Accuracy vs Efficiency of Numerical Methods

How to program a Calculator

Jake Darby

Abstract

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

1 Introduction

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

Nowadays we have computers and calculators to calculate functions such as square roots or the trigonometric functions. These devices however must be told how to correctly calculate these values, those methods used will be looked at here. Specifically, for most purposes it does not matter if some small error is present in the results, as long as the value found is close enough to the actual value.

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 int 1969-1970, which has lead to several very efficient compilers.

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

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

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

- 1 bit for the sign of the number, s
- 11 bits for the exponent, e
- 52 bits for the significand, $b = b_0 b_1 b_2 \ldots b_{51}$

- A value that is understood to be:

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

This gives a `double` value a precision of around 15-17 significant decimal digits. While this is good for most applications, there are applications where we may want even more precision than this. To solve this I will be implementing certain algorithms using the GNU Multiple Precision Arithmetic Library (reffered to as GMP) as well as GNU MPFR Library(reffered to as GMP), wich 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.

The downside is that the increased precision does increase computation time for these calculations.

I will be compiling and testing all of my code on a benchmark machine running a light version of Ubuntu ¡VERSION¿, using the GNU C Compiler. The specifications of the machine, that may impact perrformance 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

TODO: Expand the introduction and make it more eloquent

2 General Definitions and Theorems

This section will discuss those definitions and theorems that will be applicable and refferenced later in the document.

2.1 Methods

In this document we will look at various functions, such as root functions, trigonometric functions, among others. Despite the variety of functions being analysed there are several methods that are useful for more than one function, we will discuss these methods below.

2.1.1 Newton-Raphson Method

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

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

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

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

The hope is that, if we start with a good initial guess, that the method will converge to some $x \in \mathbb{R} : f(x) = 0$.

We derive the above method in the following way. If we have an approximation $x_n \in \mathbb{R} : f(x_n) \approx 0$ we consider the tangent to $f(x)$ at x_n, which is given by $y = f'(x_n)(x - x_n) + f(x_n)$. If we set $y = 0$ and solve for x we get that

$$x = x_n - \frac{f(x)}{f'(x)}$$

The iterative formula follows by letting $x_{n+1} := x$ in the above.

TODO: Re-arrange into a better order
TODO: Refferences
TODO: Possibly include discussions of pros and cons

2.1.2 Taylor Series Expansion

The Taylor Series formulation was created by Brook Taylor in 1715, based off of the work of Scottish mathematician James Gregory. The Taylor Series describes a method of representing a given function by a polynomial, where any finite number of terms will give an approximation to the function itself.

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

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

We can then, from our definition of a Taylor Series, 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

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

Some examples of simple Taylor Series are:

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

TODO: Eloquate this section
TODO: Discuss when a Taylor Series is not equal to it's function
TODO: Describe intervals for which Taylor Series converge
TODO: Create more examples of Taylor Series that are not used later in the document
TODO: Possibly prove Taylor's Theorem

2.1.3 CORDIC

TODO: Describe the CORDIC Algorithm in general
TODO: Let other sections define specific changes to CORDIC

2.2 Errors

Errors are very useful in this document for discussing convergence and measuring the accuracy of a particular method. There are two distinct types of error that we are interested in:

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 will be useful in guaranteeing a certain level of accuracy that a given implementation of a method will give. Uses of absolute error in the document will use ϵ as their absolute error variable.

Definition 2.2.2. If we have a value v and it's approximation $\tilde{v}$, then the relative error is

$$\eta := \frac{\epsilon}{|v|} = \left|1 - \frac{\tilde{v}}{v}\right|$$

The relative error will also be useful in guaranteeing a certain level of accuracy that a given implementation of a method will give, particularly when the absolute error varies with the size of the input. Uses of relative error in the document will use η as their absolute error variable.

As the previous two errors are hard or impossible to accurately estimate, in a practical manner, then we want an error estimate that we can calculate in an algorithm. Thus we define the iteration error, as the absolute difference of consecutive iterations.

Definition 2.2.3. If we have the sequence $(x_n)_{n\in\mathbb{N}}$, then the iteration error is defined as:

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

For our practical purposes we can note that it is almost impossible to calculate ϵ_n exactly, while δ_n is easy to calculate and tends to be a good enough approximation of ϵ_n; particularly if the convergence is rapid.

