNDN);
261       else
262         mpfr_set(b, x, MPFR_RNDN);
263
264       //Update x
265       mpfr_add(x, a, b, MPFR_RNDN);
266       mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);
267
268       //Update f
269       mpfr_pow_ui(f, x, n, MPFR_RNDN);
270       mpfr_sub(f, f, N, MPFR_RNDN);
271       mpfr_abs(fab, f, MPFR_RNDN);
272     }
273
274     mpfr_set(R, x, MPFR_RNDN);
275 }
276
277 #ifdef COMPILE_MAIN
278 int main(int argc, char** argv)
279 {
280     double N, T;
281     unsigned int n, D, p;
282     mpfr_t Nr, Tr, R;
283     int c;
284     char sf[50];
285
286     if (argc == 1)
287     {
288       printf("Usage: %s [a/b/c/d/e] [arguments]\n", argv[0]);
289       exit(1);
290     }
291
292     switch(argv[1][0])
```

```
293 {
294   case 'a':
295     if (argc == 5 &&
296         sscanf(argv[2], "%lf", &N) == 1 &&
297         sscanf(argv[3], "%lf", &T) == 1 &&
298         sscanf(argv[4], "%u", &D) == 1)
299       printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, bisect_sqrt(N, T));
300     else
301       printf("Usage: %s a <N=Value to sqrt> "
302            "<T=Tolerance> <D=Number of digits to display>\n",
303            argv[0]);
304     break;
305
306   case 'b':
307     if (argc == 6 &&
308         sscanf(argv[2], "%lf", &N) == 1 &&
309         sscanf(argv[3], "%lf", &T) == 1 &&
310         sscanf(argv[4], "%u", &n) == 1 &&
311         sscanf(argv[5], "%u", &D) == 1)
312       printf("%u_Root(%.*lf) =~ %.*lf\n",
313           n, d(D), N, D, bisect_nRoot(N, T, n));
314     else
315       printf("Usage: %s b <N=Value to root> <T=Tolerance>"
316            "<n=nth Root> <D=Number of digits to display>\n",
317            argv[0]);
318     break;
319
320   case 'c':
321     if (argc == 5 &&
322         sscanf(argv[3], "%u", &D) == 1 &&
323         sscanf(argv[4], "%u", &p) == 1)
324     {
325       mpfr_set_default_prec(p);
326       INIT_CONSTANTS
327
328       if (mpfr_init_set_str(Nr, argv[2], 10, MPFR_RNDN) == 0)
329       {
330         mpfr_init(R);
331         //Sets the tolerance to Tr = 10^-D
332         mpfr_digits_to_tolerance(D, Tr);
333
334         //Generates the required format string
335         sprintf(sf, "sqrt(%%.%uRNf) =~\n\t%%.%uRNf\n", d(D), D);
336
337         mpfr_bisect_sqrt(R, Nr, Tr);
338         mpfr_printf(sf, Nr, R);
339       }
340       else
341         printf("Usage: %s c <N=Value to sqrt> "
342              "<D=Number of digits to calculate to> "
343            "<p=bits of precision>\n", argv[0]);
344     }
345     else
346       printf("Usage: %s c <N=Value to sqrt> "
347            "<D=Number of digits to calculate to> "
348          "<p=bits of precision>\n", argv[0]);
349     break;
350
```

```
    case 'd':
      if (argc == 6 &&
          sscanf(argv[3], "%u", &D) == 1 &&
          sscanf(argv[4], "%u", &n) == 1 &&
          sscanf(argv[5], "%u", &p) == 1)
      {
        mpfr_set_default_prec(p);
        INIT_CONSTANTS

        if (mpfr_init_set_str(Nr, argv[2], 10, MPFR_RNDN) == 0)
        {
          mpfr_init(R);

          //Sets the tolerance to Tr = 10^-D
          mpfr_digits_to_tolerance(D, Tr);

          //Generates the required format string
          sprintf(sf, "%%u_Root(%%.%uRNf) =~\n\t%%.%uRNf\n", d(D), D);

          mpfr_bisect_nRoot(R, Nr, Tr, n);
          mpfr_printf(sf, n, Nr, R);
        }
        else
          printf("Usage: %s d <N=Value to root> "
                 "<D=Number of digits to calculate to> "
                "<n=nth root> <p=bits of precision>\n", argv[0]);
      }
      else
        printf("Usage: %s d <N=Value to root> "
               "<D=Number of digits to calculate to> "
              "<n=nth root> <p=bits of precision>\n", argv[0]);
      break;

    case 'e':
      if (argc == 5 &&
          sscanf(argv[2], "%lf", &N) == 1 &&
          sscanf(argv[3], "%u", &p) == 1 &&
          sscanf(argv[4], "%u", &D) == 1)
        printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D,
                          bisect_sqrt_it(N, p));
      else
        printf("Usage: %s a <N=Value to sqrt> "
             "<I=iterartions> <D=Number of digits to display>\n",
              argv[0]);
      break;
    default:
      printf("Usage: %s [a/b/c/d/e] [arguments]", argv[0]);
  }
}
#endif
```

Code for Newton Square Root Methods:

File : newton_root.c

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
```

```
5  #include <assert.h>
6  #include <math.h>
7
8  #include "utilities.h"
9  #include "exact_root.h"
10 #include "newton_sqrt.h"
11
12 #define INIT_CONSTANTS mpfr_init_set_d(MPFR_HALF, 0.5, MPFR_RNDN); \
13              in = fopen(ROOT_2_INFILE, "r"); \
14              mpfr_init(MPFR_ROOT_2); \
15              mpfr_inp_str(MPFR_ROOT_2, in, 10, MPFR_RNDN); \
16              fclose(in); \
17              in = fopen(ROOT_2_INV_INFILE, "r"); \
18              mpfr_init(MPFR_ROOT_2_INV); \
19              mpfr_inp_str(MPFR_ROOT_2_INV, in, 10, MPFR_RNDN); \
20              fclose(in);
21
22 mpfr_t MPFR_ROOT_2, MPFR_ROOT_2_INV, MPFR_HALF;
23
24 double newton_sqrt_v1(double N, double T)
25 {
26   assert(N >= 0);
27   assert(T >= 0);
28
29   double x, px, d;
30
31   //sets the initial guess for x
32   x = N > 1 ? N : 1;
33   //initial iterative error set to a high value to ensure it is larger
34   //  than the given tolerance, provided T is a reasonable tolerance.
35   d = 1000000;
36
37   //Continues while the current iterative error is more than the
38   //  provided tolerance.
39   while(d > T)
40   {
41     //Updates store of the previously approximation
42     px = x;
43     //Calculates the next approximation
44     x = 0.5 * (x + N/x);
45     //Updates the iterative error
46     d = fabs(x - px);
47   }
48   return x;
49 }
50
51 double newton_sqrt_v2(double N, double T)
52 {
53   assert(N >= 0);
54   assert(T >= 0);
55
56   double x, px, d;
57
58   //Sets the initial guess as explained in the report section on
59   //  Newton Square Root approximations
60   if(N >= 4)
61     x = uint_sqrt((unsigned long) N);
62   else
```

```
63      x = N > 1 ? N : 1;
64
65    //initial iterative error set to a high value to ensure it is larger
66    //  than the given tolerance, provided T is a reasonable tolerance.
67    d = 1000000;
68
69    //Continues while the current iterative error is more than the
70    //  provided tolerance.
71    while(d > T)
72    {
73      //Updates store of the previously approximation
74      px = x;
75      //Calculates the next approximation
76      x = 0.5 * (x + N/x);
77      //Updates the iterative error
78      d = fabs(x - px);
79    }
80
81    return x;
82  }
83
84  double newton_sqrt_v3(double N, double T)
85  {
86    assert(N >= 0);
87    assert(T >= 0);
88
89    int e;
90    double x, px, d;
91
92    //frexp ensures 1/2 <= N < 1 and N*2^e = input N
93    N = frexp(N, &e);
94
95    //Initial approximation is now a fixed constant
96    x = 1;
97    //initial iterative error set to a high value to ensure it is larger
98    //  than the given tolerance, provided T is a reasonable tolerance.
99    d = 1000000;
100
101   //Continues while the current iterative error is more than the
102   //  provided tolerance.
103   while(d > T)
104   {
105     //Updates store of the previously approximation
106     px = x;
107     //Calculates the next approximation
108     x = 0.5 * (x + N/x);
109     //Updates the iterative error
110     d = fabs(x - px);
111   }
112
113   //If e is odd then the result must be multiplied by either sqrt(2) or
114   //  1/sqrt(2) to give the correct approximation
115   if(e%2)
116     x *= e > 0 ? ROOT_2 : ROOT_2_INV;
117   //return x*2^(e/2)
118   return ldexp(x, e / 2);
119 }
120
```

```
121 double newton_sqrt_v3_it(double N, unsigned int I)
122 {
123   assert(N >= 0);
124
125   int e;
126   double x;
127
128   N = frexp(N, &e);
129
130   x = 1;
131
132   for(int i = 0; i < I; ++i)
133     x = 0.5 * (x + N/x);
134
135   if(e%2)
136     x *= e > 0 ? ROOT_2 : ROOT_2_INV;
137   return ldexp(x, e / 2);
138 }
139
140
141 void mpfr_newton_sqrt_v3(mpfr_t R, mpfr_t N, mpfr_t T)
142 {
143   mpfr_t x, px, d, t, n;
144   mpfr_exp_t e;
145
146   mpfr_init(n);
147   mpfr_frexp(&e, n, N, MPFR_RNDN);
148
149   mpfr_init_set_ui(x, 1, MPFR_RNDN);
150   mpfr_init(px);
151   mpfr_init_set_ui(d, 1000000, MPFR_RNDN);
152   mpfr_init(t);
153
154   while(mpfr_cmp(d, T) > 0)
155   {
156     mpfr_set(px, x, MPFR_RNDN);
157     mpfr_div(t, n, x, MPFR_RNDN);
158     mpfr_add(x, x, t, MPFR_RNDN);
159     mpfr_mul(x, MPFR_HALF, x, MPFR_RNDN);
160     mpfr_sub(d, x, px, MPFR_RNDN);
161     mpfr_abs(d, d, MPFR_RNDN);
162   }
163
164   if(e%2)
165     if(e > 0)
166       mpfr_mul(x, MPFR_ROOT_2, x, MPFR_RNDN);
167     else
168       mpfr_mul(x, MPFR_ROOT_2_INV, x, MPFR_RNDN);
169   mpfr_mul_2si(R, x, e/2, MPFR_RNDN);
170 }
171
172 #ifdef COMPILE_MAIN
173 int main(int argc, char **argv)
174 {
175   double N, T;
176   unsigned int n, D, p;
177   mpfr_t Nr, Tr, R;
178   int c;
```

```
179   char sf[50];
180   FILE *in;
181
182   if(argc==1)
183   {
184     printf("Usage: %s [a/b/c/d/e] <Arguments>\n", argv[0]);
185     exit(1);
186   }
187
188   switch(argv[1][0])
189   {
190     case 'a':
191       if (argc == 5 &&
192           sscanf(argv[2], "%lf", &N) == 1 &&
193           sscanf(argv[3], "%lf", &T) == 1 &&
194           sscanf(argv[4], "%u" , &D) == 1)
195         printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_sqrt_v1(N, T));
196       else
197         printf("Usage: %s a <N=Value to sqrt> "
198             "<T=Tolerance> <D=Number of digits to display>\n",
199             argv[0]);
200       break;
201
202     case 'b':
203       if (argc == 5 &&
204           sscanf(argv[2], "%lf", &N) == 1 &&
205           sscanf(argv[3], "%lf", &T) == 1 &&
206           sscanf(argv[4], "%u" , &D) == 1)
207         printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_sqrt_v2(N, T));
208       else
209         printf("Usage: %s b <N=Value to sqrt> "
210             "<T=Tolerance> <D=Number of digits to display>\n",
211             argv[0]);
212       break;
213
214     case 'c':
215       if (argc == 5 &&
216           sscanf(argv[2], "%lf", &N) == 1 &&
217           sscanf(argv[3], "%lf", &T) == 1 &&
218           sscanf(argv[4], "%u" , &D) == 1)
219         printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_sqrt_v3(N, T));
220       else
221         printf("Usage: %s c <N=Value to sqrt> "
222             "<T=Tolerance> <D=Number of digits to display>\n",
223             argv[0]);
224       break;
225
226     case 'd':
227       if (argc == 5 &&
228           sscanf(argv[3], "%u", &D) == 1 &&
229           sscanf(argv[4], "%u" , &p) == 1)
230       {
231         mpfr_set_default_prec(p);
232         INIT_CONSTANTS
233
234         if (mpfr_init_set_str(Nr, argv[2], 10, MPFR_RNDN) == 0)
235         {
236           mpfr_init(R);
```

```
237
238             mpfr_digits_to_tolerance(D, Tr);
239
240             sprintf(sf, "sqrt(%%.%uRNf) =~\n\t%%.%uRNf\n", d(D), D);
241
242             mpfr_newton_sqrt_v3(R, Nr, Tr);
243             mpfr_printf(sf, Nr, R);
244           }
245           else
246             printf("Usage: %s d <N=Value to sqrt> "
247                  "<D=Number of digitsto calculate to> "
248                  "<p=bits of precision>\n", argv[0]);
249         }
250         else
251           printf("Usage: %s d <N=Value to sqrt> "
252                "<D=Number of digitsto calculate to> "
253                "<p=bits of precision>\n", argv[0]);
254         break;
255
256       case 'e':
257         if (argc == 5 &&
258             sscanf(argv[2], "%lf", &N) == 1 &&
259             sscanf(argv[3], "%u",  &p) == 1 &&
260             sscanf(argv[4], "%u" , &D) == 1)
261           printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D,
262                           newton_sqrt_v3_it(N, p));
263         else
264           printf("Usage: %s e <N=Value to sqrt> "
265                "<I=Iterations> <D=Number of digits to display>\n",
266                argv[0]);
267         break;
268       default:
269         printf("Usage: %s [a/b/c/d/e] <Arguments>\n", argv[0]);
270     }
271 }
272 #endif
```

Code for Newton Inverse Square Root Methods:

File : newton_inv_sqrt.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <gmp.h>
4  #include <mpfr.h>
5  #include <assert.h>
6  #include <math.h>
7
8  #include "utilities.h"
9  #include "newton_inv_sqrt.h"
10
11 #define INIT_CONSTANTS mpfr_init_set_d(MPFR_THREE_HALF, 1.5, MPFR_RNDN); \
12                 in = fopen(ROOT_2_INFILE, "r"); \
13                 mpfr_init(MPFR_ROOT_2); \
14                 mpfr_inp_str(MPFR_ROOT_2, in, 10, MPFR_RNDN); \
15                 fclose(in); \
16                 in = fopen(ROOT_2_INV_INFILE, "r"); \
17                 mpfr_init(MPFR_ROOT_2_INV); \
18                 mpfr_inp_str(MPFR_ROOT_2_INV, in, 10, MPFR_RNDN); \
```

```
19              fclose(in);
20
21  mpfr_t MPFR_ROOT_2, MPFR_ROOT_2_INV, MPFR_THREE_HALF;
22
23  double newton_inv_sqrt(double N, double T)
24  {
25    assert(N >= 0);
26    assert(T >= 0);
27
28    int e;
29    double x, px, d, NN, N_2;
30
31    //Keeps the initial input N, required at the end.
32    NN  = N;
33    //frexp ensures that 1/2 <= N < 1 and N*2^e == NN
34    N   = frexp(N, &e);
35    //Pre-calculates N/2
36    N_2 = 0.5*N;
37
38    //Sets initial approximation as a constant and initial iterative
39    //  error to be a high value
40    x = 1;
41    d = 1000000;
42
43    //Continues while the iterative error is still too large
44    while(d > T)
45    {
46      //Updates the previously calculated approximation
47      px = x;
48      //Performs the update step of the method
49      x = x * (1.5 - N_2*x*x);
50      //Updates the iterative error
51      d = fabs(x - px);
52    }
53
54    //Corrects for the case when e is odd
55    if(e%2)
56      x *= e > 0 ? ROOT_2_INV : ROOT_2;
57    //x == 1/sqrt(N), so x*NN == NN/(2^(e/2)*N) == NN/sqrt(NN) == sqrt(NN)
58    x *= NN;
59    return ldexp(x, -e / 2);
60  }
61
62  double newton_inv_sqrt_it(double N, unsigned int I)
63  {
64    assert(N >= 0);
65
66    int e;
67    double x, NN, N_2;
68
69    NN  = N;
70    N   = frexp(N, &e);
71    N_2 = 0.5*N;
72
73    x = 1;
74
75    for(int i = 0; i < I; ++i)
76      x = x * (1.5 - N_2*x*x);
```

