

**Calculator Code:  
Programming Code for Use within a Scientific Calculator**

**Gregory M. Moore  
Fall 2005  
Advisor: Professor Michael Miller**

The calculator is an extension of a mathematician and it has opened up new possibilities within mathematics. It is a machine though, and it is only capable of doing what it is programmed to do. Accordingly, this project aims to develop the internal programmed computational code in the form of a computer program that a scientific calculator could use to compute functions such as square root, the exponential, and sine functions.<sup>1</sup> The idea of this project assumes that the programmer has already developed the very basic addition, subtraction, multiplication, division and integer splicing<sup>2</sup> functions. Then using these basic functions, the program will then compute other more complicated functions found on a typical scientific calculator such as the sine and logarithmic functions.

With the necessity to conform to reality, there are limitations set upon this calculator and they are similar to those found in popular calculators such as the Texas Instruments TI-83 Plus. They are that:

- (1) functions inputs and outputs must be calculated accurately to ten significant figures<sup>3</sup>,
- (2) the absolute values of inputs and outputs, excluding zero, can never be greater than or equal to  $1 \times 10^{100}$  or less than or equal to  $1 \times 10^{-100}$ .

Similar to these mandatory requirements, there are a few goals that I wish to accomplish within the confines of the limitations. They are that:

- (1) the code is as easily programmed as possible,
- (2) and that the program is to be as efficient as possible at computing the functions.

Every programming language has advantages and disadvantages; because I was familiar with the C++ programming language and learning a new language was beyond the scope of this project, I chose to work within the C++ language. A few advantages with choosing this language are that the code is easily transferred into other programming languages and that it is universally known. I choose to use a Windows Console application due to its simplicity and my familiarity with it, but this is a disadvantage because it is not an optimal interface for a calculator due to its lack of visual resemblance of a handheld calculator.

---

<sup>1</sup> Appendix II (Page 17) contains the C++ code used in this program. An executable file of the compiled code, called "Calculator by Gregory Moore," can be found on the Le Moyne College shared drive or at <http://web.lemoyne.edu/~mooregm/>.

<sup>2</sup> Integer splicing is when a real number is "spliced" and only the integer portion of the number is used. An example of this would be 5.24 becoming 5 or -24.5 becoming -24.

<sup>3</sup> This means for example that an input of  $\pi \times 10^{25}$  only necessitates that the calculation be accurate to 3.141592654  $\times 10^{25}$ .

The interface created for user inputs and outputs was created for basic test purposes only, and if I had more time, I would have liked to use Visual Basic to create a visual calculator with buttons that were able to be pressed. Even though the created interface looks rudimentary, it contains all the possible tasks that the program is able to do. There were other programming and computations issues that needed to be taken into account as well when transferring mathematical theory into a working calculator. One issue was the large difference in computational time. Computing multiplication and division takes significantly longer time than computing addition and subtraction, and therefore should be avoided whenever possible. Thus throughout the program there are instances where addition and subtraction is used to eliminate the need for multiplication. Another issue is that the program is constantly rounding numbers and thus losing trailing digits. Therefore in some cases it is very difficult or impossible to adhere to the goals previously described. An example of this is within the modulo function which cannot be evaluated for relatively large values modulo a relatively small value.

### **Methods of Approximation**

It is unnecessary to get exact values for the results of many of the functions because they are either irrational (and thus it's impossible) or they contain more digits than necessary to obtain the required 10 significant figures accuracy. Therefore, instead of attempting to get the exact value of a function, the goal of this calculator is to get 10 significant figure approximations of a function for a given input as stated previously. To approximate the functions, while conforming to the limitations of addition, subtraction, multiplication, and division, there are many different possible methods.

There are a few methods used frequently throughout the program. The first method is to take advantage of identities wherever possible such as  $\sqrt{xy} = \sqrt{x}\sqrt{y}$ . There are typically a few identities used in every approximation. Next is to use encode exact answers into the code wherever it is felt appropriate. An example of this is "hard coding" the natural log of one to be zero. Thirdly is to use a polynomial equivalent to:

$$a_0 + a_1x + a_2x^2 + a_3x^3 \dots \quad (i)$$

to estimate intervals of a function.