2.3 Convergence

definition
In this section we consider what it means for a sequence to converge to a limit value, and some results that will be useful in future chapters.

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

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

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

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

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

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

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

For $\Leftarrow$:
Omitted for brevity. ☐

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

$$\lim_{n\to\infty} \frac{|x_{n+1} - x|}{|x_n - x|^\alpha} = \lambda$$

For the following proof we will have $\epsilon_n := |x^* - x_n|$.

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

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

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

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

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

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

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

Then we get the following derivation:

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

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

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

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

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

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

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

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

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

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

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

$\square$

2.4 Efficiency and Accuracy Metrics

3 Division and Multiplication

3.1 Introduction

Though the idea of Division and Multiplication can seem fairly simple, particularly from an abstract pure mathematical point of view, these calculations can be computationally difficult. This section will show a few algorithms designed to calculated these values, and discuss their implementation and efficiency.

3.2 Multiplication

3.2.1 Basic Methods

3.2.2 Advanced Methods

3.2.3 Analysis

3.3 Division

3.3.1 Basic Methods

3.3.2 Advanced Methods

3.3.3 Analysis

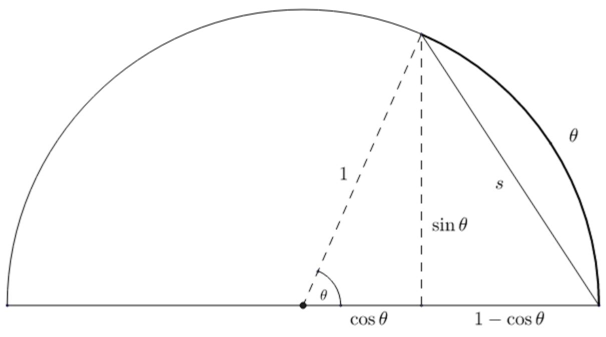
TODO: Section to be filled out, no work currently done on this section

4 Trigonometric Functions

4.1 Geometric Method

The first method I will be discussing is a method based on geometric properties that are derived on a circle, and we will start by considering values of $\cos$ in the range $[0, \frac{\pi}{2})$. To do this we will consider the following figure of the unit circle:

Figure 4.1.1: Diagram showing angles to be dealt with

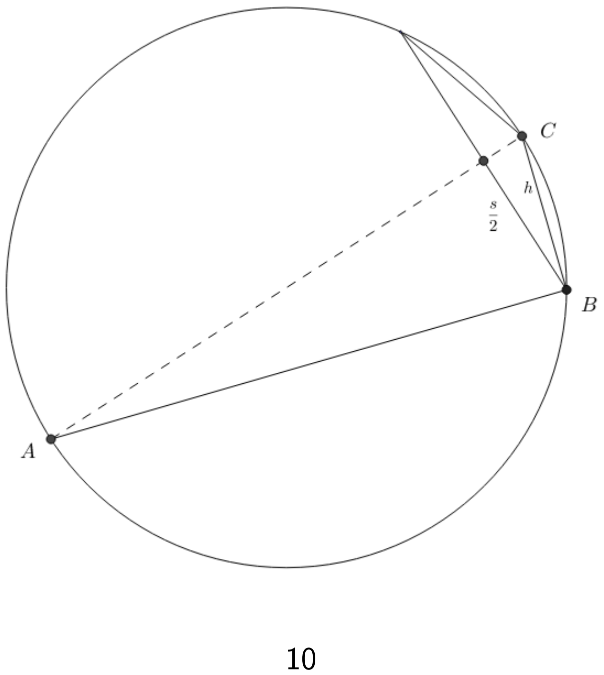


Here theta will be given in radians, and we can note that the labelled arc has length θ due the formula for the circumference of a circle. By using the following derivation we can find a formula for θ in terms of s:

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

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

Figure 4.1.2: Diagram detailing how to calculate s

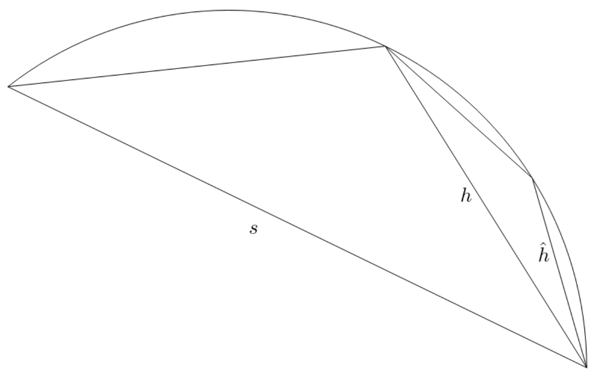


We will first note that by an elementary geometry result we can know that the angle ABC is a right-angle; also we can consider that h is an approximation of $\frac{\theta}{2}$, which will become relevant later. Now because AC is a diameter of our circle then it's length is 2 and thus, by utilising Pythagarus'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