```
77
78    if(e%2)
79      x *= e > 0 ? ROOT_2_INV : ROOT_2;
80    x *= NN;
81    return ldexp(x, -e / 2);
82  }
83
84  void mpfr_newton_inv_sqrt(mpfr_t R, mpfr_t N, mpfr_t T)
85  {
86    mpfr_t x, px, d, t, n, n_2;
87    mpfr_exp_t e;
88
89    mpfr_init(n);
90    mpfr_frexp(&e, n, N, MPFR_RNDN);
91    mpfr_init_set(n_2, n, MPFR_RNDN);
92    mpfr_div_ui(n_2, n, 2, MPFR_RNDN);
93
94    mpfr_init_set_ui(x, 1, MPFR_RNDN);
95    mpfr_init(px);
96    mpfr_init_set_ui(d, 1000000, MPFR_RNDN);
97    mpfr_init(t);
98
99    while(mpfr_cmp(d, T) > 0)
100   {
101     mpfr_set(px, x, MPFR_RNDN);
102     mpfr_mul(t, x, x, MPFR_RNDN);
103     mpfr_mul(t, t, n_2, MPFR_RNDN);
104     mpfr_sub(t, MPFR_THREE_HALF, t, MPFR_RNDN);
105     mpfr_mul(x, x, t, MPFR_RNDN);
106     mpfr_sub(d, x, px, MPFR_RNDN);
107     mpfr_abs(d, d, MPFR_RNDN);
108   }
109
110   if(e%2)
111     if(e > 0)
112       mpfr_mul(x, MPFR_ROOT_2_INV, x, MPFR_RNDN);
113     else
114       mpfr_mul(x, MPFR_ROOT_2, x, MPFR_RNDN);
115   mpfr_mul(x, x, N, MPFR_RNDN);
116   mpfr_mul_2si(R, x, -e/2, MPFR_RNDN);
117 }
118
119 #ifdef COMPILE_MAIN
120 int main(int argc, char **argv)
121 {
122   double N, T;
123   unsigned int n, D, p;
124   mpfr_t Nr, Tr, R;
125   int c;
126   char sf[50];
127   FILE *in;
128
129   if(argc == 1)
130   {
131     printf("Usage: %s [a/b/c] <Arguments>\n", argv[0]);
132     exit(1);
133   }
134
```

```
135   switch(argv[1][0])
136   {
137     case 'a':
138       if (argc == 5 &&
139           sscanf(argv[2], "%lf", &N) == 1 &&
140           sscanf(argv[3], "%lf", &T) == 1 &&
141           sscanf(argv[4], "%u" , &D) == 1)
142         printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_inv_sqrt(N, T));
143       else
144         printf("Usage: %s a <N=Value to sqrt> "
145             "<T=Tolerance> <D=Number of digits to display>\n",
146             argv[0]);
147       break;
148
149     case 'b':
150       if (argc == 5 &&
151           sscanf(argv[3], "%u", &D) == 1 &&
152           sscanf(argv[4], "%u", &p) == 1)
153       {
154         mpfr_set_default_prec(p);
155         INIT_CONSTANTS
156
157         if(mpfr_init_set_str(Nr, argv[2], 10, MPFR_RNDN) == 0)
158         {
159           mpfr_init(R);
160
161           mpfr_digits_to_tolerance(D, Tr);
162
163           sprintf(sf, "sqrt(%%.%uRNf) =~\n\t%%.%uRNf\n", d(D), D);
164
165           mpfr_newton_inv_sqrt(R, Nr, Tr);
166           mpfr_printf(sf, Nr, R);
167         }
168         else
169           printf("Usage: %s b <N=Value to sqrt> "
170               "<D=Number of digits to calculate to> "
171               "<p=bits of precision>\n", argv[0]);
172       }
173       else
174         printf("Usage: %s b <N=Value to sqrt> "
175             "<D=Number of digits to calculate to> "
176             "<p=bits of precision>\n", argv[0]);
177       break;
178
179     case 'c':
180       if (argc == 5 &&
181           sscanf(argv[2], "%lf", &N) == 1 &&
182           sscanf(argv[3], "%u",  &p) == 1 &&
183           sscanf(argv[4], "%u" , &D) == 1)
184         printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D,
185                         newton_inv_sqrt_it(N, p));
186       else
187         printf("Usage: %s a <N=Value to sqrt> "
188             "<I=iterations> <D=Number of digits to display>\n",
189             argv[0]);
190       break;
191
192     default:
```

```
193         printf("Usage: %s [a/b/c] <Arguments>\n", argv[0]);
194     }
195 }
196 #endif
```

Header files for Square Root Code:

File : exact_root.h

```
1 #ifndef EXACT_ROOT_HEADER
2 #define EXACT_ROOT_HEADER
3   #include <inttypes.h>
4
5   static const char *DIGITS = "0123456789";
6
7   char *root_digits_precise(char*, unsigned int);
8   uintmax_t uint_sqrt(uintmax_t);
9 #endif
```

File : bisect_root.h

```
1 #ifndef BISECT_ROOT_HEADER
2   #define BISECT_ROOT_HEADER
3
4   double bisect_sqrt(double, double);
5   double bisect_sqrt_it(double, unsigned int);
6   double ipow(double, unsigned int);
7   double bisect_nRoot(double, double, unsigned int);
8   void mpfr_bisect_sqrt(mpfr_t, mpfr_t, mpfr_t);
9   void mpfr_bisect_nRoot(mpfr_t, mpfr_t, mpfr_t, unsigned int);
10 #endif
```

File : newton_root.h

```
1 #ifndef NEWTON_ROOT_HEADER
2   #define NEWTON_ROOT_HEADER
3
4   double newton_sqrt_v1(double, double);
5   double newton_sqrt_v2(double, double);
6   double newton_sqrt_v3(double, double);
7   double newton_sqrt_v3_it(double, unsigned int);
8   void mpfr_newton_sqrt_v3(mpfr_t, mpfr_t, mpfr_t);
9 #endif
```

File : newton_inv_sqrt.h

```
1 #ifndef NEWTON_INV_SQRT_HEADER
2 #define NEWTON_INV_SQRT_HEADER
3
4   double newton_inv_sqrt(double, double);
5   double newton_inv_sqrt_it(double, unsigned int);
6   void mpfr_newton_inv_sqrt(mpfr_t, mpfr_t, mpfr_t);
7 #endif
```

A.3 Trigonometric Code

Code for Geometric Trigonometric Functions:

File : geometric_trig.c

```c
#include <stdio.h>
#include <assert.h>
#include <math.h>
#include <gmp.h>
#include <mpfr.h>

#include "geometric_trig.h"
#include "trig_utilities.h"
#include "utilities.h"

#define INIT_CONSTANTS in = fopen(PI_INFILE, "r"); \
              mpfr_init(MPFR_PI); \
              mpfr_inp_str(MPFR_PI, in, 10, MPFR_RNDN); \
              fclose(in); \
              mpfr_init(MPFR_TWO_PI); \
              mpfr_init(MPFR_HALF_PI); \
              mpfr_div_ui(MPFR_HALF_PI, MPFR_PI, 2, MPFR_RNDN); \
              mpfr_mul_ui(MPFR_TWO_PI, MPFR_PI, 2, MPFR_RNDN);

mpfr_t MPFR_PI, MPFR_HALF_PI, MPFR_TWO_PI;

double geometric_cos_bounded(double x, unsigned int n)
{
  //Ensures that x is in the range [0, HALF_PI) and raises an error
  //  message if this is not the case.
  assert(x >= 0 && x <= HALF_PI);

  //Sets the first chord length that will be the basis or our induction
  double h = (x*x)/pow(4, n);

  //Performs the induction steps
  for(int i = 0; i < n; i++)
    h = h*(4-h);
  //Returns the approximation of cos(x)
  return 1 - h/2;
}

void mpfr_geometric_cos_bounded(mpfr_t R, mpfr_t x, unsigned int n)
{
  assert(mpfr_cmp_ui(x, 0) >= 0 && mpfr_cmp(x, MPFR_HALF_PI) <= 0);

  mpfr_t h, t;
  mpz_t k;

  mpfr_init(t);

  mpfr_init(h);
  mpfr_mul(h, x, x, MPFR_RNDN);
  mpz_init(k);
  mpz_ui_pow_ui(k, 4, n);
  mpfr_div_z(h, h, k, MPFR_RNDN);


  for(int i = 0; i < n; i++)
  {
    mpfr_ui_sub(t, 4, h, MPFR_RNDN);
    mpfr_mul(h, h, t, MPFR_RNDN);
```

```
58    }
59
60    mpfr_div_ui(h, h, 2, MPFR_RNDN);
61    mpfr_ui_sub(R, 1, h, MPFR_RNDN);
62  }
63
64
65  double geometric_cos(double x, unsigned int n)
66  {
67    // We have two cases to consider, x >= 0 and x < 0
68    if(x >= 0)
69    {
70      //Ensures x is in the range [0, TWO_PI) as ;
71      //  cos(x + TWO_PI) == cos(x)
72      while(x >= TWO_PI)
73        x -= TWO_PI;
74
75      //Calcualtes the correct modification of x to accurately
76      //  calculate cos(x) when it is reduced to the range [0, HALF_PI)
77      if(x >= PI)
78        if(x - PI >= HALF_PI)
79          return geometric_cos_bounded(TWO_PI - x, n);
80        else
81          return -1 * geometric_cos_bounded(x - PI, n);
82      else
83        if(x >= HALF_PI)
84          return -1 * geometric_cos_bounded(PI - x, n);
85        else
86          return geometric_cos_bounded(x, n);
87    }
88
89    //cos(x) = cos(-x) in the second case
90    return geometric_cos(-x, n);
91  }
92
93  void mpfr_geometric_cos(mpfr_t R, mpfr_t x, unsigned int n)
94  {
95    mpfr_t y, t;
96    mpfr_init_set(y, x, MPFR_RNDN);
97    mpfr_init(t);
98
99    if(mpfr_cmp_ui(y, 0) >= 0)
100   {
101     while(mpfr_cmp(y, MPFR_TWO_PI) >= 0)
102       mpfr_sub(y, y, MPFR_TWO_PI, MPFR_RNDN);
103
104     if(mpfr_cmp(y, MPFR_PI) >= 0)
105     {
106       mpfr_sub(t, y, MPFR_PI, MPFR_RNDN);
107       if(mpfr_cmp(t, MPFR_HALF_PI) >= 0)
108       {
109         mpfr_sub(y, MPFR_TWO_PI, y, MPFR_RNDN);
110         mpfr_geometric_cos_bounded(R, y ,n);
111       }
112       else
113       {
114         mpfr_sub(y, y, MPFR_PI, MPFR_RNDN);
115         mpfr_geometric_cos_bounded(R, y, n);
```

```
116           mpfr_neg(R, R, MPFR_RNDN);
117         }
118       }
119       else
120       {
121         if(mpfr_cmp(y, MPFR_HALF_PI) >= 0)
122         {
123           mpfr_sub(y, MPFR_PI, y, MPFR_RNDN);
124           mpfr_geometric_cos_bounded(R, y, n);
125           mpfr_neg(R, R, MPFR_RNDN);
126         }
127         else
128         {
129           mpfr_geometric_cos_bounded(R, y, n);
130         }
131       }
132     }
133     else
134     {
135       mpfr_neg(y, y, MPFR_RNDN);
136       mpfr_geometric_cos(R, y, n);
137     }
138   }
139
140   //sin(x) = cos(x - HALF_PI)
141   double geometric_sin(double x, unsigned int n)
142   {
143     return geometric_cos(x - HALF_PI, n);
144   }
145
146   void mpfr_geometric_sin(mpfr_t R, mpfr_t x, unsigned int n)
147   {
148     mpfr_t y;
149     mpfr_init(y);
150     mpfr_sub(y, x, MPFR_HALF_PI, MPFR_RNDN);
151     mpfr_geometric_cos(R, y, n);
152   }
153
154   //tan(x) = sin(x)/cos(x)
155   double geometric_tan(double x, unsigned int n)
156   {
157     return geometric_sin(x, n)/geometric_cos(x, n);
158   }
159
160   void mpfr_geometric_tan(mpfr_t R, mpfr_t x, unsigned int n)
161   {
162     mpfr_t S, C;
163
164     mpfr_init(S);
165     mpfr_init(C);
166     mpfr_geometric_sin(S, x, n);
167     mpfr_geometric_cos(C, x, n);
168
169     mpfr_div(R, S, C, MPFR_RNDN);
170     if(mpfr_cmp_ui(R, 1000000) > 0)
171       mpfr_set_inf(R, 1);
172     else if(mpfr_cmp_si(R, -1000000) < 0)
173       mpfr_set_inf(R, -1);
```

```
174 }
175
176 #ifdef COMPILE_MAIN
177 int main(int argc, char **argv)
178 {
179   double x, y;
180   unsigned int n, p, D;
181   mpfr_t R, X;
182   char sf[50];
183   FILE *in;
184
185   if(argc > 1)
186   {
187     switch(argv[1][0])
188     {
189       case 'a':
190         if(argc == 5 &&
191            sscanf(argv[2], "%lf", &x) == 1 &&
192            sscanf(argv[3], "%u" , &n) == 1 &&
193            sscanf(argv[4], "%u" , &D) == 1)
194           printf("Cos(%.*lf) = %.*lf\n",
195               d(D), x, D, geometric_cos(x, n));
196         else
197           printf("Usage: %s a <x=value for Cos(x)> <n> "
198                "<D=Number of digits to display>\n",
199
200                argv[0]);
201         break;
202
203       case 'b':
204         if(argc == 5 &&
205            sscanf(argv[2], "%lf", &x) == 1 &&
206            sscanf(argv[3], "%u" , &n) == 1 &&
207            sscanf(argv[4], "%u" , &D) == 1)
208           printf("Sin(%.*lf) = %.*lf\n",
209               d(D), x, D, geometric_sin(x, n));
210         else
211           printf("Usage: %s b <x=value for Sin(x)> <n> "
212                "<D=Number of digits to display>\n",
213
214                argv[0]);
215         break;
216
217       case 'c':
218         if(argc == 5 &&
219            sscanf(argv[2], "%lf", &x) == 1 &&
220            sscanf(argv[3], "%u" , &n) == 1 &&
221            sscanf(argv[4], "%u" , &D) == 1)
222           printf("Tan(%.*lf) = %.*lf\n",
223               d(D), x, D, geometric_tan(x, n));
224         else
225           printf("Usage: %s a <x=value for Tan(x)> <n> "
226                "<D=Number of digits to display>\n",
227
228                argv[0]);
229         break;
230
231       case 'd':
```

```
232           if(argc == 6 &&
233              sscanf(argv[3], "%u", &D) == 1 &&
234              sscanf(argv[4], "%u", &n) == 1 &&
235              sscanf(argv[5], "%u", &p) == 1)
236           {
237             mpfr_set_default_prec(p);
238             INIT_CONSTANTS
239             if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
240             {
241               mpfr_init(R);
242
243               sprintf(sf, "cos(%%.%uRNf) =~\n\t%%.%uRNf\n",
244                 d(D), D);
245
246               mpfr_geometric_cos(R, X, n);
247               mpfr_printf(sf, X, R);
248             }
249             else
250               printf("Usage: %s d <x=value for Cos(x)> "
251                    "<D=Number of digits to display> "
252                    "<n> <p=bits of precision to use>\n",
253                    argv[0]);
254           }
255           else
256             printf("Usage: %s d <x=value for Cos(x)> "
257                  "<D=Number of digits to display> "
258                  "<n> <p=bits of precision to use>\n",
259                  argv[0]);
260           break;
261
262         case 'e':
263           if(argc == 6 &&
264              sscanf(argv[3], "%u", &D) == 1 &&
265              sscanf(argv[4], "%u", &n) == 1 &&
266              sscanf(argv[5], "%u", &p) == 1)
267           {
268             mpfr_set_default_prec(p);
269             INIT_CONSTANTS
270             if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
271             {
272               mpfr_init(R);
273
274               sprintf(sf, "sin(%%.%uRNf) =~\n\t%%.%uRNf\n",
275                 d(D), D);
276
277               mpfr_geometric_sin(R, X, n);
278               mpfr_printf(sf, X, R);
279             }
280             else
281               printf("Usage: %s d <x=value for Sin(x)> "
282                    "<D=Number of digits to display> "
283                    "<n> <p=bits of precision to use>\n",
284                    argv[0]);
285           }
286           else
287             printf("Usage: %s d <x=value for Sin(x)> "
288                  "<D=Number of digits to display> "
289                  "<n> <p=bits of precision to use>\n",
```