To obtain the variable values in expression (i) the first method that was attempted is called the discrete least squares method. If “Approx(x)” is the approximating polynomial and “Function(x)” is the function being estimated, this method minimizes:

$$\sum_k (Approx(k) - Function(k))^2 .$$

for a discrete set of chosen points, k. In minimizing this, the a<sub>k</sub> terms of (i) are then found and used to generate a best-fit polynomial of a degree preset by the user. While this method works, for the purposes of this project, it is not ideal because it does not calculate for an entire interval, rather only finite number of points. Therefore it was disregarded in search of a method that will calculate along intervals and for a specific set of values.

The next method found was the continuous least squares method and it is very similar to the discrete least squares method, but it calculates along intervals. This method minimizes the integral:

$$\int_{LowerBound}^{UpperBound} (Approx(x) - Function(x))^2 dx ,$$

to determine the best approximation polynomial for an interval of a predetermined user defined degree. This method is far better than the discrete least squares method at approximating intervals, but there exists an adaptation of this idea that is even more accurate for specifically estimating polynomials such as those used in this program called the Chebyshev least squares approximation method.

### **Chebyshev’s Polynomials<sup>4</sup>**

The Chebyshev least square approximation method is specifically designed for approximating functions with polynomials. Through its use of interpolation with each successive increase in degree of the polynomial, the approximation polynomial converges onto the desired function more efficiently than the standard least squares method. This is unique because each successive term in the Chebyshev polynomial adjusts the previous ones to make the entire polynomial more accurate.

This estimation method is based on Chebyshev’s polynomials, which are defined by:

---

<sup>4</sup> All information referring to Chebyshev’s Polynomials was taken from Numerical Analysis: Theory and Practice by L.N. Asaithambi, Saunders College Publishing (Orlando, Florida: 1995) Pages 422-436.

$$T_k(x) = \cos(k \cos^{-1}(x)), \quad k \geq 0.$$

Concerning this application of Chebyshev's polynomials, we can define  $T_k(x)$  by:

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x), \quad k \geq 2.$$

To find the estimating polynomial the Maple program<sup>5</sup> that was created uses a version of the continuous least squares method of integration. Given a function to estimate with the polynomials,  $Function(x)$ , this Maple program finds an intermediate term,  $t_i$ , which will eventually be used to find the final estimating polynomial through the integrals:

$$t_1 = \frac{1}{\pi} \int_{-1}^1 \frac{Function(x)}{\sqrt{1-x^2}} dx$$

$$t_k = \frac{2}{\pi} \int_{-1}^1 \frac{Function(x) * T_k(x)}{\sqrt{1-x^2}} dx, \quad k \geq 2.$$

These integrals are based upon the least squares method of approximation with a weight function of:

$$\frac{1}{\sqrt{1-x^2}}.$$

Once this preliminary computation is completed, a summation is used to find the final estimating polynomial. This summation is:

$$f(x) = \sum_{k=0}^{\text{Degree Of Function}} t_k T_k(x).$$

This method only computes an estimate for a function on the interval  $[-1, 1]$ . To remedy this in order to estimate functions not on this preset interval, the function, which is estimated on a user-defined interval, is translated into the bounds of  $[-1, 1]$  by the translation:

$$Function(x), \quad x \in [A, B] \rightarrow Function\left(\left(x + \frac{2 * B}{B - A}\right) * \left(\frac{B - A}{2}\right)\right), \quad x \in [-1, 1].$$

To convert this back to the required final estimation polynomial,  $f(x)$ , the summation:

$$f(x) = \sum_{k=0}^{\text{Degree Of Function}} t_k T_k(x)$$

---

<sup>5</sup> See appendix I (Page 15) for Maple program code.

is changed to:

$$f(x) = \sum_{k=0}^{\text{Degree Of Function}} t_k T_k \left( \frac{x * 2}{B - A} - \left( \frac{2 * B}{B - A} \right) \right)$$

An advantage that the Chebyshev method has is that each successive  $t_i$ , that create the polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n ,$$

is independent of the previous or later  $t_i$ 's. This allows the computation of specific parts of the polynomial at different times that was useful for computing an approximation polynomial of the arc sine function. This is because the computation of the arc sine function took excessive hours of computational time on Maple and, for estimating higher degree polynomials, used up the 1056 MB of RAM on my computer.