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

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

Figure 4.1.3: Detailing the recursive steps



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

Algorithm 4.1.1: Geometric calculation of $\cos$

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

4.2 Taylor Series

4.3 CORDIC

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

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

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

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

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

5 Root Functions

In this section of the document we will consider several methods for approximating root functions. For our purposes 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 $\sqrt{N} = i\sqrt{|N|}$.

5.1 Digit by Digit Method

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

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

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

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

This allows us to get successive apporximations 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 alllow us to create an algorithm that will give successive approximations of $sqrtN = a_0 10^n + a_1 10^{n-1} + \dots$, more importantly each approximation will give us the exact next digit in the decimal representation of $\sqrt{N}$.

Thus we can have an iterative method to solve the problem, where at each stage we are trying to find the largest digit which satisfies the inequality $(20\sum_{i=0}^{k-1} a_i 10^{k-i-1} + a_k)a_k 10^{2n-2k} \leq N - N_{k-1}$. Thus we get the following pseudocode, 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 5.1.1: Exact Digit by Digits Square Root

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

```
10                  $Digits_b \mapsto Digits_b \cup \{a_k\}$
11              else:
12                  $Digits_a \mapsto Digits_a \cup \{a_k\}$
13              $k \mapsto k+1$
14          if $Digits_a = \emptyset$:
15              $Digits_a := \{0\}$
16          if $Digits_b = \emptyset$:
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 $k = 0 \to d$. We will see that by considering some changes to the algorithm we can change the complexity class to be $\mathcal{O}(d)$.

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

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

Algorithm 5.1.2: Exact Digit by Digits Square Root version 2

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

```
        return ($Digits_a, Digits_b$)
```

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

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

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

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

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

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

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

Now we can consider our choice of e, the most practical method is to test successeive $e_m := 2^m$, 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 5.1.3: Integer Square Root Algorithm

```
1   integerSquareRoot($N \in \mathbb{Z}_0^+$):
2       $M := N$
3       $m := \max m \in \mathbb{Z}_0^+ : 4^m \leq M$
4       $r := 0$
5       while $m \geq 0$:
6           if $2r(2^m) + 4^m \leq M$:
7               $M \mapsto M - 2r(2^m) + 4^m$
8               $r \mapsto r + 2^m$
9           $m \mapsto m - 1$
10      return $r$
```

If we now conisder 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 algorihtm 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 bitshift operations. Finally we will use `num` to represent M.

Now that we have discussed the setup 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 bitshifts 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 bitshifting and a bitshift 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 acheived 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 implementaion of this in C using the unsigned 32 bit integer type `uint32_t`. A more commented and slightly modified version can be found in Appendix **??**, File **??**.

```
uint32_t int_sqrt(uint32_t num)
```

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

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

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

                bit = bit >> 2;
        }

        return res;
}
```

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

Now that the algorithm is correctly constructed using simple unsigned integer addition, subtraction and bitshifting (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 opertaions, including assignments, comparrisons and calucluations. This means that the integer square root of any number up to 4294967295 can be calculated extremely quickly.

5.2 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 pseudocode 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 5.2.1: General Bisection Method

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

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

Algorithm 5.2.2: Bisection Method for Square Roots

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

The implementation of this method is efficiently acheived 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 5.2.1. $\lim_{n\to\infty} x_n = \sqrt{N}$ *for Algorithm 5.2.2*

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

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

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

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

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

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

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

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

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

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

Algorithm 5.2.3: Bisection Method for General Roots

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

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

We can now consider the accuracy that can be acheived 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 $\epsilon_n := \left| x_n - \sqrt{N} \right| \leq \frac{1}{2}(b_n - a_n) \forall n \in \mathbb{N}$. Then as we know that $b_{n+1} - a_{n+1} = \frac{1}{2}(b_n - a_n)$, we know that $\epsilon_n \leq \frac{1}{2^n}(b_0 - a_0)$.