```
                argv[0]);
        break;

      case 'f':
        if(argc == 6 &&
           sscanf(argv[3], "%u", &D) == 1 &&
           sscanf(argv[4], "%u", &n) == 1 &&
           sscanf(argv[5], "%u", &p) == 1)
        {
          mpfr_set_default_prec(p);
          INIT_CONSTANTS
          if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
          {
            mpfr_init(R);

            sprintf(sf, "tan(%%.%uRNf) =~\n\t%%.%uRNf\n",
              d(D), D);

            mpfr_geometric_tan(R, X, n);
            mpfr_printf(sf, X, R);
          }
          else
            printf("Usage: %s d <x=value for Tan(x)> "
                 "<D=Number of digits to display> "
                 "<n> <p=bits of precision to use>\n",
                 argv[0]);
        }
        else
          printf("Usage: %s d <x=value for Tan(x)> "
               "<D=Number of digits to display> "
               "<n> <p=bits of precision to use>\n",
               argv[0]);
        break;

      default:
        printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
    }
  }
  else
    printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
}
#endif
```

Code for Geometric Inverse Trigonometric Functions:

File : geometric_inv_trig.c

```
#include <assert.h>
#include <stdio.h>
#include <math.h>
#include <gmp.h>
#include <mpfr.h>

#include "geometric_inv_trig.h"
#include "trig_utilities.h"
#include "utilities.h"

#define INIT_CONSTANTS in = fopen(PI_INFILE, "r"); \
              mpfr_init(MPFR_PI); \
```

```
13              mpfr_inp_str(MPFR_PI, in, 10, MPFR_RNDN); \
14              fclose(in); \
15              mpfr_init(MPFR_HALF_PI); \
16              mpfr_div_ui(MPFR_HALF_PI, MPFR_PI, 2, MPFR_RNDN);
17
18  mpfr_t MPFR_PI, MPFR_HALF_PI;
19
20  double geometric_acos_bounded(double x, unsigned int n)
21  {
22    //Ensures the given value is a valid cosine value
23    assert(x >= 0 && x <= 1);
24
25    //Reversing the last line of geometric_cos_bounded
26    double h = 2-2*x;
27
28    //Reverses the iterative process oof geometric_cos_bounded
29    for(int i = 0; i < n; i++)
30      h = 2 - sqrt(4 - h);
31
32    //Reverses the initialisation proceduce in geometric_cos_bounded
33    h *= pow(4, n);
34    return sqrt(h);
35  }
36
37  void mpfr_geometric_acos_bounded(mpfr_t R, mpfr_t x, unsigned int n)
38  {
39    assert(mpfr_cmp_ui(x, 0) >= 0 && mpfr_cmp_ui(x, 1) <= 0);
40
41    mpfr_t h;
42
43    mpfr_init(h);
44    mpfr_ui_sub(h, 1, x, MPFR_RNDN);
45    mpfr_mul_ui(h, h, 2, MPFR_RNDN);
46
47    for(int i = 0; i < n; i++)
48    {
49      mpfr_ui_sub(h, 4, h, MPFR_RNDN);
50      mpfr_sqrt(h, h, MPFR_RNDN);
51      mpfr_ui_sub(h, 2, h, MPFR_RNDN);
52    }
53
54    mpfr_ui_pow_ui(R, 4, n, MPFR_RNDN);
55    mpfr_mul(R, h, R, MPFR_RNDN);
56    mpfr_sqrt(R, R, MPFR_RNDN);
57  }
58
59  double geometric_acos(double x, unsigned int n)
60  {
61    assert(x >= -1 && x <= 1);
62    return x >= 0 ? geometric_acos_bounded(x,n)
63            : PI - geometric_acos_bounded(-x,n);
64  }
65
66  void mpfr_geometric_acos(mpfr_t R, mpfr_t x, unsigned int n)
67  {
68    assert(mpfr_cmp_si(x, -1) >= 0 && mpfr_cmp_si(x, 1) <= 0);
69    mpfr_t y;
70
```

```
71    if(mpfr_cmp_ui(x, 0) < 0)
72    {
73      mpfr_init_set(y, x, MPFR_RNDN);
74      mpfr_neg(y, x, MPFR_RNDN);
75      mpfr_geometric_acos_bounded(R,y,n);
76      mpfr_sub(R, MPFR_PI, R, MPFR_RNDN);
77    }
78    else
79      mpfr_geometric_acos_bounded(R,x,n);
80  }
81
82  double geometric_asin(double x, unsigned int n)
83  {
84    assert(x >= -1 && x <= 1);
85    return HALF_PI - geometric_acos(x, n);
86  }
87
88  void mpfr_geometric_asin(mpfr_t R, mpfr_t x, unsigned int n)
89  {
90    assert(mpfr_cmp_si(x, -1) >= 0 && mpfr_cmp_si(x, 1) <= 0);
91
92    mpfr_geometric_acos(R, x, n);
93    mpfr_sub(R, MPFR_HALF_PI, R, MPFR_RNDN);
94  }
95
96  double geometric_atan(double x, unsigned int n)
97  {
98    return geometric_asin(x/sqrt(x*x + 1), n);
99  }
100
101 void mpfr_geometric_atan(mpfr_t R, mpfr_t x, unsigned int n)
102 {
103   mpfr_t y;
104
105   mpfr_init(y);
106   mpfr_mul(y, x, x, MPFR_RNDN);
107   mpfr_add_ui(y, y, 1, MPFR_RNDN);
108   mpfr_sqrt(y, y, MPFR_RNDN);
109   mpfr_div(y, x, y, MPFR_RNDN);
110
111   mpfr_geometric_asin(R, y, n);
112 }
113
114 #ifdef COMPILE_MAIN
115 int main(int argc, char **argv)
116 {
117   double x, y;
118   unsigned int n, p, D;
119   mpfr_t R, X;
120   char sf[50];
121   FILE *in;
122
123   if(argc > 1)
124   {
125     switch(argv[1][0])
126     {
127       case 'a':
128         if(argc == 5 &&
```

```
129             sscanf(argv[2], "%lf", &x) == 1 &&
130             sscanf(argv[3], "%u" , &n) == 1 &&
131             sscanf(argv[4], "%u" , &D) == 1)
132           printf("arcCos(%.*lf) = %.*lf\n",
133               d(D), x, D, geometric_acos(x, n));
134         else
135           printf("Usage: %s a <x=value for arcCos(x)> <n> "
136                "<D=Number of digits to display>\n",
137               argv[0]);
138         break;
139
140       case 'b':
141         if(argc == 5 &&
142            sscanf(argv[2], "%lf", &x) == 1 &&
143            sscanf(argv[3], "%u" , &n) == 1 &&
144            sscanf(argv[4], "%u" , &D) == 1)
145           printf("arcSin(%.*lf) = %.*lf\n",
146               d(D), x, D, geometric_asin(x, n));
147         else
148           printf("Usage: %s b <x=value for arcSin(x)> <n> "
149                "<D=Number of digits to display>\n",
150               argv[0]);
151         break;
152
153       case 'c':
154         if(argc == 5 &&
155            sscanf(argv[2], "%lf", &x) == 1 &&
156            sscanf(argv[3], "%u" , &n) == 1 &&
157            sscanf(argv[4], "%u" , &D) == 1)
158           printf("Tan(%.*lf) = %.*lf\n",
159               d(D), x, D, geometric_atan(x, n));
160         else
161           printf("Usage: %s a <x=value for arcTan(x)> <n> "
162                "<D=Number of digits to display>\n",
163               argv[0]);
164         break;
165
166       case 'd':
167         if(argc == 6 &&
168            sscanf(argv[3], "%u", &D) == 1 &&
169            sscanf(argv[4], "%u", &n) == 1 &&
170            sscanf(argv[5], "%u", &p) == 1)
171         {
172           mpfr_set_default_prec(p);
173           INIT_CONSTANTS
174           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
175           {
176             mpfr_init(R);
177
178             sprintf(sf, "arcCos(%%.%uRNf) =~\n\t%%.%uRNf\n",
179               d(D), D);
180
181             mpfr_geometric_acos(R, X, n);
182             mpfr_printf(sf, X, R);
183           }
184           else
185             printf("Usage: %s d <x=value for arcCos(x)> "
186                  "<D=Number of digits to display> "
```

```
187                   "<n> <p=bits of precision to use>\n",
188                   argv[0]);
189         }
190         else
191           printf("Usage: %s d <x=value for arcCos(x)> "
192                "<D=Number of digits to display> "
193                "<n> <p=bits of precision to use>\n",
194                argv[0]);
195         break;
196
197       case 'e':
198         if(argc == 6 &&
199            sscanf(argv[3], "%u", &D) == 1 &&
200            sscanf(argv[4], "%u", &n) == 1 &&
201            sscanf(argv[5], "%u", &p) == 1)
202         {
203           mpfr_set_default_prec(p);
204           INIT_CONSTANTS
205           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
206           {
207             mpfr_init(R);
208
209             sprintf(sf, "arcSin(%%.%uRNf) =~\n\t%%.%uRNf\n",
210               d(D), D);
211
212             mpfr_geometric_asin(R, X, n);
213             mpfr_printf(sf, X, R);
214           }
215           else
216             printf("Usage: %s d <x=value for arcSin(x)> "
217                  "<D=Number of digits to display> "
218                  "<n> <p=bits of precision to use>\n",
219                  argv[0]);
220         }
221         else
222           printf("Usage: %s d <x=value for arcSin(x)> "
223                "<D=Number of digits to display> "
224                "<n> <p=bits of precision to use>\n",
225                argv[0]);
226         break;
227
228       case 'f':
229         if(argc == 6 &&
230            sscanf(argv[3], "%u", &D) == 1 &&
231            sscanf(argv[4], "%u", &n) == 1 &&
232            sscanf(argv[5], "%u", &p) == 1)
233         {
234           mpfr_set_default_prec(p);
235           INIT_CONSTANTS
236           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
237           {
238             mpfr_init(R);
239
240             sprintf(sf, "arcTan(%%.%uRNf) =~\n\t%%.%uRNf\n",
241               d(D), D);
242
243             mpfr_geometric_atan(R, X, n);
244             mpfr_printf(sf, X, R);
```