Once a polynomial has been determined using Chebyshev's method above, it is then copied into the code to be computed once it is given a value of "x" by the user. A trick used in the code called Horner's Method of Computing Polynomials is then used to significantly reduce the number of computations needed by the computer to calculate a polynomial. The idea is simple; it is only a rearrangement of the polynomial. Given a polynomial the form

$$a_0 + a_1x + a_2x^2 + a_3x^3 ,$$

the number of multiplications necessary to compute this is six. The number of multiplications required to compute this polynomial is based on the idea that given a value of x, a computer will then multiply  $a_1$  with x,  $a_2$  times x times x, and  $a_3$  times x times x times x for a total of six multiplications and then add these sums together with  $a_0$  to obtain the result. If a factor of x is taken out of the terms containing an x, the polynomial becomes

$$a_0 + (a_1 + a_2x + a_3x^2)x ,$$

which, under the same method of counting multiplications, only takes four multiplications to compute. Factoring again gives

$$a_0 + (a_1 + (a_2 + a_3x)x)x ,$$

and takes only three multiplications to compute. Using this method, any given polynomial can be computed using only the degree of the function multiplications and at most the degree of the function additions. Appendix I contains the Maple program, based on the Chebyshev polynomials, estimating the square root function.

## The Square Root Function (Sqrt(x))

The square root function is a necessary part of this calculator because it will be used in the estimation of the Arc Sine function, among others described later, as well as it is a commonly used function that is useful operation on a calculator. The calculation of this function involves a few different identities, some hard coded values, and then an estimation polynomial. For the purposes of this calculator the square root function's domain is  $\{0\} \cup (10^{-100}, 10^{100})$ .

For the values of x equal to zero and one, it is hard coded into the program that the square root of x is equal to zero and one respectively. For values of x in the interval (0, 1), the identity  $\sqrt{x} = 1 / \sqrt{1/x}$  is used to convert the values that will be computed into the set (1,  $10^{100}$ ) because of the reciprocal of a number in the interval ( $10^{-100}$ , 1) is in the interval (1,  $10^{100}$ ).

Therefore it is only necessary to estimate the function for values of x contained in (1,  $10^{100}$ ). Estimating the interval of the square root function from 1 to  $10^{100}$  requires a very high degree polynomial to obtain the required ten significant figures accuracy. This high degree polynomial would require many computations of multiplication and be very time consuming for the processor to compute. Therefore to avoid this and make the process for efficient to compute, first the estimation will make use the identity

$$\sqrt{xy} = \sqrt{x}\sqrt{y}.$$

Using this, it is possible to break down large numbers into smaller ones that make the estimation easier to compute because it makes the required estimation interval smaller. Given any value of x in [ $10^{50}$ ,  $10^{100}$ ), the square root can be computed by the formula:

$$\sqrt{x} = \sqrt{10^{50}} \sqrt{x/10^{50}} = 10^{25} \sqrt{x/10^{50}}.$$

Thus, the square root of x in the interval [ $10^{50}$ ,  $10^{100}$ ) would now be  $10^{25}$  (a constant) times the square root of  $x / 10^{50}$ , and  $x / 10^{50}$  is contained in the interval [1,  $10^{50}$ ). Continuing this method of breaking down values of x, we can make the polynomial estimation interval of the square root function relatively small. Continuing along this principle of breaking down the interval, the next few intervals would be

$$[1, 10^{25}), [1, 10^{12.5}), [1, 10^{6.25}), [1, 10^{3.125}), \dots$$

This can continue into infinitely small intervals, but at a cost of one division and one multiplication per break down. Therefore it becomes necessary to determine when to stop this

process. For this function it was determined that the ideal time to stop was for values of x in the interval (1, 1.0578255745].<sup>6</sup> Therefore the interval of values (1,1.0578255745] is the only interval of the square root function that has yet to be estimated. To estimate this interval, a fourth degree polynomial is used to obtain the required accuracy, and with this, the entire square root function can be computed.

The polynomial used for the square root function on the interval (1, 1.0578255745 ] is:

$$(.27731022259508+(1.0784488163225+(-.52420398582544+ (.20382065911444-0.35375712175953e-1*x)*x)*x)*x).^7$$

### The Power Function (X^Y)

The power function is an easy function to compute using an identity based upon the logarithmic and exponential functions that will be discussed later. The basic idea that it uses is the identity

$$x^y = \exp(y * \ln(x)).$$

The only issue that needs to be dealt with is that this identity is only defined for positive values of x, but the power function is also defined for x values of  $\{0\} \cup (-10^{100}, -10^{-100})$ . Therefore to deal with the first part of this set, it is hard coded that if x = 0, then the result equals 0 as long as y does not equal zero as well. If x = y = 0, then an error is reported.