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

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

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

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

Algorithm 5.2.4: Bisection Method for Square Roots with fixed bounds

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

5.3 Newton's Method for Square Roots

If we consider $f(x) = x^2 - N$ then if x^* is a solution to $f(x) = 0$ we see that $x^* = \sqrt{N}$. As $f'(x) = 2x$, then the Newton's Method, will give $x_{n+1} = x_n - \frac{x^2 - N}{2x}$, where x_0 is a given initial guess.

We can see that, in C, each iteration will calculate `x = x - (x*x - N) / (2*x)`, which requires 5 operations; however if we re-arrange our equation, we instead get $x_{n+1} = \frac{1}{2}x_n + \frac{N}{x}$. Implementing our new iterative formula we get `x = 0.5 * (x + N/x)`, which now uses only 3 operations.

We can then use the following pseudocode as the basis of our implementaions of the Newton-Raphson Method for Square Roots:

Algorithm 5.3.1: Basic Newton Method for Square Root

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

$n \mapsto n+1$

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

Proposition 5.3.1. *If $x_0 \in \sqrt{N}, \infty$ and $\{x_n : n \in \mathbb{N}\}$ is a sequnence of approximations of $\sqrt{N}$ found via the Newton-Raphson Method, as detailed above, then:*

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

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

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

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

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

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

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

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

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

This then leads to an updated version of the above pseudocode:

Algorithm 5.3.2: Basic Newton Method for Square Root

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

TODO: Write up examination and implementation of this pseudocode

An alternative would be to use the integer square root method discussed in Section 5.1 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 5.3.2. *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 $I \mathbb{R}^+$ and so we cannot guarantee the satisfaction of NR_1. Therefore we can proceed with our analysis of the case that $N > 1$.

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

We can guarantee that $\epsilon_0 \leq 1$ because $x_0 \in I$ from the integer square root algortihm; 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 pseudocode:

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

Algorithm 5.3.3: Basic Newton Method for Square Root

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

TODO: Write up examination of different versions tried, such as using $x_0 = N$, etc...

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

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

We then get the following algorithm, which implements this:

Algorithm 5.3.4: Newton Method for Square Root v3

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

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

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

Proposition 5.3.3. *Algorithm 5.3.4, satisfies the criteria of Theorem 2.3.2, and thus has quadratic convergence to $\sqrt{N}$.*

Proof. To fulfill 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 5.3.2 we get that NR_1 and NR_2 ar 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:

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

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 acheive 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 arrise from multiplications afterwards).

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

In terms of efficiency versus accuracy tradeoff modifying the problem thus would improve it's efficiency by removing, now unneccesary, calculation and comparrison 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

5.4 Newton's Inverse Square Root Method

As discussed in Section **??**, computers are more efficient at multiplication over division. We would therefore prefer to find a way of utilising Neton's Method without having to perform any costly 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 5.3.4, 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 transormations.

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

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

Algorithm 5.4.1: Newton Inverse Square Root Method

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

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

Proposition 5.4.1. *Algorithm 5.4.1 satisfies the criteria of Theorem 2.3.2, and thus has quadratic convergence to $\sqrt{N}$.*

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

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

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

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

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

Now we consider $M\epsilon_0$:

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

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

5.5 Comparrison 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 theory and practice. The exact root method that we first discussed is the hardest to compare to the other methods as it works in a very different manner to the others. For now we will merely observe that it is an inefficient method that can be will be shown to tae longer than the others.

Now for the other methods we will be comparing Algorithms 5.2.4, 5.3.4 and 5.4.1; we will first consider their computational complexity. As all of these methods have a variable number of iterations that they can perform then we need to consider the complexity fore each iteration step.

First for the bisection method if we calculate $x_n^2 - N$ (which takes 3 operations) whenever we generate n, and as we also calculate $x_n := \frac{1}{2}(a_n + b_n)$ each iteration. Then the total number of operations for each iteration is 6.

Next the regular newton method calculates $\frac{1}{2}(x_n + \frac{a}{x_n})$ and $|x_n + 1 - x_n|$ each iteration whic take 3 and 3 operations respectively, giving a total of 6 operations per iteration; however as previously noted one of those operations is a costly division operation. Finally the inverse newton method calculates $x_n(\frac{3}{2} + \frac{a}{2}x_n^2$ and $|x_n + 1 - x_n|$ each iteration which take 4 and 3 opertaions respectively, giving a total of 7 operations per iteration.