```
245             }
246             else
247               printf("Usage: %s d <x=value for arcTan(x)> "
248                    "<D=Number of digits to display> "
249                    "<n> <p=bits of precision to use>\n",
250                    argv[0]);
251           }
252           else
253             printf("Usage: %s d <x=value for arcTan(x)> "
254                  "<D=Number of digits to display> "
255                  "<n> <p=bits of precision to use>\n",
256                  argv[0]);
257           break;
258
259         default:
260           printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
261       }
262     }
263     else
264       printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
265 }
266 #endif
```

Code for Taylor Trigonometric Functions:

File : taylor_trig.c

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <gmp.h>
4  #include <mpfr.h>
5
6  #include "taylor_trig.h"
7  #include "trig_utilities.h"
8  #include "utilities.h"
9
10 #define INIT_CONSTANTS in = fopen(PI_INFILE, "r"); \
11               mpfr_init(MPFR_PI); \
12               mpfr_inp_str(MPFR_PI, in, 10, MPFR_RNDN); \
13               fclose(in); \
14               mpfr_init(MPFR_TWO_PI); \
15               mpfr_init(MPFR_HALF_PI); \
16               mpfr_div_ui(MPFR_HALF_PI, MPFR_PI, 2, MPFR_RNDN); \
17               mpfr_mul_ui(MPFR_TWO_PI, MPFR_PI, 2, MPFR_RNDN);
18
19 mpfr_t MPFR_PI, MPFR_HALF_PI, MPFR_TWO_PI;
20
21 double taylor_cos_bounded(double x, unsigned int N)
22 {
23   assert(x >= 0 && x <= HALF_PI);
24   //Sets the initial values
25   double c = 1, x_2 = x*x, a = 1, b = 1;
26   for(int n = 1; n < N; n++)
27   {
28     //Performs first taylor series update
29     a /= (2*n - 1)*(2*(n++));
30     b *= x_2;
31     c -= a*b;
32     //Performs second taylor series update
```

```
33      a /= (2*n - 1)*(2*(n));
34      b *= x_2;
35      c += a*b;
36    }
37    return c;
38  }
39
40  void mpfr_taylor_cos_bounded(mpfr_t R, mpfr_t x, unsigned int N)
41  {
42    assert(mpfr_cmp_ui(x, 0) >= 0 && mpfr_cmp(x, MPFR_HALF_PI) <= 0);
43    mpfr_t t, x_2;
44
45    mpfr_init_set_ui(R, 1, MPFR_RNDN);
46    mpfr_init_set_ui(t, 1, MPFR_RNDN);
47    mpfr_init(x_2);
48    mpfr_mul(x_2, x, x, MPFR_RNDN);
49
50    for(int n = 1; n < N; n++)
51    {
52      mpfr_div_ui(t, t, (2*n-1)*(2*(n++)), MPFR_RNDN);
53      mpfr_mul(t, t, x_2, MPFR_RNDN);
54      mpfr_sub(R, R, t, MPFR_RNDN);
55      mpfr_div_ui(t, t, (2*n-1)*(2*n), MPFR_RNDN);
56      mpfr_mul(t, t, x_2, MPFR_RNDN);
57      mpfr_add(R, R, t, MPFR_RNDN);
58    }
59  }
60
61  double taylor_sin_bounded(double x, unsigned int N)
62  {
63    assert(x >= 0 && x <= HALF_PI);
64    double s = x, x_2 = x*x, a = 1, b = x;
65    for(int n = 1; n < N; n++)
66    {
67      a /= (2*n + 1)*(2*(n++));
68      b *= x_2;
69      s -= a*b;
70      a /= (2*n + 1)*(2*n);
71      b *= x_2;
72      s += a*b;
73    }
74    return s;
75  }
76
77  void mpfr_taylor_sin_bounded(mpfr_t R, mpfr_t x, unsigned int N)
78  {
79    mpfr_printf("%.20RNF\n", x);
80    assert(mpfr_cmp_ui(x, 0) >= 0 && mpfr_cmp(x, MPFR_HALF_PI) <= 0);
81    mpfr_t t, x_2;
82
83    mpfr_init_set(R, x, MPFR_RNDN);
84    mpfr_init_set(t, x, MPFR_RNDN);
85    mpfr_init(x_2);
86    mpfr_mul(x_2, x, x, MPFR_RNDN);
87
88    for(int n = 1; n < N; n++)
89    {
90      mpfr_div_ui(t, t, (2*n+1)*(2*(n++)), MPFR_RNDN);
```

```
91      mpfr_mul(t, t, x_2, MPFR_RNDN);
92      mpfr_sub(R, R, t, MPFR_RNDN);
93      mpfr_div_ui(t, t, (2*n+1)*(2*n), MPFR_RNDN);
94      mpfr_mul(t, t, x_2, MPFR_RNDN);
95      mpfr_add(R, R, t, MPFR_RNDN);
96    }
97  }
98
99  double taylor_cos(double x, unsigned int N)
100 {
101   if(x >= 0)
102   {
103     while(x >= TWO_PI)
104       x -= TWO_PI;
105
106     if(x >= PI)
107       if(x - PI >= HALF_PI)
108         return taylor_cos_bounded(TWO_PI - x, N);
109       else
110         return -1  * taylor_cos_bounded(x - PI, N);
111     else
112       if(x >= HALF_PI)
113         return -1 * taylor_cos_bounded(PI - x, N);
114       else
115         return taylor_cos_bounded(x, N);
116   }
117
118   return taylor_cos(-x, N);
119 }
120
121 void mpfr_taylor_cos(mpfr_t R, mpfr_t x, unsigned int N)
122 {
123   mpfr_t y, t;
124   mpfr_init_set(y, x, MPFR_RNDN);
125   mpfr_init(t);
126   if(mpfr_cmp_ui(y, 0) >= 0)
127   {
128     while(mpfr_cmp(y, MPFR_TWO_PI) >= 0)
129       mpfr_sub(y, y, MPFR_TWO_PI, MPFR_RNDN);
130
131     if(mpfr_cmp(y, MPFR_PI) >= 0)
132     {
133       mpfr_sub(t, y, MPFR_PI, MPFR_RNDN);
134       if(mpfr_cmp(t, MPFR_HALF_PI) >= 0)
135       {
136         mpfr_sub(y, MPFR_TWO_PI, y, MPFR_RNDN);
137         mpfr_taylor_cos_bounded(R, y, N);
138       }
139       else
140       {
141         mpfr_sub(y, y, MPFR_PI, MPFR_RNDN);
142         mpfr_taylor_cos_bounded(R, y, N);
143         mpfr_neg(R, R, MPFR_RNDN);
144       }
145     }
146     else
147     {
148       if(mpfr_cmp(y, MPFR_HALF_PI) >= 0)
```

```
149         {
150           mpfr_sub(y, MPFR_PI, y, MPFR_RNDN);
151           mpfr_taylor_cos_bounded(R, y, N);
152           mpfr_neg(R, R, MPFR_RNDN);
153         }
154         else
155         {
156           mpfr_taylor_cos_bounded(R, y, N);
157         }
158       }
159     }
160     else
161     {
162       mpfr_neg(y, y, MPFR_RNDN);
163       mpfr_taylor_cos_bounded(R, y, N);
164     }
165 }
166
167 double taylor_sin(double x, unsigned int N)
168 {
169     if(x >= 0)
170     {
171       while(x >= TWO_PI)
172         x -= TWO_PI;
173
174       if(x >= PI)
175         if(x - PI >= HALF_PI)
176           return -1 * taylor_sin_bounded(TWO_PI - x, N);
177         else
178           return -1 * taylor_sin_bounded(x - PI, N);
179       else
180         if(x >= HALF_PI)
181           return taylor_sin_bounded(PI - x, N);
182         else
183           return taylor_sin_bounded(x, N);
184     }
185
186     return -1 * taylor_sin(-x, N);
187 }
188
189 void mpfr_taylor_sin(mpfr_t R, mpfr_t x, unsigned int N)
190 {
191     mpfr_t y, t;
192     mpfr_init_set(y, x, MPFR_RNDN);
193     mpfr_init(t);
194
195     if(mpfr_cmp_ui(y, 0) >= 0)
196     {
197       while(mpfr_cmp(y, MPFR_TWO_PI) >= 0)
198         mpfr_sub(y, y, MPFR_TWO_PI, MPFR_RNDN);
199
200       if(mpfr_cmp(y, MPFR_PI) >= 0)
201       {
202         mpfr_sub(t, y, MPFR_PI, MPFR_RNDN);
203         if(mpfr_cmp(t, MPFR_PI) >= 0)
204         {
205           mpfr_sub(y, MPFR_TWO_PI, y, MPFR_RNDN);
206           mpfr_taylor_sin_bounded(R, y, N);
```

```
207           mpfr_neg(R, R, MPFR_RNDN);
208         }
209         else
210         {
211           mpfr_sub(y, y, MPFR_PI, MPFR_RNDN);
212           mpfr_taylor_sin_bounded(R, y, N);
213           mpfr_neg(R, R, MPFR_RNDN);
214         }
215       }
216       else
217       {
218         if(mpfr_cmp(y, MPFR_HALF_PI) >= 0)
219         {
220           mpfr_sub(y, MPFR_PI, y, MPFR_RNDN);
221           mpfr_taylor_sin_bounded(R, y, N);
222         }
223         else
224         {
225           mpfr_taylor_sin_bounded(R, y, N);
226         }
227       }
228     }
229     else
230     {
231       mpfr_neg(y, y, MPFR_RNDN);
232       mpfr_taylor_sin(R, y, N);
233       mpfr_neg(R, R, MPFR_RNDN);
234     }
235 }
236
237
238 double taylor_tan(double x, unsigned int N)
239 {
240   return taylor_sin(x,N)/taylor_cos(x,N);
241 }
242
243 void mpfr_taylor_tan(mpfr_t R, mpfr_t x, unsigned int N)
244 {
245   mpfr_t S, C;
246   mpfr_init(S);
247   mpfr_init(C);
248   mpfr_taylor_sin(S, x, N);
249   mpfr_taylor_cos(C, x, N);
250   mpfr_div(R, S, C, N);
251 }
252
253 #ifdef COMPILE_MAIN
254 int main(int argc, char **argv)
255 {
256   double x, y;
257   unsigned int n, p, D;
258   mpfr_t R, X;
259   char sf[50];
260   FILE *in;
261
262   if(argc > 1)
263   {
264     switch(argv[1][0])
```

```
265     {
266       case 'a':
267         if(argc == 5 &&
268            sscanf(argv[2], "%lf", &x) == 1 &&
269            sscanf(argv[3], "%u" , &n) == 1 &&
270            sscanf(argv[4], "%u" , &D) == 1)
271           printf("Cos(%.*lf) = %.*lf\n",
272               d(D), x, D, taylor_cos(x, n));
273         else
274           printf("Usage: %s a <x=value for Cos(x)> <n> "
275                "<D=Number of digits to display>\n",
276
277                argv[0]);
278         break;
279
280       case 'b':
281         if(argc == 5 &&
282            sscanf(argv[2], "%lf", &x) == 1 &&
283            sscanf(argv[3], "%u" , &n) == 1 &&
284            sscanf(argv[4], "%u" , &D) == 1)
285           printf("Sin(%.*lf) = %.*lf\n",
286               d(D), x, D, taylor_sin(x, n));
287         else
288           printf("Usage: %s b <x=value for Sin(x)> <n> "
289                "<D=Number of digits to display>\n",
290
291                argv[0]);
292         break;
293
294       case 'c':
295         if(argc == 5 &&
296            sscanf(argv[2], "%lf", &x) == 1 &&
297            sscanf(argv[3], "%u" , &n) == 1 &&
298            sscanf(argv[4], "%u" , &D) == 1)
299           printf("Tan(%.*lf) = %.*lf\n",
300               d(D), x, D, taylor_tan(x, n));
301         else
302           printf("Usage: %s a <x=value for Tan(x)> <n> "
303                "<D=Number of digits to display>\n",
304
305                argv[0]);
306         break;
307
308       case 'd':
309         if(argc == 6 &&
310            sscanf(argv[3], "%u", &D) == 1 &&
311            sscanf(argv[4], "%u", &n) == 1 &&
312            sscanf(argv[5], "%u", &p) == 1)
313         {
314           mpfr_set_default_prec(p);
315           INIT_CONSTANTS
316           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
317           {
318             mpfr_init(R);
319
320             sprintf(sf, "cos(%%.%uRNf) =~\n\t%%.%uRNf\n",
321               d(D), D);
322
```

```
323             mpfr_taylor_cos(R, X, n);
324             mpfr_printf(sf, X, R);
325           }
326           else
327             printf("Usage: %s d <x=value for Cos(x)> "
328                  "<D=Number of digits to display> "
329                  "<n> <p=bits of precision to use>\n",
330                  argv[0]);
331         }
332         else
333           printf("Usage: %s d <x=value for Cos(x)> "
334                "<D=Number of digits to display> "
335                "<n> <p=bits of precision to use>\n",
336                argv[0]);
337         break;
338
339       case 'e':
340         if(argc == 6 &&
341            sscanf(argv[3], "%u", &D) == 1 &&
342            sscanf(argv[4], "%u", &n) == 1 &&
343            sscanf(argv[5], "%u", &p) == 1)
344         {
345           mpfr_set_default_prec(p);
346           INIT_CONSTANTS
347           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
348           {
349             mpfr_init(R);
350
351             sprintf(sf, "sin(%%.%uRNf) =~\n\t%%.%uRNf\n",
352               d(D), D);
353
354             mpfr_taylor_sin(R, X, n);
355             mpfr_printf(sf, X, R);
356           }
357           else
358             printf("Usage: %s d <x=value for Sin(x)> "
359                  "<D=Number of digits to display> "
360                  "<n> <p=bits of precision to use>\n",
361                  argv[0]);
362         }
363         else
364           printf("Usage: %s d <x=value for Sin(x)> "
365                "<D=Number of digits to display> "
366                "<n> <p=bits of precision to use>\n",
367                argv[0]);
368         break;
369
370       case 'f':
371         if(argc == 6 &&
372            sscanf(argv[3], "%u", &D) == 1 &&
373            sscanf(argv[4], "%u", &n) == 1 &&
374            sscanf(argv[5], "%u", &p) == 1)
375         {
376           mpfr_set_default_prec(p);
377           INIT_CONSTANTS
378           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
379           {
380             mpfr_init(R);
```

```
381
382             sprintf(sf, "tan(%%.%uRNf) =~\n\t%%.%uRNf\n",
383               d(D), D);
384
385             mpfr_taylor_tan(R, X, n);
386             mpfr_printf(sf, X, R);
387           }
388           else
389             printf("Usage: %s d <x=value for Tan(x)> "
390                  "<D=Number of digits to display> "
391                  "<n> <p=bits of precision to use>\n",
392                  argv[0]);
393         }
394         else
395           printf("Usage: %s d <x=value for Tan(x)> "
396                "<D=Number of digits to display> "
397                "<n> <p=bits of precision to use>\n",
398                argv[0]);
399         break;
400
401       default:
402         printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
403     }
404   }
405   else
406     printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
407 }
408 #endif
```

Code for Taylor Inverse Trigonometric Functions:

File : taylor_inv_trig.c

```
1  #include <assert.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <gmp.h>
5  #include <mpfr.h>
6
7  #include "taylor_inv_trig.h"
8  #include "trig_utilities.h"
9  #include "utilities.h"
10
11 #define INIT_CONSTANTS in = fopen(PI_INFILE, "r"); \
12              mpfr_init(MPFR_PI); \
13              mpfr_inp_str(MPFR_PI, in, 10, MPFR_RNDN); \
14              fclose(in); \
15              mpfr_init(MPFR_HALF_PI); \
16              mpfr_div_ui(MPFR_HALF_PI, MPFR_PI, 2, MPFR_RNDN);
17
18 mpfr_t MPFR_PI, MPFR_HALF_PI;
19
20 double taylor_asin(double x, unsigned int N)
21 {
22   assert(x >= -1 && x <= 1);
23
24   //Checks for boundry cases
25   if(x == 1)
26     return HALF_PI;
```

```
27    else if(x == -1)
28      return -1 * HALF_PI;
29
30    //Sets the initial variables, t takes the place of a, b, c and y from
31    //  the report text
32    double s = x, x_2 = x*x, t = x;
33
34    for(int n = 1; n < N; n++)
35    {
36      //Performs a single taylor series update step
37      t *= 2*n*(2*n - 1)*x_2;
38      t /= 4*n*n;
39      s += t/(2*n+1);
40    }
41    return s;
42  }
43
44  void mpfr_taylor_asin(mpfr_t R, mpfr_t x, unsigned int N)
45  {
46    assert(mpfr_cmp_si(x, -1) >= 0 && mpfr_cmp_si(x, 1) <= 0);
47
48    if(mpfr_cmp_si(x, 1) == 0)
49      mpfr_set(R, MPFR_HALF_PI, MPFR_RNDN);
50    else if(mpfr_cmp_si(x, -1) == 0)
51    {
52      mpfr_set(R, MPFR_HALF_PI, MPFR_RNDN);
53      mpfr_neg(R, R, MPFR_RNDN);
54    }
55    else
56    {
57      mpfr_t x_2, t, T;
58      mpfr_init(x_2);
59      mpfr_mul(x_2, x, x, MPFR_RNDN);
60      mpfr_init_set(t, x, MPFR_RNDN);
61      mpfr_init(T);
62      mpfr_set(R, x, MPFR_RNDN);
63      for(int n = 1; n < N; n++)
64      {
65        mpfr_mul_ui(t, t, 2*n*(2*n - 1), MPFR_RNDN);
66        mpfr_mul(t, t, x_2, MPFR_RNDN);
67        mpfr_div_ui(t, t, 4*n*n, MPFR_RNDN);
68        mpfr_div_ui(T, t, 2*n + 1, MPFR_RNDN);
69        mpfr_add(R, R, T, MPFR_RNDN);
70      }
71    }
72  }
73
74  double taylor_acos(double x, unsigned int N)
75  {
76    return HALF_PI - taylor_asin(x, N);
77  }
78
79  void mpfr_taylor_acos(mpfr_t R, mpfr_t x, unsigned int N)
80  {
81    mpfr_taylor_asin(R, x, N);
82    mpfr_sub(R, MPFR_HALF_PI, R, MPFR_RNDN);
83  }
84
```

```
85  double taylor_atan_bounded(double x, unsigned int N)
86  {
87    assert(x >= 0 && x <= 1);
88    //Sets initial values
89    double t = 0, x_2 = x*x, y = x;
90    for(int n = 0; n < N; n++)
91    {
92      //Performs 2 taylor series updates of the value
93      t += y/(2*(n++) + 1);
94      y *= x_2;
95      t -= y/(2*n + 1);
96      y *= x_2;
97    }
98    return t;
99  }
100
101 void mpfr_taylor_atan_bounded(mpfr_t R, mpfr_t x, unsigned int N)
102 {
103   assert(mpfr_cmp_ui(x, 0) >= 0 && mpfr_cmp_ui(x, 1) <= 1);
104   mpfr_t x_2, y, a;
105   mpfr_init_set(y, x, MPFR_RNDN);
106   mpfr_init_set_ui(R, 0, MPFR_RNDN);
107   mpfr_init(x_2);
108   mpfr_mul(x_2, x, x, MPFR_RNDN);
109   mpfr_init(a);
110   for(int n = 0; n < N; n++)
111   {
112     mpfr_div_ui(a, y, 2*(n++) + 1, MPFR_RNDN);
113     mpfr_add(R, R, a, MPFR_RNDN);
114     mpfr_mul(y, y, x_2, MPFR_RNDN);
115     mpfr_div_ui(a, y, 2*n + 1, MPFR_RNDN);
116     mpfr_sub(R, R, a, MPFR_RNDN);
117     mpfr_mul(y, y, x_2, MPFR_RNDN);
118   }
119 }
120
121 double taylor_atan(double x, unsigned int N)
122 {
123   if(x < 0)
124     return -taylor_atan(-x, N);
125
126   if(x >= 1)
127     return HALF_PI/2 + taylor_atan_bounded((x - 1)/(x + 1), N);
128
129   return taylor_atan_bounded(x, N);
130 }
131
132 void mpfr_taylor_atan(mpfr_t R, mpfr_t x, unsigned int N)
133 {
134   mpfr_t y, pi_4, z;
135   mpfr_init(y);
136   mpfr_init(pi_4);
137   mpfr_init(z);
138
139   if(mpfr_cmp_ui(x, 0) < 0)
140   {
141     mpfr_neg(y, x, MPFR_RNDN);
142     mpfr_taylor_atan(R, y, N);
```

```
143       mpfr_neg(R, R, MPFR_RNDN);
144     }
145     else if(mpfr_cmp_ui(x, 1) >= 0)
146     {
147       mpfr_div_ui(pi_4, MPFR_HALF_PI, 2, MPFR_RNDN);
148       mpfr_add_ui(z, x, 1, MPFR_RNDN);
149       mpfr_sub_ui(y, x, 1, MPFR_RNDN);
150       mpfr_div(y, y, z, MPFR_RNDN);
151       mpfr_taylor_atan_bounded(R, y, N);
152       mpfr_add(R, R, pi_4, MPFR_RNDN);
153     }
154     else
155     {
156       mpfr_taylor_atan_bounded(R, x, N);
157     }
158   }
159
160   #ifdef COMPILE_MAIN
161   int main(int argc, char **argv)
162   {
163     double x, y;
164     unsigned int n, p, D;
165     mpfr_t R, X;
166     char sf[50];
167     FILE *in;
168
169     if(argc > 1)
170     {
171       switch(argv[1][0])
172       {
173         case 'a':
174           if(argc == 5 &&
175              sscanf(argv[2], "%lf", &x) == 1 &&
176              sscanf(argv[3], "%u" , &n) == 1 &&
177              sscanf(argv[4], "%u" , &D) == 1)
178             printf("arcCos(%.*lf) = %.*lf\n",
179                 d(D), x, D, taylor_acos(x, n));
180           else
181             printf("Usage: %s a <x=value for arcCos(x)> <n> "
182                  "<D=Number of digits to display>\n",
183                 argv[0]);
184           break;
185
186         case 'b':
187           if(argc == 5 &&
188              sscanf(argv[2], "%lf", &x) == 1 &&
189              sscanf(argv[3], "%u" , &n) == 1 &&
190              sscanf(argv[4], "%u" , &D) == 1)
191             printf("arcSin(%.*lf) = %.*lf\n",
192                 d(D), x, D, taylor_asin(x, n));
193           else
194             printf("Usage: %s b <x=value for arcSin(x)> <n> "
195                  "<D=Number of digits to display>\n",
196                 argv[0]);
197           break;
198
199         case 'c':
200           if(argc == 5 &&
```

```
201             sscanf(argv[2], "%lf", &x) == 1 &&
202             sscanf(argv[3], "%u" , &n) == 1 &&
203             sscanf(argv[4], "%u" , &D) == 1)
204           printf("Tan(%.*lf) = %.*lf\n",
205               d(D), x, D, taylor_atan(x, n));
206         else
207           printf("Usage: %s a <x=value for arcTan(x)> <n> "
208                "<D=Number of digits to display>\n",
209               argv[0]);
210         break;
211
212       case 'd':
213         if(argc == 6 &&
214            sscanf(argv[3], "%u", &D) == 1 &&
215            sscanf(argv[4], "%u", &n) == 1 &&
216            sscanf(argv[5], "%u", &p) == 1)
217         {
218           mpfr_set_default_prec(p);
219           INIT_CONSTANTS
220           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
221           {
222             mpfr_init(R);
223
224             sprintf(sf, "arcCos(%%.%uRNf) =~\n\t%%.%uRNf\n",
225               d(D), D);
226
227             mpfr_taylor_acos(R, X, n);
228             mpfr_printf(sf, X, R);
229           }
230           else
231             printf("Usage: %s d <x=value for arcCos(x)> "
232                  "<D=Number of digits to display> "
233                  "<n> <p=bits of precision to use>\n",
234                  argv[0]);
235         }
236         else
237           printf("Usage: %s d <x=value for arcCos(x)> "
238                "<D=Number of digits to display> "
239                "<n> <p=bits of precision to use>\n",
240                argv[0]);
241         break;
242
243       case 'e':
244         if(argc == 6 &&
245            sscanf(argv[3], "%u", &D) == 1 &&
246            sscanf(argv[4], "%u", &n) == 1 &&
247            sscanf(argv[5], "%u", &p) == 1)
248         {
249           mpfr_set_default_prec(p);
250           INIT_CONSTANTS
251           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
252           {
253             mpfr_init(R);
254
255             sprintf(sf, "arcSin(%%.%uRNf) =~\n\t%%.%uRNf\n",
256               d(D), D);
257
258             mpfr_taylor_asin(R, X, n);
```

```
259             mpfr_printf(sf, X, R);
260           }
261           else
262             printf("Usage: %s d <x=value for arcSin(x)> "
263                  "<D=Number of digits to display> "
264                  "<n> <p=bits of precision to use>\n",
265                  argv[0]);
266         }
267         else
268           printf("Usage: %s d <x=value for arcSin(x)> "
269                "<D=Number of digits to display> "
270                "<n> <p=bits of precision to use>\n",
271                argv[0]);
272         break;
273
274       case 'f':
275         if(argc == 6 &&
276            sscanf(argv[3], "%u", &D) == 1 &&
277            sscanf(argv[4], "%u", &n) == 1 &&
278            sscanf(argv[5], "%u", &p) == 1)
279         {
280           mpfr_set_default_prec(p);
281           INIT_CONSTANTS
282           if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0)
283           {
284             mpfr_init(R);
285
286             sprintf(sf, "arcTan(%%.%uRNf) =~\n\t%%.%uRNf\n",
287               d(D), D);
288
289             mpfr_taylor_atan(R, X, n);
290             mpfr_printf(sf, X, R);
291           }
292           else
293             printf("Usage: %s d <x=value for arcTan(x)> "
294                  "<D=Number of digits to display> "
295                  "<n> <p=bits of precision to use>\n",
296                  argv[0]);
297         }
298         else
299           printf("Usage: %s d <x=value for arcTan(x)> "
300                "<D=Number of digits to display> "
301                "<n> <p=bits of precision to use>\n",
302                argv[0]);
303         break;
304
305       default:
306         printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
307     }
308   }
309   else
310     printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
311 }
312 #endif
```