As we can see the number of operations per iteration are similar for each of these methods and so the number of required iterations will have a large impact on the speed of these methdos. Now 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

6 Logarithms and Exponentials

6.1 Taylor Series Method

6.2 Hyperbolic Series Method

6.3 CORDIC

TODO: Fill this out with stuff

7 Preliminary References

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

A Code

Code for Digit by Digit Square root method:

File : exact_root.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 here as placeholder
  //  this may be replaced by my own log function later
  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)
  {
    R[0] = '0';
    R[1] = '.';
    //Offset set to 2 to indicate there are 2 pre-assigned characters
    o = 2;
  }
```

```
//Main loop
for(i=0; i <= D; i++)
{
  //Calculates 10^(2t) and 20*P
  mpfr_mul(tmpr_0, T, T, MPFR_RNDN);
  mpz_mul_ui(tmpz, P, 20);
  //tmpr_1 is used to prevent re-calculation later on
  mpfr_set_ui(tmpr_1, 0, MPFR_RNDN);

  /*
  This loop stops when any digit produces a Y too large or all
    digits have been conisdered.
  In both cases a will be one greater than required and thus
    must be decremented afterwards
  */
  for(a=1; a <= 9; a++)
  {
    //Calculates N - (20*P + a)*a*10^(2t)
    mpz_add_ui(Yz, tmpz, a);
    mpz_mul_ui(Yz, Yz, a);
    mpfr_mul_z(Yr, tmpr_0, Yz, MPFR_RNDN);

    if(mpfr_cmp(Yr, Nr) > 0)
      //The exit condition for the loop has been met
      break;
    else
      //tmpr_1 updated to remove the need for re-calculation
      mpfr_set(tmpr_1, Yr, MPFR_RNDN);
  }

  //Decrements a and adds the correct digit to the result string
  a--;
  R[i+o] = DIGITS[a];

  //Reduces Nr by the largest Yr found in the previous loop
  mpfr_sub(Nr, Nr, tmpr_1, MPFR_RNDN);

  //Break if an exact solution is found
  //Note that due to the representation of floating point numbers it
  //  is possible to have found an exact solution with a positive
  //  remainder that is very close to zero. Unfortunately there is
  //  no way to test for this without knowing, the exact precision
  //  of the input beforehand.
  if(mpfr_cmp_ui(Nr, 0) == 0)
  {
    //This loop adds 0s to a string where an exact solution has
    //  been found but needs right padding with zeros.
    while(mpfr_cmp_ui(T, 1) > 0)
    {
      R[++i + o] = '0';
      mpfr_div_ui(T, T, 10, MPFR_RNDN);
    }
    break;
  }

  //Calculates P = 10*P + a
  mpz_mul_ui(P, P, 10);
```

```
    mpz_add_ui(P, P, a);
    //Calculates T = T/10 => 10^t -> 10^(t-1)
    mpfr_div_ui(T, T, 10, MPFR_RNDN);

    //If we have dropped below 10^0 for the first time then add
    //  a '.' to the result string and increase the offset to 1
    //This case only occurs if no '.' is in the string already
    if(o == 0 && mpfr_cmp_ui(T, 1) < 0)
    {
      o = 1;
      R[i+o] = '.';
    }
  }

  //Adds a null character to terminate the string
  R[i+o+1] = '\0';
  return R;
}

//The use of uintmax_t gives the largest number of unsigned integers
//  for which this function will work with.
uintmax_t uint_sqrt(uintmax_t num)
{
  //Represents the value of 2r(2^m), where r is the
  //  current known part of the integer root
  uintmax_t res = 0;
  //Represents the largest power of (2^m)^2 = 4^m, the initial value
  //  is calculated as 011...11 XOR 0011...11 as the size
  //  of uintmax_t is not known beforehand
  uintmax_t bit = (UINTMAX_MAX >> 1) ^ (UINTMAX_MAX >> 2);

  //Finds the largest power of 4 that is at most 'num' in value
  while(bit > num)
    bit >>= 2;

  //while(bit) is equivalent to while(bit > 0) for unsigned integers
  while(bit)
  {
    //Checks the two cases for updating 'res' and 'num'
    if(num >= res + bit)
    {
      //'num' is used to keep track of the difference betweek
      //  r and the original value, N, that was to be rooted.
      num -= res + bit;
      //This calculates 'res' -> 2(r + 2^m)*2^(m-1) using addition
      //  and bitshifting
      res = (res >> 1) + bit;
    }
    //In the other case 'res' -> 2r(2^m-1)
    else
      res >>= 1;