Code for CORDIC Functions:

File : cordic_trig.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#include "cordic_trig.h"
#include "utilities.h"

//FIXED_ANGLES and FIXED_K_VALUES are defined in "cordic_trig.h",
//  they represent arrays of fixed point pre calculated constants
const cordic_fixed_t TRIG_ANGLES[]  = FIXED_ANGLES;
const cordic_fixed_t TRIG_K_VALUES[] = FIXED_K_VALUES;

double *cordic_trig(const double theta, const unsigned int iter)
{
  //Checks that -pi/2 <= theta <= pi/2 before continuing
  assert(-HALF_PI <= theta && theta <= HALF_PI);
  //If the given iter value is more thant the value of MAX_ITER, or
  //  iter is 0 then n is set to MAX_ITER, otherwise n is set to iter
  unsigned int n = (iter > MAX_ITER || iter == 0) ? MAX_ITER : iter;

  //Initialises the initial values for
  cordic_fixed_t x = TRIG_K_VALUES[n-1], y = 0, t;
  cordic_fixed_t beta = double_to_fixed(theta);
  //Assigns memory for the return array
  double *result = malloc(2*sizeof(*result));

  //Checks for the extreme values of -pi/2, 0 and pi/2
  if(beta == 0)
  {
    x = FIXED_ONE;
    beta = 0;
  }
  else if(beta == FIXED_HALF_PI)
  {
    x = 0;
    y = FIXED_ONE;
    beta = 0;
  }
  else if(beta == -FIXED_HALF_PI)
  {
    x = 0;
    y = -FIXED_ONE;
    beta = 0;
  }

  //Executes if beta != 0
  if(beta)
  {
    //Main loop
    for(int i = 0; i < n; i++)
    {
      //t is a temporary variable for x, to help in the update
      //  process
      t = x;
      if(beta >= 0)
      {
        x = x - (y >> i);
```

```
59          y = y + (t >> i);
60          beta -= TRIG_ANGLES[i];
61        }
62        else
63        {
64          x = x + (y >> i);
65          y = y - (t >> i);
66          beta += TRIG_ANGLES[i];
67        }
68      }
69    }
70
71    result[0] = fixed_to_double(x);
72    result[1] = fixed_to_double(y);
73    return result;
74  }
75
76  double cordic_atan_bounded(const double z, const unsigned int iter)
77  {
78    //Ensures that the given value is in the range of 0 <= z <= 1
79    assert(0 <= z && z <= 1);
80    //If the given iter value is more thant the value of MAX_ITER, or
81    //  iter is 0 then n is set to MAX_ITER, otherwise n is set to iter
82    unsigned int n = (iter > MAX_ITER || iter == 0) ? MAX_ITER : iter;
83
84    //Sets the initial values, note that y/x = z is the important factor
85    cordic_fixed_t x = FIXED_ONE >> 1, y = double_to_fixed(z) >> 1, t;
86    cordic_fixed_t beta = 0;
87
88    //Checks for the extreme case
89    if(y == 0)
90      return 0;
91
92    //Main loop
93    for(int i = 0; i < n; i++)
94    {
95      t = x;
96      if(y < 0)
97      {
98        x = x - (y >> i);
99        y = y + (t >> i);
100       beta -= TRIG_ANGLES[i];
101     }
102     else
103     {
104       x = x + (y >> i);
105       y = y - (t >> i);
106       beta += TRIG_ANGLES[i];
107     }
108   }
109
110   return fixed_to_double(beta);
111 }
112
113 double cordic_cos(double x, unsigned int n)
114 {
115   double *r;
116
```

```
117   if(x >= 0)
118   {
119     while(x >= TWO_PI)
120       x -= TWO_PI;
121
122     if(x >= PI)
123       if(x - PI >= HALF_PI)
124         r = cordic_trig(TWO_PI - x, n);
125       else
126       {
127         r = cordic_trig(x - PI, n);
128         r[0] = -r[0];
129       }
130     else
131       if(x >= HALF_PI)
132       {
133         r = cordic_trig(PI - x, n);
134         r[0] = -r[0];
135       }
136       else
137         r = cordic_trig(x, n);
138     return r[0];
139   }
140   return cordic_cos(-x, n);
141 }
142
143 double cordic_sin(double x, unsigned int n)
144 {
145   return cordic_cos(x - HALF_PI, n);
146 }
147
148 double cordic_tan(double x, unsigned int n)
149 {
150   double *r;
151   if(x >= 0)
152   {
153     while(x >= PI)
154       x -= PI;
155
156     if(x >= HALF_PI)
157     {
158       r = cordic_trig(PI - x, n);
159       return -1 * r[1]/r[0];
160     }
161
162     r = cordic_trig(x, n);
163     return r[1]/r[0];
164   }
165   return -1 * cordic_tan(-x, n);
166 }
167
168 double cordic_acos(double x, unsigned int n)
169 {
170   assert(-1 <= x && x <= 1);
171   return x == 0 ? HALF_PI
172           : x > 0 ? cordic_atan(sqrt(1 - x*x)/x, n)
173                   : HALF_PI + cordic_asin(-x, n);
174 }
```

```
175
176 double cordic_asin(double x, unsigned int n)
177 {
178   assert(-1 <= x && x <= 1);
179   return x == 1 ? HALF_PI
180          : x == -1 ? -HALF_PI
181                : cordic_atan(x/sqrt(1 - x*x), n);
182 }
183
184 double cordic_atan(double x, unsigned int n)
185 {
186   if(x < 0)
187     return -cordic_atan(-x, n);
188
189   if(x >= 1)
190     return HALF_PI/2 + cordic_atan_bounded((x-1)/(x+1), n);
191
192   return cordic_atan_bounded(x, n);
193 }
194
195 #ifdef COMPILE_MAIN
196 int main(int argc, char **argv)
197 {
198   double x;
199   unsigned int n, D;
200
201   if(argc > 1)
202   {
203     switch(argv[1][0])
204     {
205       case 'a':
206         if(argc == 5 &&
207            sscanf(argv[2], "%lf", &x) == 1 &&
208            sscanf(argv[3], "%u" , &n) == 1 &&
209            sscanf(argv[4], "%u" , &D) == 1)
210           printf("Cos(%.*lf) = %.*lf\n",
211               d(D), x, D, cordic_cos(x, n));
212         else
213           printf("Uasge: %s a <x=value for Cos(x)> <n> "
214                "<D=Number of digits to display>\n",
215                argv[0]);
216         break;
217
218       case 'b':
219         if(argc == 5 &&
220            sscanf(argv[2], "%lf", &x) == 1 &&
221            sscanf(argv[3], "%u" , &n) == 1 &&
222            sscanf(argv[4], "%u" , &D) == 1)
223           printf("Sin(%.*lf) = %.*lf\n",
224               d(D), x, D, cordic_sin(x, n));
225         else
226           printf("Uasge: %s a <x=value for Sin(x)> <n> "
227                "<D=Number of digits to display>\n",
228                argv[0]);
229         break;
230
231       case 'c':
232         if(argc == 5 &&
```

```
233             sscanf(argv[2], "%lf", &x) == 1 &&
234             sscanf(argv[3], "%u" , &n) == 1 &&
235             sscanf(argv[4], "%u" , &D) == 1)
236           printf("Tan(%.*lf) = %.*lf\n",
237               d(D), x, D, cordic_tan(x, n));
238         else
239           printf("Uasge: %s a <x=value for Tan(x)> <n> "
240                "<D=Number of digits to display>\n",
241                argv[0]);
242         break;
243
244       case 'd':
245         if(argc == 5 &&
246            sscanf(argv[2], "%lf", &x) == 1 &&
247            sscanf(argv[3], "%u" , &n) == 1 &&
248            sscanf(argv[4], "%u" , &D) == 1)
249           printf("aTan(%.*lf) = %.*lf\n",
250               d(D), x, D, cordic_atan(x, n));
251         else
252           printf("Uasge: %s d <x=value for aTan(x)> <n> "
253                "<D=Number of digits to display>\n",
254                argv[0]);
255         break;
256
257       case 'e':
258         if(argc == 5 &&
259            sscanf(argv[2], "%lf", &x) == 1 &&
260            sscanf(argv[3], "%u" , &n) == 1 &&
261            sscanf(argv[4], "%u" , &D) == 1)
262           printf("aCos(%.*lf) = %.*lf\n",
263               d(D), x, D, cordic_acos(x, n));
264         else
265           printf("Uasge: %s d <x=value for aCos(x)> <n> "
266                "<D=Number of digits to display>\n",
267                argv[0]);
268         break;
269
270       case 'f':
271         if(argc == 5 &&
272            sscanf(argv[2], "%lf", &x) == 1 &&
273            sscanf(argv[3], "%u" , &n) == 1 &&
274            sscanf(argv[4], "%u" , &D) == 1)
275           printf("aSin(%.*lf) = %.*lf\n",
276               d(D), x, D, cordic_asin(x, n));
277         else
278           printf("Uasge: %s d <x=value for aSin(x)> <n> "
279                "<D=Number of digits to display>\n",
280                argv[0]);
281         break;
282
283       default:
284         printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
285     }
286   }
287   else
288     printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
289 }
290 #endif
```

Header files for Trigonometric Functions:

File : geometric_trig.h

```
#ifndef GEOMETRIC_TRIG_HEADER
#define GEOMETRIC_TRIG_HEADER

  double geometric_cos_bounded(double, unsigned int);
  void mpfr_geometric_cos_bounded(mpfr_t, mpfr_t, unsigned int);

  double geometric_cos(double, unsigned int);
  double geometric_sin(double, unsigned int);
  double geometric_tan(double, unsigned int);

  void mpfr_geometric_cos(mpfr_t, mpfr_t, unsigned int);
  void mpfr_geometric_sin(mpfr_t, mpfr_t, unsigned int);
  void mpfr_geometric_tan(mpfr_t, mpfr_t, unsigned int);

#endif
```

File : geometric_inv_trig.h

```
#ifndef GEOMETRIC_INV_TRIG_HEADER
#define GEOMETRIC_INV_TRIG_HEADER

  double geometric_acos_bounded(double, unsigned int);
  void mpfr_geometric_acos_bounded(mpfr_t, mpfr_t, unsigned int);

  double geometric_acos(double, unsigned int);
  double geometric_asin(double, unsigned int);
  double geometric_atan(double, unsigned int);

  void mpfr_geometric_acos(mpfr_t, mpfr_t, unsigned int);
  void mpfr_geometric_asin(mpfr_t, mpfr_t, unsigned int);
  void mpfr_geometric_atan(mpfr_t, mpfr_t, unsigned int);

#endif
```

File : taylor_trig.h

```
#ifndef TAYLOR_TRIG_HEADER
#define TAYLOR_TRIG_HEADER

  double taylor_cos_bounded(double, unsigned int);
  double taylor_sin_bounded(double, unsigned int);
  double taylor_sin(double, unsigned int);
  double taylor_cos(double, unsigned int);
  double taulor_tan(double, unsigned int);

  void mpfr_taylor_cos_bounded(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_sin_bounded(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_sin(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_cos(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_cos(mpfr_t, mpfr_t, unsigned int);

#endif
```

File : taylor_inv_trig.h

```
#ifndef TAYLOR_INV_TRIG_HEADER
```

```
#define TAYLOR_INV_TRIG_HEADER

  double taylor_asin(double, unsigned int);
  double taylor_acos(double, unsigned int);
  double taylor_atan_bounded(double, unsigned int);
  double taylor_atan(double, unsigned int);

  void mpfr_taylor_asin(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_acos(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_atan_bounded(mpfr_t, mpfr_t, unsigned int);
  void mpfr_taylor_atan(mpfr_t, mpfr_t, unsigned int);
#endif
```

File : cordic_trig.h

```
#ifndef CORDIC_TRIG_HEADER
#define CORDIC_TRIG_HEADER

#include "trig_fixed.h"
#include "trig_utilities.h"

typedef TRIG_FIXED_TYPE cordic_fixed_t;

#if BITS == 64
  #define FIXED_ONE 0x4000000000000000
  #define NEG_CONSTANT 0x8000000000000000
  #define FIXED_HALF_PI 0x6487ed5110b4611a
  #define FIXED_ANGLES {0x3243f6a8885a308d, 0x1dac670561bb4f68, \
              0x0fadbafc96406eb1, 0x07f56ea6ab0bdb71, \
                0x03feab76e59fbd38, 0x01ffd55bba97624a, \
              0x00fffaaadddb94d5, 0x007fff5556eeea5c, \
              0x003fffeaaab7776e, 0x001ffffd5555bbbb, \
              0x000fffffaaaaaddd, 0x0007fffff555556e, \
              0x0003fffffeaaaaab, 0x0001ffffffd55555, \
              0x0000fffffffaaaaa, 0x00007fffffff5555, \
              0x00003fffffffeaaa, 0x00001ffffffffd55, \
              0x00000fffffffffaa, 0x000007fffffffff5, \
              0x000003fffffffffe, 0x000001fffffffff, \
              0x0000010000000000, 0x0000008000000000, \
              0x0000004000000000, 0x0000002000000000, \
              0x0000001000000000, 0x0000000800000000, \
              0x0000000400000000, 0x0000000200000000, \
              0x0000000100000000, 0x0000000080000000, \
              0x0000000040000000, 0x0000000020000000, \
              0x0000000010000000, 0x0000000008000000, \
              0x0000000004000000, 0x0000000002000000, \
              0x0000000001000000, 0x0000000000800000, \
              0x0000000000400000, 0x0000000000200000, \
              0x0000000000100000, 0x0000000000080000, \
              0x0000000000040000, 0x0000000000020000, \
              0x0000000000010000, 0x0000000000008000, \
              0x0000000000004000, 0x0000000000002000, \
              0x0000000000001000, 0x0000000000000800, \
              0x0000000000000400, 0x0000000000000200, \
              0x0000000000000100, 0x0000000000000080, \
              0x0000000000000040, 0x0000000000000020, \
              0x0000000000000010, 0x0000000000000008, \
              0x0000000000000004, 0x0000000000000002, \
              0x0000000000000001}
```