    //Move on to the next lower power of 2
    bit >>= 2;
  }

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

```c
}

#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 Method:

File : bisect_root.c

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

```

```c
#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;

  N = frexp(N, &e);
  if(e%2)
  {
    N /= 2;
    e += 1;
  }

  printf("N=%lf\ne=%d\n", N, e);
  //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 iPow(double x, unsigned int n)
{
  double r = 1;
  while(n--)
    r *= x;
  return r;
}

double bisect_nRoot(double N, double T, unsigned int n)
```

```
{
  assert(N >= 0);
  assert(T >= 0);
  //Ensures that none of the trivial cases are requested
  assert(n >= 2);

  //Runs the more optimal bisect_sqrt if n == 2
  if(n == 2)
    return bisect_sqrt(N, T);

  double a, b, x, f;

  //Sets the initial values of a and b
  a = 0;
  //This statement is equivalent to
  //  if(N<1) b=1; else b=N;
  b = N < 1 ? 1 : N;

  x = 0.5*(a + b);
  f = iPow(x, n) - 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 = iPow(x, n) - N;
  }

  return x;
}

void mpfr_bisect_sqrt(mpfr_t R, mpfr_t N, mpfr_t T)
{
  if(mpfr_cmp_ui(N, 0) < 0)
  {
    fprintf(stderr, "The value to square root must be non-negative\n");
    exit(-1);
  }
  if(mpfr_cmp_ui(T, 0) < 0)
  {
    fprintf(stderr, "The tolerance must be non-negative\n");
    exit(-1);
  }

  mpfr_exp_t e;
  mpfr_t a, b, x, f, d, fab, n;

  mpfr_init(n);
  mpfr_frexp(&e, n, N, MPFR_RNDN);
  if(e%2)
  {
    mpfr_div_ui(n, n, 2, MPFR_RNDN);
    e += 1;
  }

  //Set a == 0
  mpfr_init_set_ui(a, 0, MPFR_RNDN);

  //Set b == 1
  mpfr_init_set_ui(b, 1, MPFR_RNDN);

  //Set x = (a + b)/2
  mpfr_init(x);
  mpfr_add(x, a, b, MPFR_RNDN);
  mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);

  //Set f = x^2 - N and fab = |f|
  mpfr_init(f);
  mpfr_init(fab);
  mpfr_mul(f, x, x, MPFR_RNDN);
  mpfr_sub(f, f, N, MPFR_RNDN);
  mpfr_abs(fab, f, MPFR_RNDN);

  //Set d = b - a
  mpfr_init(d);
  mpfr_sub(d, b, a, MPFR_RNDN);

  while(mpfr_cmp(fab, T) > 0 && mpfr_cmp(d, T) > 0)
  {
    //Update the bounds, a and b
    if(mpfr_cmp_ui(f, 0) < 0)
      mpfr_set(a, x, MPFR_RNDN);
    else
      mpfr_set(b, x, MPFR_RNDN);

    //Update x
    mpfr_add(x, a, b, MPFR_RNDN);
    mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);

    //Update f and fab
    mpfr_mul(f, x, x, MPFR_RNDN);
    mpfr_sub(f, f, n, MPFR_RNDN);
    mpfr_abs(fab, f, MPFR_RNDN);
  }

  printf("beep");
  mpfr_mul_2si(R, x, e/2, MPFR_RNDN);
}

void mpfr_bisect_nRoot(mpfr_t R, mpfr_t N, mpfr_t T, unsigned int n)
{
  if(mpfr_cmp_ui(N, 0) < 0)
  {
    fprintf(stderr, "The value to square root must be non-negative\n");