```c
  #define FIXED_K_VALUES {0x2d413cccfe779921, 0x287a26c490921db6, \
              0x2744c374daf46d2f, 0x26f72283bd67fbda, \
              0x26e3b58305ddeb19, 0x26ded9f57b2c3e7a, \
              0x26dda30d3e4fd185, 0x26dd5552e1641def, \
              0x26dd41e4454da117, 0x26dd3d089dfa47c8, \
              0x26dd3bd1b42095ce, 0x26dd3b83f9a9db95, \
              0x26dd3b708b0c282b, 0x26dd3b6baf64bb03, \
              0x26dd3b6a787adfb4, 0x26dd3b6a2ac068e0, \
              0x26dd3b6a1751cb2b, 0x26dd3b6a127623be, \
              0x26dd3b6a113f39e3, 0x26dd3b6a10f17f6c, \
              0x26dd3b6a10de10ce, 0x26dd3b6a10d93527, \
              0x26dd3b6a10d7fe3d, 0x26dd3b6a10d7b082, \
              0x26dd3b6a10d79d14, 0x26dd3b6a10d79838, \
              0x26dd3b6a10d79701, 0x26dd3b6a10d796b3, \
              0x26dd3b6a10d796a0, 0x26dd3b6a10d7969b, \
              0x26dd3b6a10d7969a, 0x26dd3b6a10d7969a, \
              0x26dd3b6a10d7969a, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699, 0x26dd3b6a10d79699, \
              0x26dd3b6a10d79699}
  #define MAX_ITER 63
#elif BITS == 32
  #define FIXED_ONE 0x40000000
  #define NEG_CONSTANT 0x80000000
  #define FIXED_HALF_PI 0x6487ed51
  #define FIXED_ANGLES {0x3243f6a8, 0x1dac6705, 0x0fadbafc, \
              0x07f56ea6, 0x03feab76, 0x01ffd55b, \
              0x00fffaaa, 0x007fff55, 0x003fffea, \
              0x001ffffd, 0x00100000, 0x00080000, \
              0x00040000, 0x00020000, 0x00010000, \
              0x00008000, 0x00004000, 0x00002000, \
              0x00001000, 0x00000800, 0x00000400, \
              0x00000200, 0x00000100, 0x00000080, \
              0x00000040, 0x00000020, 0x00000010, \
              0x00000008, 0x00000004, 0x00000002, \
              0x00000001}
  #define FIXED_K_VALUES {0x2d413ccc, 0x287a26c4, 0x2744c374, \
              0x26f72283, 0x26e3b583, 0x26ded9f5, \
              0x26dda30d, 0x26dd5552, 0x26dd41e4, \
              0x26dd3d08, 0x26dd3bd1, 0x26dd3b83, \
              0x26dd3b70, 0x26dd3b6b, 0x26dd3b6a, \
              0x26dd3b6a, 0x26dd3b6a, 0x26dd3b6a, \
              0x26dd3b6a, 0x26dd3b6a, 0x26dd3b6a, \
              0x26dd3b6a, 0x26dd3b6a, 0x26dd3b6a, \
              0x26dd3b6a, 0x26dd3b6a, 0x26dd3b6a, \
              0x26dd3b6a, 0x26dd3b6a, 0x26dd3b6a, \
```

```
                0x26dd3b6a}
  #define max_iter 30
#elif BITS == 16
  #define FIXED_ONE 0x4000
  #define NEG_CONSTANT 0x8000
  #define FIXED_HALF_PI 0x6487
  #define FIXED_ANGLES {0x3243, 0x1dac, 0x0fad, 0x07f5, 0x03fe, \
                0x01ff, 0x0100, 0x0080, 0x0040, 0x0020, \
                0x0010, 0x0008, 0x0004, 0x0002, 0x0001}
  #define FIXED_K_VALUES {0x2d41, 0x287a, 0x2744, 0x26f7, 0x26e3, \
                0x26de, 0x26dd, 0x26dd, 0x26dd, 0x26dd, \
                0x26dd, 0x26dd, 0x26dd, 0x26dd, 0x26dd}
  #define MAX_ITER 15
#elif BITS == 8
  #define FIXED_ONE 0x40
  #define NEG_CONSTANT 0x80
  #define FIXED_HALF_PI 0x64
  #define FIXED_ANGLES {0x32, 0x1d, 0x10, 0x08, 0x04, 0x02, 0x01}
  #define FIXED_K_VALUES {0x2d, 0x28, 0x27, 0x26, 0x26, 0x26, 0x26}
  #define MAX_ITER 7
#else
  #error "you shouldn't be able to get here; you done messed up"
#endif

  double *cordic_trig(const double, const unsigned int);

  double cordic_cos(double, unsigned int);
  double cordic_sin(double, unsigned int);
  double cordic_tan(double, unsigned int);

  double cordic_atan_bounde(const double, const unsigned int);
  double cordic_atan(double, unsigned int);
  double cordic_acos(double, unsigned int);
  double cordic_asin(double, unsigned int);

#endif
```

A.4 Exponential and Logarithm Code

Code for Integer Exponentiation:

File : int_exp.c

```
#include <stdio.h>
#include <gmp.h>
#include <mpfr.h>

#include "int_exp.h"
#include "utilities.h"

double naive_int_exp(const double x, const int a)
{
  //x^a == 1/x^(-a) for all a
  if(a < 0)
    return 1/naive_int_exp(x, -a);
  //Sets the initial values
  double z = 1;
  int n = a;
```

```
16    //Loops until n == 0
17    while(n--)
18      z *= x;
19    return z;
20  }
21
22  double squaring_int_exp(const double x, const int a)
23  {
24    //x^a == 1/x^(-a) for all a
25    if(a < 0)
26      return 1/squaring_int_exp(x, -a);
27    //Sets the initial values
28    double y = x, z = 1;
29    int n = a;
30    //Loops until n == 0
31    while(n)
32    {
33      //n%2 is true if n is odd
34      if(n%2)
35      {
36        z *= y;
37        --n;
38      }
39      y *= y;
40      n >>= 1;
41    }
42    return z;
43  }
44
45  void mpfr_naive_int_exp(mpfr_t R, mpfr_t x, mpz_t a)
46  {
47    if(mpz_cmp_ui(a, 0) < 0)
48    {
49      mpz_t b;
50      mpz_init_set(b, a);
51      mpz_neg(b, b);
52      mpfr_naive_int_exp(R, x, b);
53      mpfr_ui_div(R, 1, R, MPFR_RNDN);
54    }
55    else
56    {
57      mpz_t n;
58      mpz_init_set(n, a);
59      mpfr_set_ui(R, 1, MPFR_RNDN);
60      while(mpz_cmp_ui(n, 0) > 0)
61      {
62        mpfr_mul(R, R, x, MPFR_RNDN);
63        mpz_sub_ui(n, n, 1);
64      }
65    }
66  }
67
68  void mpfr_squaring_int_exp(mpfr_t R, mpfr_t x, mpz_t a)
69  {
70    if(mpz_cmp_ui(a, 0) < 0)
71    {
72      mpz_t b;
73      mpz_init_set(b, a);
```

```
74      mpz_neg(b, b);
75      mpfr_squaring_int_exp(R, x, b);
76      mpfr_ui_div(R, 1, R, MPFR_RNDN);
77    }
78    else
79    {
80      mpfr_t y;
81      mpz_t n;
82      mpfr_init_set(y, x, MPFR_RNDN);
83      mpfr_set_ui(R, 1, MPFR_RNDN);
84      mpz_init_set(n, a);
85      while(mpz_cmp_ui(n, 0) > 0)
86      {
87        if(mpz_odd_p(n))
88        {
89          mpfr_mul(R, R, y, MPFR_RNDN);
90          mpz_sub_ui(n, n, 1);
91        }
92        mpfr_mul(y, y, y, MPFR_RNDN);
93        mpz_div_ui(n, n, 2);
94      }
95    }
96  }
97
98  #ifdef COMPILE_MAIN
99  int main(int argc, char **argv)
100 {
101   double x;
102   unsigned int n, D, p;
103   mpfr_t X, R;
104   mpz_t N;
105   char sf[50];
106
107   if(argc > 1)
108   {
109     switch(argv[1][0])
110     {
111       case 'a':
112         if(argc == 5 &&
113            sscanf(argv[2], "%lf", &x) == 1 &&
114            sscanf(argv[3], "%u" , &n) == 1 &&
115            sscanf(argv[4], "%u" , &D) == 1)
116           printf("(%.*lf)^(%d) = %.*lf (Naive)\n",
117               d(D), x, n, D, naive_int_exp(x, n));
118         else
119           printf("Uasge: %s a <x=Base for exp> "
120                  "<n=Exponent for exp> "
121                "<D=Number of digits to display>\n",
122                argv[0]);
123         break;
124
125       case 'b':
126         if(argc == 5 &&
127            sscanf(argv[2], "%lf", &x) == 1 &&
128            sscanf(argv[3], "%u" , &n) == 1 &&
129            sscanf(argv[4], "%u" , &D) == 1)
130           printf("(%.*lf)^(%d) = %.*lf (Squaring)\n",
131               d(D), x, n, D, squaring_int_exp(x, n));
```

```
132           else
133             printf("Uasge: %s b <x=Base for exp> "
134                    "<n=Exponent for exp> "
135                  "<D=Number of digits to display>\n",
136                  argv[0]);
137           break;
138
139         case 'c':
140           if (argc == 6 &&
141               sscanf(argv[4], "%u", &D) == 1 &&
142               sscanf(argv[5], "%u" , &p) == 1)
143           {
144             mpfr_set_default_prec(p);
145
146             if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN)==0 &&
147               mpz_init_set_str(N, argv[3], 10) == 0)
148             {
149               mpfr_init(R);
150
151               sprintf(sf, "(%%.%uRNf)^%%Zd =~\t(Naive)"
152                     "\n\t%%.%uRNf\n", d(D), D);
153
154               mpfr_naive_int_exp(R, X, N);
155               mpfr_printf(sf, X, N, R);
156             }
157             else
158               printf("Usage: %s c <X=Base for exp> "
159                      "<N=Exponent for exp> "
160                    "<D=Number of digitsto calculate to> "
161                    "<p=bits of precision>\n", argv[0]);
162           }
163           else
164             printf("Usage: %s c <X=Base for exp> "
165                    "<N=Exponent for exp> "
166                  "<D=Number of digitsto calculate to> "
167                  "<p=bits of precision>\n", argv[0]);
168           break;
169
170         case 'd':
171           if (argc == 6 &&
172               sscanf(argv[4], "%u", &D) == 1 &&
173               sscanf(argv[5], "%u" , &p) == 1)
174           {
175             mpfr_set_default_prec(p);
176
177             if (mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN)==0 &&
178               mpz_init_set_str(N, argv[3], 10) == 0)
179             {
180               mpfr_init(R);
181
182               sprintf(sf, "(%%.%uRNf)^%%Zd =~\t(Squaring)"
183                     "\n\t%%.%uRNf\n", d(D), D);
184
185               mpfr_squaring_int_exp(R, X, N);
186               mpfr_printf(sf, X, N, R);
187             }
188             else
189               printf("Usage: %s d <X=Base for exp> "
```

```
                    "<N=Exponent for exp> "
                  "<D=Number of digitsto calculate to> "
                  "<p=bits of precision>\n", argv[0]);
        }
        else
          printf("Usage: %s d <X=Base for exp> "
                 "<N=Exponent for exp> "
               "<D=Number of digitsto calculate to> "
               "<p=bits of precision>\n", argv[0]);
        break;

      default:
        printf("Usage: %s <a/b/c/d> <arguments>\n", argv[0]);
    }
  }
  else
    printf("Usage: %s <a/b/c/d> <arguments>\n", argv[0]);
}
#endif
```

Code for Taylor Exponentials and Logarithms:

File : taylor_exp_log.c

```
#include <stdio.h>
#include <assert.h>
#include <gmp.h>
#include <mpfr.h>
#include <math.h>

#include "int_exp.h"
#include "log_exp_utilities.h"
#include "utilities.h"
#include "taylor_exp_log.h"

#define INIT_CONSTANTS in = fopen(NAT_LOG_2_INFILE, "r"); \
              mpfr_init(MPFR_NAT_LOG_2); \
              mpfr_inp_str(MPFR_NAT_LOG_2, in, 10, MPFR_RNDN); \
              fclose(in);

mpfr_t MPFR_NAT_LOG_2;

double naive_exp(double x, unsigned int n)
{
  double z = 1 + x/n;
  return z > 0 ? squaring_int_exp(z, n)
         : n%2 ? -squaring_int_exp(-z, n)
             : squaring_int_exp(-z, n);
}

void mpfr_naive_exp(mpfr_t R, mpfr_t x, mpz_t n)
{
  assert(mpz_cmp_ui(n, 0) >= 0);

  mpfr_t z;

  mpfr_init_set(z, x, MPFR_RNDN);
  mpfr_div_z(z, z, n, MPFR_RNDN);
  mpfr_add_ui(z, z, 1, MPFR_RNDN);
```

```
36
37    if(mpfr_cmp_ui(z, 0) > 0)
38      mpfr_squaring_int_exp(R, z, n);
39    else
40    {
41      mpfr_neg(z, z, MPFR_RNDN);
42      mpfr_squaring_int_exp(R, z, n);
43
44      if(mpz_odd_p(n))
45        mpfr_neg(R, R, MPFR_RNDN);
46    }
47  }
48
49  double taylor_exp(double x, unsigned int n)
50  {
51    //Sets initial values
52    double t, z;
53    t = 1;
54    z = 1;
55
56    //Main loop
57    for(int k = 1; k < n; ++k)
58    {
59      //Performs the taylor series update
60      t *= x;
61      t /= k;
62      z += t;
63    }
64
65    return z;
66  }
67
68  void mpfr_taylor_exp(mpfr_t R, mpfr_t x, mpz_t n)
69  {
70    assert(mpz_cmp_ui(n, 0) >= 0);
71
72    mpfr_t t;
73    mpz_t k;
74
75    mpfr_init_set_ui(t, 1, MPFR_RNDN);
76    mpfr_set_ui(R, 1, MPFR_RNDN);
77
78    for(mpz_init_set_ui(k, 1); mpz_cmp(k, n) < 0; mpz_add_ui(k, k, 1))
79    {
80      mpfr_mul(t, t, x, MPFR_RNDN);
81      mpfr_div_z(t, t, k, MPFR_RNDN);
82      mpfr_add(R, R, t, MPFR_RNDN);
83    }
84  }
85
86  double taylor_nat_log(double x, unsigned int n)
87  {
88    //Ensures that both the provided values are positive
89    assert(n > 0);
90    assert(x > 0);
91
92    double a, t, z;
93    int b;
```

```
94
95    //1/2 <= a < 1 and a*2^b = x
96    a = frexp(x, &b);
97    a = 1 - a;
98
99    t = a;
100   z = a;
101
102   //main loop
103   for(int k = 2; k < n; ++k)
104   {
105     //Performs the taylor series update step
106     t *= a;
107     z += t/k;
108   }
109
110   return b*NAT_LOG_2 - z;
111 }
112
113 void mpfr_taylor_nat_log(mpfr_t R, mpfr_t x, mpz_t n)
114 {
115   assert(mpz_cmp_ui(n, 0) > 0);
116   assert(mpfr_cmp_ui(x, 0) > 0);
117
118   mpfr_t a, t, tt;
119   mpfr_exp_t b;
120   mpz_t k;
121   unsigned int f = 1000, F = 1000;
122
123   mpfr_init(a);
124   mpfr_frexp(&b, a, x, MPFR_RNDN);
125   mpfr_ui_sub(a, 1, a, MPFR_RNDN);
126
127   mpfr_init_set(t, a, MPFR_RNDN);
128   mpfr_init(tt);
129   mpfr_set(R, a, MPFR_RNDN);
130
131   for(mpz_init_set_ui(k, 2); mpz_cmp(k, n) < 0; mpz_add_ui(k, k, 1))
132   {
133     mpfr_mul(t, t, a, MPFR_RNDN);
134     mpfr_div_z(tt, t, k, MPFR_RNDN);
135     mpfr_add(R, R, tt, MPFR_RNDN);
136   }
137
138   mpfr_mul_si(a, MPFR_NAT_LOG_2, b, MPFR_RNDN);
139   mpfr_sub(R, a, R, MPFR_RNDN);
140 }
141
142 double taylor_log(double x, double y, unsigned int n)
143 {
144   assert(x > 0);
145   assert(y > 0);
146   assert(n > 0);
147
148   return taylor_nat_log(y, n)/taylor_nat_log(x, n);
149 }
150
151 double taylor_pow(double x, double y, unsigned int n)
```

```
152 {
153   assert(x > 0);
154   assert(n > 0);
155 
156   return taylor_exp(y*taylor_nat_log(x, n), n);
157 }
158 
159 void mpfr_taylor_log(mpfr_t R, mpfr_t x, mpfr_t y, mpz_t n)
160 {
161   assert(mpfr_cmp_ui(x, 0) > 0);
162   assert(mpfr_cmp_ui(y, 0) > 0);
163   assert(mpz_cmp_ui(n, 0)  >= 0);
164 
165   mpfr_t A;
166   mpfr_init(A);
167 
168   mpfr_taylor_nat_log(A, x, n);
169   mpfr_taylor_nat_log(R, y, n);
170 
171   mpfr_div(R, R, A, MPFR_RNDN);
172 }
173 
174 void mpfr_taylor_pow(mpfr_t R, mpfr_t x, mpfr_t y, mpz_t n)
175 {
176   assert(mpfr_cmp_ui(x, 0) > 0);
177   assert(mpz_cmp_ui(n, 0)  > 0);
178 
179   mpfr_t A;
180   mpfr_init(A);
181 
182   mpfr_taylor_nat_log(A, x, n);
183   mpfr_mul(A, y, A, MPFR_RNDN);
184   mpfr_taylor_exp(R, A, n);
185 }
186 
187 #ifdef COMPILE_MAIN
188 int main(int argc, char **argv)
189 {
190   double x, y;
191   unsigned int n, D, p;
192   mpfr_t X, Y, R;
193   mpz_t N;
194   char sf[50];
195   FILE *in;
196 
197   if(argc > 1)
198   {
199     switch(argv[1][0])
200     {
201       case 'a':
202         if(argc == 5 &&
203           sscanf(argv[2], "%lf", &x) == 1 &&
204           sscanf(argv[3], "%u",  &n) == 1 &&
205           sscanf(argv[4], "%u",  &D) == 1)
206           printf("exp(%.*lf) ~= %.*lf (naive)\n",
207             d(D), x, D, naive_exp(x, n));
208         else
209           printf("Usage: %s a <x=Value for exp(x)> <n> "
```

```
210                 "<D=digits to display>\n", argv[0]);
211         break;
212
213       case 'b':
214         if(argc == 5 &&
215           sscanf(argv[2], "%lf", &x) == 1 &&
216           sscanf(argv[3], "%u",  &n) == 1 &&
217           sscanf(argv[4], "%u",  &D) == 1)
218           printf("exp(%.*lf) ~= %.*lf\n",
219             d(D), x, D, taylor_exp(x, n));
220         else
221           printf("Usage: %s b <x=Value for exp(x)> <n> "
222                "<D=digits to display>\n", argv[0]);
223         break;
224
225       case 'c':
226         if(argc == 5 &&
227           sscanf(argv[2], "%lf", &x) == 1 &&
228           sscanf(argv[3], "%u",  &n) == 1 &&
229           sscanf(argv[4], "%u",  &D) == 1)
230           printf("ln(%.*lf) ~= %.*lf\n",
231             d(D), x, D, taylor_nat_log(x, n));
232         else
233           printf("Usage: %s c <x=Value for ln(x)> <n> "
234                "<D=digits to display>\n", argv[0]);
235         break;
236
237       case 'd':
238         if(argc == 6 &&
239           sscanf(argv[2], "%lf", &x) == 1 &&
240           sscanf(argv[3], "%lf", &y) == 1 &&
241           sscanf(argv[4], "%u",  &n) == 1 &&
242           sscanf(argv[5], "%u",  &D) == 1)
243           printf("pow(%.*lf, %.*lf) ~= %.*lf\n",
244             d(D), x, d(D), y, D, taylor_pow(x, y, n));
245         else
246           printf("Usage: %s d <x=Value for pow(x,y)> "
247                "<y=Value for pow(x,y)> <n> "
248                "<D=digits to display>\n", argv[0]);
249         break;
250
251       case 'e':
252         if(argc == 6 &&
253           sscanf(argv[2], "%lf", &x) == 1 &&
254           sscanf(argv[3], "%lf", &y) == 1 &&
255           sscanf(argv[4], "%u",  &n) == 1 &&
256           sscanf(argv[5], "%u",  &D) == 1)
257           printf("log(%.*lf, %.*lf) ~= %.*lf\n",
258             d(D), x, d(D), y, D, taylor_log(x, y, n));
259         else
260           printf("Usage: %s e <x=Value for log(x, y)> "
261                "<y=Value for log(x, y)> <n> "
262                "<D=digits to display>\n", argv[0]);
263         break;
264
265       case 'f':
266         if(argc == 6 &&
267           sscanf(argv[4], "%u", &D) == 1 &&
```

```
268             sscanf(argv[5], "%u", &p) == 1)
269           {
270             mpfr_set_default_prec(p);
271             INIT_CONSTANTS
272
273             if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
274               mpz_init_set_str(N, argv[3], 10) == 0)
275             {
276               mpfr_init(R);
277
278               sprintf(sf, "exp(%%.%uRNf) =~\t(naive)"
279                       "\n\t%%.%uRNf\n", d(D), D);
280
281               mpfr_naive_exp(R, X, N);
282               mpfr_printf(sf, X, R);
283             }
284             else
285               printf("Usage: %s f <X=value for exp(X)> <N> "
286                    "<D=Digits to display> "
287                    "<p=bits of precision>\n", argv[0]);
288           }
289           else
290             printf("Usage: %s f <X=value for exp(X)> <N> "
291                  "<D=Digits to display> "
292                  "<p=bits of precision>\n", argv[0]);
293           break;
294
295         case 'g':
296           if(argc == 6 &&
297             sscanf(argv[4], "%u", &D) == 1 &&
298             sscanf(argv[5], "%u", &p) == 1)
299           {
300             mpfr_set_default_prec(p);
301             INIT_CONSTANTS
302
303             if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
304               mpz_init_set_str(N, argv[3], 10) == 0)
305             {
306               mpfr_init(R);
307
308               sprintf(sf, "exp(%%.%uRNf) =~"
309                       "\n\t%%.%uRNf\n", d(D), D);
310
311               mpfr_taylor_exp(R, X, N);
312               mpfr_printf(sf, X, R);
313             }
314             else
315               printf("Usage: %s g <X=value for exp(X)> <N> "
316                    "<D=Digits to display> "
317                    "<p=bits of precision>\n", argv[0]);
318           }
319           else
320             printf("Usage: %s g <X=value for exp(X)> <N> "
321                  "<D=Digits to display> "
322                  "<p=bits of precision>\n", argv[0]);
323           break;
324
325         case 'h':
```

```
326           if(argc == 6 &&
327             sscanf(argv[4], "%u", &D) == 1 &&
328             sscanf(argv[5], "%u", &p) == 1)
329           {
330             mpfr_set_default_prec(p);
331             INIT_CONSTANTS
332
333             if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
334               mpz_init_set_str(N, argv[3], 10) == 0)
335             {
336               mpfr_init(R);
337
338               sprintf(sf, "ln(%%.%uRNf) =~"
339                       "\n\t%%.%uRNf\n", d(D), D);
340
341               mpfr_taylor_nat_log(R, X, N);
342               mpfr_printf(sf, X, R);
343             }
344             else
345               printf("Usage: %s h <X=value for ln(X)> <N> "
346                    "<D=Digits to display> "
347                    "<p=bits of precision>\n", argv[0]);
348           }
349           else
350             printf("Usage: %s h <X=value for ln(X)> <N> "
351                  "<D=Digits to display> "
352                  "<p=bits of precision>\n", argv[0]);
353           break;
354
355         case 'i':
356           if(argc == 7 &&
357             sscanf(argv[5], "%u", &D) == 1 &&
358             sscanf(argv[6], "%u", &p) == 1)
359           {
360             mpfr_set_default_prec(p);
361             INIT_CONSTANTS
362
363             if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
364               mpfr_init_set_str(Y, argv[3],10, MPFR_RNDN) == 0 &&
365               mpz_init_set_str(N, argv[4], 10) == 0)
366             {
367               mpfr_init(R);
368
369               sprintf(sf, "pow(%%.%uRNf, %%.%uRNf) =~"
370                       "\n\t%%.%uRNf\n", d(D), d(D), D);
371
372               mpfr_taylor_pow(R, X, Y, N);
373               mpfr_printf(sf, X, Y, R);
374             }
375             else
376               printf("Usage: %s i <X=value for pow(X,Y)> "
377                    "<Y=value for pow(X,Y)> <N> "
378                    "<D=Digits to display> "
379                    "<p=bits of precision>\n", argv[0]);
380           }
381           else
382             printf("Usage: %s i <X=value for pow(X,Y)> "
383                  "<Y=value for pow(X,Y)> <N> "
```

```
                "<D=Digits to display> "
                "<p=bits of precision>\n", argv[0]);
        break;

      case 'j':
        if(argc == 7 &&
          sscanf(argv[5], "%u", &D) == 1 &&
          sscanf(argv[6], "%u", &p) == 1)
        {
          mpfr_set_default_prec(p);
          INIT_CONSTANTS

          if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
            mpfr_init_set_str(Y, argv[3],10, MPFR_RNDN) == 0 &&
            mpz_init_set_str(N, argv[4], 10) == 0)
          {
            mpfr_init(R);

            sprintf(sf, "log(%%.%uRNf, %%.%uRNf) =~"
                   "\n\t%%.%uRNf\n", d(D), d(D), D);

            mpfr_taylor_log(R, X, Y, N);
            mpfr_printf(sf, X, Y, R);
          }
          else
            printf("Usage: %s j <X=value for log(X,Y)> "
                 "<Y=value for log(X,Y)> <N> "
                 "<D=Digits to display> "
                 "<p=bits of precision>\n", argv[0]);
        }
        else
          printf("Usage: %s j <X=value for log(X,Y)> "
               "<Y=value for log(X,Y)> <N> "
               "<D=Digits to display> "
               "<p=bits of precision>\n", argv[0]);
        break;

      default:
        printf("Usage: %s <a/b/c/d/e/f/g/h/i/j> <arguments>\n",
          argv[0]);
    }
  }
  else
    printf("Usage: %s <a/b/c/d/e/f/g/h/i/j> <arguments>\n", argv[0]);
}
#endif
```

Code for Hyperbolic Logarithms:

File : hyperbolic_log.c

```
#include <stdio.h>
#include <assert.h>
#include <gmp.h>
#include <mpfr.h>
#include <math.h>

#include "log_exp_utilities.h"
#include "utilities.h"
```

```
9  #include "hyperbolic_log.h"
10
11 #define INIT_CONSTANTS in = fopen(NAT_LOG_2_INFILE, "r"); \
12                mpfr_init(MPFR_NAT_LOG_2); \
13                mpfr_inp_str(MPFR_NAT_LOG_2, in, 10, MPFR_RNDN); \
14                fclose(in);
15
16 mpfr_t MPFR_NAT_LOG_2;
17
18 double hyperbolic_nat_log(double x, unsigned int n)
19 {
20   //Ensuers that x is positive
21   assert(x > 0);
22
23   double t, y, z, a;
24   unsigned int d;
25   int b;
26
27   //Ensures 1/2 <= a < 1 and a*2^b == x
28   a = frexp(x, &b);
29
30   //Sets the initial values to be used
31   t = (a-1)/(a+1);
32   d = 1;
33   y = t*t;
34   z = t;
35
36   //Main loop
37   for(unsigned int k = 1; k <= n; ++k)
38   {
39     //Performs a single Taylor Series update step
40     t *= y;
41     d += 2;
42     z += t/d;
43   }
44
45   return b*NAT_LOG_2 + 2*z;
46 }
47
48 void mpfr_hyperbolic_nat_log(mpfr_t R, mpfr_t x, mpz_t n)
49 {
50   assert(mpfr_cmp_ui(x, 0) > 0);
51   assert(mpz_cmp_ui(n, 0) >= 0);
52
53   mpfr_t t, y, z, tmp, a;
54   mpfr_exp_t b;
55   mpz_t d, k;
56
57   mpfr_init(a);
58   mpfr_init(t);
59   mpfr_init(tmp);
60   mpfr_init(y);
61
62   mpfr_frexp(&b, a, x, MPFR_RNDN);
63
64   mpfr_sub_ui(t, a, 1, MPFR_RNDN);
65   mpfr_add_ui(tmp, a, 1, MPFR_RNDN);
66   mpfr_div(t, t, tmp, MPFR_RNDN);
```

```
67    mpfr_mul(y, t, t, MPFR_RNDN);
68
69    mpfr_init_set(z, t, MPFR_RNDN);
70    mpz_init_set_ui(d, 1);
71
72    for(mpz_init_set_ui(k, 1); mpz_cmp(k, n) <= 0; mpz_add_ui(k, k, 1))
73    {
74      mpfr_mul(t, t, y, MPFR_RNDN);
75      mpz_add_ui(d, d, 2);
76      mpfr_div_z(tmp, t, d, MPFR_RNDN);
77      mpfr_add(z, z, tmp, MPFR_RNDN);
78    }
79
80    mpfr_mul_ui(R, z, 2, MPFR_RNDN);
81    mpfr_mul_si(tmp, MPFR_NAT_LOG_2, b, MPFR_RNDN);
82    mpfr_add(R, R, tmp, MPFR_RNDN);
83  }
84
85  double hyperbolic_log(double x, double y, unsigned int n)
86  {
87    assert(x > 0);
88    assert(y > 0);
89
90    return hyperbolic_nat_log(y, n)/hyperbolic_nat_log(x, n);
91  }
92
93  void mpfr_hyperbolic_log(mpfr_t R, mpfr_t x, mpfr_t y, mpz_t n)
94  {
95    assert(mpfr_cmp_ui(x, 0) > 0);
96    assert(mpfr_cmp_ui(y, 0) > 0);
97    assert(mpz_cmp_ui(n, 0) >= 0);
98
99    mpfr_t A;
100   mpfr_init(A);
101
102   mpfr_hyperbolic_nat_log(R, y, n);
103   mpfr_hyperbolic_nat_log(A, x, n);
104
105   mpfr_div(R, R, A, MPFR_RNDN);
106 }
107
108 #ifdef COMPILE_MAIN
109 int main(int argc, char **argv)
110 {
111   double x, y;
112   unsigned int n, D, p;
113   mpfr_t X, Y, R;
114   mpz_t N;
115   char sf[50];
116   FILE *in;
117
118   if(argc > 1)
119   {
120     switch(argv[1][0])
121     {
122       case 'a':
123         if(argc == 5 &&
124           sscanf(argv[2], "%lf", &x) == 1 &&
```

```
125             sscanf(argv[3], "%u",  &n) == 1 &&
126             sscanf(argv[4], "%u",  &D) == 1)
127             printf("ln(%.*lf) ~= %.*lf\n",
128               d(D), x, D, hyperbolic_nat_log(x, n));
129           else
130             printf("Usage: %s a <x=Value for ln(x)> <n> "
131                  "<D=digits to display>\n", argv[0]);
132           break;
133
134         case 'b':
135           if(argc == 6 &&
136             sscanf(argv[2], "%lf", &x) == 1 &&
137             sscanf(argv[3], "%lf", &y) == 1 &&
138             sscanf(argv[4], "%u",  &n) == 1 &&
139             sscanf(argv[5], "%u",  &D) == 1)
140             printf("log(%.*lf, %.*lf) ~= %.*lf\n",
141               d(D), x, d(D), y, D, hyperbolic_log(x, y, n));
142           else
143             printf("Usage: %s b <x=Value for log(x,y)> "
144                  "<y=Value for log(x,y)> <n> "
145                  "<D=digits to display>\n", argv[0]);
146           break;
147
148         case 'c':
149           if(argc == 6 &&
150             sscanf(argv[4], "%u", &D) == 1 &&
151             sscanf(argv[5], "%u", &p) == 1)
152           {
153             mpfr_set_default_prec(p);
154             INIT_CONSTANTS
155
156             if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
157               mpz_init_set_str(N, argv[3], 10) == 0)
158             {
159               mpfr_init(R);
160
161               sprintf(sf, "ln(%%.%uRNf) =~\n\t%%.%uRNf\n",
162                 d(D), D);
163
164               mpfr_hyperbolic_nat_log(R, X, N);
165               mpfr_printf(sf, X, R);
166             }
167             else
168               printf("Usage: %s c <X=value for ln(X)> <N> "
169                    "<D=Digits to display> "
170                    "<p=bits of precision>\n", argv[0]);
171           }
172           else
173             printf("Usage: %s c <X=value for ln(X)> <N> "
174                  "<D=Digits to display> "
175                  "<p=bits of precision>\n", argv[0]);
176           break;
177
178         case 'd':
179           if(argc == 7 &&
180             sscanf(argv[5], "%u", &D) == 1 &&
181             sscanf(argv[6], "%u", &p) == 1)
182           {
```

```
            mpfr_set_default_prec(p);
            INIT_CONSTANTS

            if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
               mpfr_init_set_str(Y, argv[3], 10, MPFR_RNDN) == 0 &&
                mpz_init_set_str(N, argv[4], 10) == 0)
            {
              mpfr_init(R);

              sprintf(sf, "ln(%%.%uRNf, %%.%uRNf) =~\n\t"
                    "%%.%uRNf\n", d(D), d(D), D);

              mpfr_hyperbolic_log(R, X, Y, N);
              mpfr_printf(sf, X, Y, R);
            }
            else
              printf("Usage: %s d <X=Value for ln(X,Y)> "
                   "<Y=Value for ln(X,Y)> "
                   "<D=Digits to display> "
                   "<p=bits of precision>\n", argv[0]);
          }
          else
            printf("Usage: %s d <X=Value for ln(X, Y)> "
                 "<Y=Value for ln(X,Y)> "
                 "<D=Digits to display> "
                 "<p=bits of precision>\n", argv[0]);
          break;

        default:
          printf("Usage: %s <a/b/c/d> <arguments>\n", argv[0]);
      }
    }
    else
      printf("Usage: %s <a/b/c/d> <arguments>\n", argv[0]);
}
#endif
```

Code for Continued Fraction Exponentials:

File : cont_frac_exp.c