    exit(-1);
  }
  if(mpfr_cmp_ui(T, 0) < 0)
  {
    fprintf(stderr, "The tolerance must be non-negative\n");
    exit(-1);
  }
  assert(n >= 2);

  mpfr_t a, b, x, f, d, fab;

  //Set a == 0
  mpfr_init_set_ui(a, 0, MPFR_RNDN);

  //Set b = max{1, N}
  mpfr_init(b);
  mpfr_max(b, MPFR_ONE, N, MPFR_RNDN);

  //Set x = (a + b)/2
  mpfr_init(x);
  mpfr_add(x, a, b, MPFR_RNDN);
  mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);

  //Set f = x^2 - N
  mpfr_init(f);
  mpfr_init(fab);
  mpfr_pow_ui(f, x, n, MPFR_RNDN);
  mpfr_sub(f, f, N, MPFR_RNDN);
  mpfr_abs(fab, f, MPFR_RNDN);

  //Set d = b - a
  mpfr_init(d);
  mpfr_sub(d, b, a, MPFR_RNDN);

  while(mpfr_cmp(fab, T) > 0 && mpfr_cmp(d, T) > 0)
  {
    //Update the bounds, a and b
    if(mpfr_cmp_ui(f, 0) < 0)
      mpfr_set(a, x, MPFR_RNDN);
    else
      mpfr_set(b, x, MPFR_RNDN);

    //Update x
    mpfr_add(x, a, b, MPFR_RNDN);
    mpfr_mul(x, x, MPFR_HALF, MPFR_RNDN);

    //Update f
    mpfr_pow_ui(f, x, n, MPFR_RNDN);
    mpfr_sub(f, f, N, MPFR_RNDN);
    mpfr_abs(fab, f, MPFR_RNDN);
  }

  mpfr_set(R, x, MPFR_RNDN);
}

#ifdef COMPILE_MAIN
int main(int argc, char** argv)
{
  double N, T;
  unsigned int n, D, p;
  mpfr_t Nr, Tr, R;
  int c;
  char sf[50];

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

  switch(argv[1][0])
  {
    case 'a':
      if (argc == 5 &&
          sscanf(argv[2], "%lf", &N) == 1 &&
          sscanf(argv[3], "%lf", &T) == 1 &&
          sscanf(argv[4], "%u", &D) == 1)
        printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, bisect_sqrt(N, T));
      else
        printf("Usage: %s a <N=Value to sqrt> "
               "<T=Tolerance> <D=Number of digits to display>\n",
               argv[0]);
      break;

    case 'b':
      if (argc == 6 &&
          sscanf(argv[2], "%lf", &N) == 1 &&
          sscanf(argv[3], "%lf", &T) == 1 &&
          sscanf(argv[4], "%u", &n) == 1 &&
          sscanf(argv[5], "%u", &D) == 1)
        printf("%u_Root(%.*lf) =~ %.*lf\n",
            n, d(D), N, D, bisect_nRoot(N, T, n));
      else
        printf("Usage: %s b <N=Value to root> <T=Tolerance>"
               "<n=nth Root> <D=Number of digits to display>\n",
               argv[0]);
      break;

    case 'c':
      if (argc == 5 &&
          sscanf(argv[3], "%u", &D) == 1 &&
          sscanf(argv[4], "%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, "sqrt(%%.%uRNf) =~\n\t%%.%uRNf\n", d(D), D);

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

    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;

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

Code for Newton Square Root Method:

File : newton_root.c

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