```
#include <stdio.h>
#include <assert.h>
#include <gmp.h>
#include <mpfr.h>
#include <math.h>

#include "log_exp_utilities.h"
#include "utilities.h"
#include "hyperbolic_log.h"
#include "int_exp.h"
#include "cont_frac_exp.h"

#define INIT_CONSTANTS in = fopen(NAT_LOG_2_INFILE, "r"); \
              mpfr_init(MPFR_NAT_LOG_2); \
              mpfr_inp_str(MPFR_NAT_LOG_2, in, 10, MPFR_RNDN); \
              fclose(in); \
              in = fopen(E_CONST_INFILE, "r"); \
              mpfr_init(MPFR_E_CONST); \
```

```
19              mpfr_inp_str(MPFR_E_CONST, in, 10, MPFR_RNDN); \
20              fclose(in); \
21              mpfr_init_set_ui(MPFR_TWO, 2, MPFR_RNDN);
22
23  mpfr_t MPFR_NAT_LOG_2, MPFR_TWO, MPFR_E_CONST;
24
25  double cont_frac_exp_v1(double x, unsigned int n)
26  {
27    double a = x, pA, A, nA, ca, pB, B, nB, cb;
28    unsigned int b = 0;
29
30    if (x < 0)
31      //Calculates the reciprocal if x < 0
32      return 1/cont_frac_exp_v1(-x, n);
33    else if (x == 0)
34      //Basic value check
35      return 1;
36    else if (x > 1)
37      //1/2 <= a < 1 and x == a*2^b
38      a = frexp(x, &b);
39
40    //Initialises the previous values
41    //  pA = A_1, pB = B_1
42    pA = a + 1;
43    pB = 1;
44    //Initialises the current values
45    //  A = A_2, B = B_2
46    A  = a*a + 2*a + 2;
47    B  = 2;
48
49    //Initialises the co-efficients
50    ca = -a;
51    cb = 2 + a;
52
53    for(unsigned int k = 2; k <= n; ++k)
54    {
55      ca -= a;
56      ++cb;
57
58      //Calculates the updates to A and B
59      nA = cb*A + ca*pA;
60      nB = cb*B + ca*pB;
61
62      //Ensures the variables hold the correct values
63      pA = A;
64      pB = B;
65      A  = nA;
66      B  = nB;
67    }
68
69    //If b b==0 then return A/B, otherwise return (A/B)^(2^b)
70    return b ? squaring_int_exp(A/B, (unsigned int)squaring_int_exp(2, b))
71         : A/B;
72  }
73
74  double cont_frac_exp_v2(double x, unsigned int n)
75  {
76    double a = x, pA, A, nA, ca, pB, B, nB;
```

```
77    unsigned int b = 0, cb;
78
79    if (x < 0)
80      //Calculates the reciprocal if x < 0
81      return 1/cont_frac_exp_v1(-x, n);
82    else if (x == 0)
83      //Basic value check
84      return 1;
85    else if (x > 1)
86      //1/2 <= a < 1 and x == a*2^b
87      a = frexp(x, &b);
88
89    //Sets the initial values
90    //  pA = A_1, pB = B_1
91    //  A  = A_2,  B = B_2
92    pA = 1;
93    pB = 1;
94    A  = 1;
95    B  = 1 - a;
96
97    //Initialises the co-efficients
98    ca = 0;
99    cb = 1;
100
101   //Main Loop
102   for(unsigned int k = 3; k <= n; ++k)
103   {
104     ++cb;
105
106     if (k % 2)
107     {
108       ca += a;
109       nA = cb*A + ca*pA;
110       nB = cb*B + ca*pB;
111     }
112     else
113     {
114       nA = cb*A - ca*pA;
115       nB = cb*B - ca*pB;
116     }
117
118     pA = A;
119     pB = B;
120     A  = nA;
121     B  = nB;
122   }
123
124   //If b b==0 then return A/B, otherwise return (A/B)^(2^b)
125   return b ? squaring_int_exp(A/B, (unsigned int)squaring_int_exp(2, b))
126        : A/B;
127 }
128
129 double cont_frac_exp_v3(double x, unsigned int n)
130 {
131   double a = x, pA, A, nA, ca, pB, B, nB;
132   unsigned int b = 0, cb;
133
134   if (x < 0)
```

```
135     //Calculates the reciprocal if x < 0
136     return 1/cont_frac_exp_v1(-x, n);
137   else if (x == 0)
138     //Basic value check
139     return 1;
140   else if (x > 1)
141     //1/2 <= a < 1 and x == a*2^b
142     a = frexp(x, &b);
143
144   //Sets the initial values
145   //  pA = A_0, pB = B_0
146   //  A  = A_1,  B = B_1
147   pA = 1;
148   pB = 1;
149   A  = 2 + a;
150   B  = 2 - a;
151
152   //Initialises the co-efficients
153   ca = a*a;
154   cb = 2;
155
156   //Main loop
157   for(unsigned int k = 2; k <= n; ++k)
158   {
159     cb += 4;
160
161     nA = cb*A + ca*pA;
162     nB = cb*B + ca*pB;
163
164     pA = A;
165     pB = B;
166     A  = nA;
167     B  = nB;
168   }
169
170   //If b b==0 then return A/B, otherwise return (A/B)^(2^b)
171   return b ? squaring_int_exp(A/B, (unsigned int)squaring_int_exp(2, b))
172        : A/B;
173 }
174
175 void mpfr_cont_frac_exp_v3(mpfr_t R, mpfr_t x, mpz_t n)
176 {
177   assert(mpz_cmp_ui(n, 0) >= 0);
178
179   mpfr_t a, pA, A, nA, ca, pB, B, nB, C, tmp1, tmp2;
180   mpfr_exp_t b = 0;
181   mpz_t N, k, cb;
182
183   if (mpfr_cmp_ui(x, 0) < 0)
184   {
185     mpfr_neg(x, x, MPFR_RNDN);
186     mpfr_cont_frac_exp_v3(R, x, n);
187     mpfr_ui_div(R, 1, R, MPFR_RNDN);
188   }
189   else if (mpfr_cmp_ui(x, 0) == 0)
190   {
191     mpfr_set_ui(R, 1, MPFR_RNDN);
192   }
```

```
193     else
194     {
195       mpfr_init(a);
196       if (mpfr_cmp_ui(x, 1) > 0)
197       {
198         mpfr_frexp(&b, a, x, MPFR_RNDN);
199         mpz_init(N);
200         mpfr_init(C);
201         mpz_ui_pow_ui(N, 2, b);
202       }
203       else
204       {
205         mpfr_init_set_ui(C, 0, MPFR_RNDN);
206         mpfr_set(a, x, MPFR_RNDN);
207       }
208
209       mpfr_init_set_ui(pA, 1, MPFR_RNDN);
210       mpfr_init_set_ui(pB, 1, MPFR_RNDN);
211       mpfr_init_set_ui(A,  2, MPFR_RNDN);
212       mpfr_init_set_ui(B,  2, MPFR_RNDN);
213       mpfr_add(A, A, a, MPFR_RNDN);
214       mpfr_sub(B, B, a, MPFR_RNDN);
215
216       mpfr_init_set(ca, a, MPFR_RNDN);
217       mpfr_mul(ca, ca, a, MPFR_RNDN);
218       mpz_init_set_ui(cb, 2);
219
220       mpfr_init(nA);
221       mpfr_init(nB);
222       mpfr_init(tmp1);
223       mpfr_init(tmp2);
224
225       for(mpz_init_set_ui(k, 2); mpz_cmp(k, n) <= 0; mpz_add_ui(k, k, 1))
226       {
227         mpz_add_ui(cb, cb, 4);
228
229         mpfr_mul_z(tmp1, A, cb, MPFR_RNDN);
230         mpfr_mul(tmp2, ca, pA, MPFR_RNDN);
231         mpfr_add(nA, tmp1, tmp2, MPFR_RNDN);
232
233         mpfr_mul_z(tmp1, B, cb, MPFR_RNDN);
234         mpfr_mul(tmp2, ca, pB, MPFR_RNDN);
235         mpfr_add(nB, tmp1, tmp2, MPFR_RNDN);
236
237         mpfr_set(pA, A, MPFR_RNDN);
238         mpfr_set(pB, B, MPFR_RNDN);
239         mpfr_set(A, nA, MPFR_RNDN);
240         mpfr_set(B, nB, MPFR_RNDN);
241       }
242
243       mpfr_div(R, A, B, MPFR_RNDN);
244
245       if(b)
246       {
247         mpfr_set(A, R, MPFR_RNDN);
248         mpfr_squaring_int_exp(R, A, N);
249       }
250     }
```

```
251 }
252
253 double improved_pow(double x, double y, unsigned int n)
254 {
255   assert(x > 0);
256
257   return cont_frac_exp_v3(y * hyperbolic_nat_log(x, n), n);
258 }
259
260 void mpfr_improved_pow(mpfr_t R, mpfr_t x, mpfr_t y, mpz_t n)
261 {
262   assert(mpz_cmp_ui(n, 0) >= 0);
263   assert(mpfr_cmp_ui(x, 0) > 0);
264
265   mpfr_t A;
266   mpfr_init(A);
267
268   mpfr_hyperbolic_nat_log(A, x, n);
269   mpfr_mul(A, y, A, MPFR_RNDN);
270   mpfr_cont_frac_exp_v3(R, A, n);
271 }
272
273 #ifdef COMPILE_MAIN
274 int main(int argc, char **argv)
275 {
276   double x, y;
277   unsigned int n, D, p;
278   mpfr_t X, Y, R;
279   mpz_t N;
280   char sf[50];
281   FILE *in;
282
283   if(argc > 1)
284   {
285     switch(argv[1][0])
286     {
287       case 'a':
288         if(argc == 5 &&
289           sscanf(argv[2], "%lf", &x) == 1 &&
290           sscanf(argv[3], "%u",  &n) == 1 &&
291           sscanf(argv[4], "%u",  &D) == 1)
292           printf("exp(%.*lf) ~= %.*lf (v1)\n",
293             d(D), x, D, cont_frac_exp_v1(x, n));
294         else
295           printf("Usage: %s a <x=Value for exp(x)> <n> "
296                "<D=Digits to display\n", argv[0]);
297         break;
298
299       case 'b':
300         if(argc == 5 &&
301           sscanf(argv[2], "%lf", &x) == 1 &&
302           sscanf(argv[3], "%u",  &n) == 1 &&
303           sscanf(argv[4], "%u",  &D) == 1)
304           printf("exp(%.*lf) ~= %.*lf (v2)\n",
305             d(D), x, D, cont_frac_exp_v2(x, n));
306         else
307           printf("Usage: %s b <x=Value for exp(x)> <n> "
308                "<D=Digits to display\n", argv[0]);
```

```
309           break;
310
311         case 'c':
312           if(argc == 5 &&
313             sscanf(argv[2], "%lf", &x) == 1 &&
314             sscanf(argv[3], "%u",  &n) == 1 &&
315             sscanf(argv[4], "%u",  &D) == 1)
316             printf("exp(%.*lf) ~= %.*lf (v3)\n",
317               d(D), x, D, cont_frac_exp_v3(x, n));
318           else
319             printf("Usage: %s c <x=Value for exp(x)> <n> "
320                 "<D=Digits to display\n", argv[0]);
321           break;
322
323         case 'd':
324           if(argc == 6 &&
325             sscanf(argv[2], "%lf", &x) == 1 &&
326             sscanf(argv[3], "%lf", &y) == 1 &&
327             sscanf(argv[4], "%u",  &n) == 1 &&
328             sscanf(argv[5], "%u",  &D) == 1)
329             printf("pow(%.*lf, %.*lf) ~= %.*lf\n",
330               d(D), x, d(D), y, D, improved_pow(x, y, n));
331           else
332             printf("Usage: %s d <x=Value for pow(x,y)> "
333                 "<y=Value for pow(x,y)> <n> "
334                 "<D=Digits to display>\n", argv[0]);
335           break;
336
337         case 'e':
338           if(argc == 6 &&
339             sscanf(argv[4], "%u", &D) == 1 &&
340             sscanf(argv[5], "%u", &p) == 1)
341           {
342             mpfr_set_default_prec(p);
343             INIT_CONSTANTS
344
345             if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
346               mpz_init_set_str(N, argv[3], 10) == 0)
347             {
348               mpfr_init(R);
349
350               sprintf(sf, "exp(%%.%uRNf) =~\n\t%%.%uRNf\n",
351                 d(D), D);
352
353               mpfr_cont_frac_exp_v3(R, X, N);
354               mpfr_printf(sf, X, R);
355             }
356             else
357               printf("Usage: %s e <X=Value for exp(X)> <N> "
358                   "<D=Digits to display> "
359                   "<p=bits of precision>\n", argv[0]);
360           }
361           else
362             printf("Usage: %s e <X=Value for exp(X)> <N> "
363                 "<D=Digits to display> "
364                 "<p=bits of precision>\n", argv[0]);
365           break;
366
```

```
      case 'f':
        if(argc == 7 &&
          sscanf(argv[5], "%u", &D) == 1 &&
          sscanf(argv[6], "%u", &p) == 1)
        {
          mpfr_set_default_prec(p);
          INIT_CONSTANTS

          if(mpfr_init_set_str(X, argv[2], 10, MPFR_RNDN) == 0 &&
             mpfr_init_set_str(Y, argv[3], 10, MPFR_RNDN) == 0 &&
              mpz_init_set_str(N, argv[4], 10) == 0)
          {
            mpfr_init(R);

            sprintf(sf, "pow(%%.%uRNf, %%.%uRNf) =~"
                    "\n\t%%.%uRNf\n",
                d(D), d(D), D);

            mpfr_improved_pow(R, X, Y, N);
            mpfr_printf(sf, X, Y, R);
          }
          else
            printf("Usage: %s f <X=Value for pow(X, Y)> "
                 "<Y=Value for pow(X,Y)> <N> "
                 "<D=Digits to display> "
                 "<p=bits of precision>\n", argv[0]);
        }
        else
          printf("Usage: %s f <X=Value for pow(X, Y)> "
               "<Y=Value for pow(X,Y)> <N> "
               "<D=Digits to display> "
               "<p=bits of precision>\n", argv[0]);
        break;

      default:
        printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
    }
  }
  else
    printf("Usage: %s <a/b/c/d/e/f> <arguments>\n", argv[0]);
}
#endif
```

Header Files for Exponential and Logarithmic Functions:

File : int_exp.h

```
#ifndef INT_EXP_HEADER
#define INT_EXP_HEADER

  double naive_int_exp(const double, const int);
  double squaring_int_exp(const double, const int);
  void mpfr_naive_int_exp(mpfr_t, mpfr_t, mpz_t);
  void mpfr_squaring_int_exp(mpfr_t, mpfr_t, mpz_t);

#endif
```

File : taylor_exp_log.h

```
#ifndef TAYLOR_EXP_LOG_HEADER
```

```
#define TAYLOR_EXP_LOG_HEARDR

   double naive_exp(double, unsigned int);
   void mpfr_naive_exp(mpfr_t, mpfr_t, mpz_t);

   double taylor_exp(double, unsigned int);
   double taylor_nat_log(double, unsigned int);

   double taylor_log(double, double, unsigned int);
   double taylor_pow(double, double, unsigned int);

   void mpfr_taylor_exp(mpfr_t, mpfr_t, mpz_t);
   void mpfr_taylor_nat_log(mpfr_t, mpfr_t, mpz_t);

   void mpfr_taylor_log(mpfr_t, mpfr_t, mpfr_t, mpz_t);
   void mpfr_taylor_pow(mpfr_t, mpfr_t, mpfr_t, mpz_t);

#endif
```

File : hyperbolic_log.h

```
#ifndef HYPERBOLIC_LOG_HEADER
#define HYPERBOLIC_LOG_HEADER

   double hyperbolic_nat_log(double, unsigned int);
   double hyperbolic_log(double, double, unsigned int);
   void mpfr_hyperbolic_nat_log(mpfr_t, mpfr_t, mpz_t);
   void mpfr_hyperbolic_log(mpfr_t, mpfr_t, mpfr_t, mpz_t);

#endif
```

File : cont_frac_exp.h

```
#ifndef CONT_FRAC_EXP_HEADER
#define CONT_FRAC_EXP_HEADER

   double cont_frac_exp_v1(double, unsigned int);
   double cont_frac_exp_v2(double, unsigned int);
   double cont_frac_exp_v3(double, unsigned int);
   void mpfr_cont_frac_v3(mpfr_t, mpfr_t, mpz_t);

   double improved_pow(double, double, unsigned int);
   void mpfr_impfroved_pow(mpfr_t, mpfr_t, mpfr_t, mpz_t);

#endif
```