```

```c
#include "utilities.h"
#include "exact_root.h"
#include "newton_sqrt.h"

#define INIT_CONSTANTS mpfr_init_set_d(MPFR_HALF, 0.5, MPFR_RNDN); \
              in = fopen(ROOT_2_INFILE, "r"); \
              mpfr_init(MPFR_ROOT_2); \
              mpfr_inp_str(MPFR_ROOT_2, in, 10, MPFR_RNDN); \
              fclose(in); \
              in = fopen(ROOT_2_INV_INFILE, "r"); \
              mpfr_init(MPFR_ROOT_2_INV); \
              mpfr_inp_str(MPFR_ROOT_2_INV, in, 10, MPFR_RNDN); \
              fclose(in);

mpfr_t MPFR_ROOT_2, MPFR_ROOT_2_INV, MPFR_HALF;

double newton_sqrt_v1(double N, double T)
{
  assert(N >= 0);
  assert(T >= 0);

  double x, px, d;

  x = N > 1 ? N : 1;
  d = 1000000;

  while(d > T)
  {
    px = x;
    x = 0.5 * (x + N/x);
    d = fabs(x - px);
  }
  return x;
}

double newton_sqrt_v2(double N, double T)
{
  assert(N >= 0);
  assert(T >= 0);

  double x, px, d;

  if(N >= 4)
    x = uint_sqrt((unsigned long) N);
  else
    x = N > 1 ? N : 1;

  d = 1000000;

  while(d > T)
  {
    px = x;
    x = 0.5 * (x + N/x);
    d = fabs(x - px);
  }

  return x;
}
```

```c
double newton_sqrt_v3(double N, double T)
{
  assert(N >= 0);
  assert(T >= 0);

  int e;
  double x, px, d;

  N = frexp(N, &e);

  x = 1;
  d = 1000000;

  while(d > T)
  {
    px = x;
    x = 0.5 * (x + N/x);
    d = fabs(x - px);
  }

  if(e%2)
    x *= e > 0 ? ROOT_2 : ROOT_2_INV;
  return ldexp(x, e / 2);
}

void mpfr_newton_sqrt_v3(mpfr_t R, mpfr_t N, mpfr_t T)
{
  mpfr_t x, px, d, t, n;
  mpfr_exp_t e;

  mpfr_init(n);
  mpfr_frexp(&e, n, N, MPFR_RNDN);

  mpfr_init_set_ui(x, 1, MPFR_RNDN);
  mpfr_init(px);
  mpfr_init_set_ui(d, 1000000, MPFR_RNDN);
  mpfr_init(t);

  while(mpfr_cmp(d, T) > 0)
  {
    mpfr_set(px, x, MPFR_RNDN);
    mpfr_div(t, n, x, MPFR_RNDN);
    mpfr_add(x, x, t, MPFR_RNDN);
    mpfr_mul(x, MPFR_HALF, x, MPFR_RNDN);
    mpfr_sub(d, x, px, MPFR_RNDN);
    mpfr_abs(d, d, MPFR_RNDN);
  }

  if(e%2)
    if(e > 0)
      mpfr_mul(x, MPFR_ROOT_2, x, MPFR_RNDN);
    else
      mpfr_mul(x, MPFR_ROOT_2_INV, x, MPFR_RNDN);
  mpfr_mul_2si(R, x, e/2, MPFR_RNDN);
}

#ifdef COMPILE_MAIN
```

```c
int main(int argc, char **argv)
{
  double N, T;
  unsigned int n, D, p;
  mpfr_t Nr, Tr, R;
  int c;
  char sf[50];
  FILE *in;

  if(argc==1)
  {
    printf("Usage: %s [a/b/c/d] <Arguments>\n", argv[0]);
    exit(1);
  }

  switch(argv[1][0])
  {
    case 'a':
      if (argc == 5 &&
          sscanf(argv[2], "%lf", &N) == 1 &&
          sscanf(argv[3], "%lf", &T) == 1 &&
          sscanf(argv[4], "%u" , &D) == 1)
        printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_sqrt_v1(N, T));
      else
        printf("Usage: %s a <N=Value to sqrt> "
               "<T=Tolerance> <D=Number of digits to display>\n",
               argv[0]);
      break;

    case 'b':
      if (argc == 5 &&
          sscanf(argv[2], "%lf", &N) == 1 &&
          sscanf(argv[3], "%lf", &T) == 1 &&
          sscanf(argv[4], "%u" , &D) == 1)
        printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_sqrt_v2(N, T));
      else
        printf("Usage: %s b <N=Value to sqrt> "
               "<T=Tolerance> <D=Number of digits to display>\n",
               argv[0]);
      break;

    case 'c':
      if (argc == 5 &&
          sscanf(argv[2], "%lf", &N) == 1 &&
          sscanf(argv[3], "%lf", &T) == 1 &&
          sscanf(argv[4], "%u" , &D) == 1)
        printf("sqrt(%.*lf) =~ %.*lf\n", d(D), N, D, newton_sqrt_v3(N, T));
      else
        printf("Usage: %s c <N=Value to sqrt> "
               "<T=Tolerance> <D=Number of digits to display>\n",
               argv[0]);
      break;

    case 'd':
      if (argc == 5 &&
          sscanf(argv[3], "%u", &D) == 1 &&
          sscanf(argv[4], "%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);

            mpfr_digits_to_tolerance(D, Tr);

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

            mpfr_newton_sqrt_v3(R, Nr, Tr);
            mpfr_printf(sf, Nr, R);
          }
          else
            printf("Usage: %s d <N=Value to sqrt> "
                   "<D=Number of digitsto calculate to> "
                   "<p=bits of precision>\n", argv[0]);
        }
        else
          printf("Usage: %s d <N=Value to sqrt> "
                 "<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]);
  }
}
#endif
```