If x is in the interval  $(-10^{100}, -10^{-100})$  then x is negated and then the equation needs to be adjusted. To adjust it we must deal with the problem that only integer values of y return a real valued solution if x is in the interval  $(-10^{100}, -10^{-100})$ . Therefore, the function will first abort if there would be a complex result, and then it will determine if the answer should be positive or negative. To do this, it uses the Modulo Estimator Function that will be discussed shortly, and determines if the y value is an odd number, and if is, it will negate the result.

Therefore the Power Function is based almost entirely on other functions and is easy to code. This does have its downfalls though, first it that it relies heavily on the natural logarithmic function and the exponential function. Thus it can only be as efficient and accurate as these functions are. This is the reason why the square root function was developed separately. While

---

<sup>6</sup> This is based on the idea that with each successive break down, it required an additional division, but the required polynomial approximation for the given interval was not losing any degree.

<sup>7</sup> Appendix I (Page 15) contains the Maple program that used the Chebyshev method to compute this polynomial estimation.

the square root function takes time to develop and space in the memory, it is, on average, significantly more efficient than the powers function at computing the square root of a number, and because it is commonly used and a necessary computation in other functions, it was determined that it was worth the time and memory space.

### Trigonometry Functions - Modulo

The first function that needs to be addressed in order to be able to compute trigonometric functions is the modulo function. It is a very important function because used in the sine, cosine, tangent functions and others throughout the program. The first thing that needs to be dealt with in the modulo function as developed here is non-positive values of y, if we write the function  $x \pmod{y}$ . In this program, this is not valid, and thus will return an error.

Following this detail, the program uses the idea that if x is positive or zero

$$x \pmod{y} = \left( \text{remainder of } \left( \frac{x}{y} \right) \right) * y$$

and if x is negative

$$x \pmod{y} = \left( \text{remainder of } \left( \frac{x}{y} \right) + 1 \right) * y .$$

Therefore all that is left to do it determine the remainder of  $\left( \frac{x}{y} \right)$ . To do this the program

subtracts  $\left( \frac{x}{y} \right)$  by the integer portion of  $\left( \frac{x}{y} \right)$ .<sup>8</sup> Therefore if this result is greater than or equal to zero, it only needs to be multiplied by y to find the final result to return, if it is negative, then one is added to it and then multiplied by y to find the final result to return.

There are limitations on this due to the necessity of having remainders. For large values of x, and small values of  $y > 0$ , the computer is unable to store enough significant figures to compute the remainder accurately.

### Sine Function (Sin(x))

---

<sup>8</sup> This is not to be confused with the greatest integer less than x function, [x], if x = -4.5, it will splice to -4, not -5.

The sine function's symmetry makes this function easy to compute. The first thing that is done is that  $x$  is converted by the equation  $x = x \pmod{2\pi}$ . This immediately changes the interval that needs to be estimated into the interval  $[0, 2\pi)$ . To this symmetry is exploited even further using the identities:

- 1.) if  $x$  is in the interval  $(\pi/2, \pi]$ , then  $\sin(x) = \sin(\pi - x)$ ,
- 2.) if  $x$  is in the interval  $(\pi, 3\pi/2]$ , then  $\sin(x) = -\sin(x - \pi)$ , and
- 3.) if  $x$  is in the interval  $(3\pi/2, 2\pi)$ , then  $\sin(x) = -\sin(2\pi - x)$ .

Therefore it is only necessary to compute sine on the interval  $[0, \pi/2]$ . To estimate this interval, polynomial approximations of sine will be used. This interval is then broken up into three smaller intervals for accuracy reasons, and then three polynomials are used to compute sine.

The only non-basic exterior function that sine calls upon is the modulo function which is unable, due to round off error, to compute large numbers modulo  $2\pi$ . Accordingly sine adheres to the limits of the modulo function and for this reason, the sine function cannot be computed accurately or at all if the absolute value of the computed number is large.<sup>9</sup>

### **Cosine and Tangent Functions (Cos(x) and Tan(x))**

The cosine and tangent functions are largely based on the sine function. For cosine, the computation is easy,  $\cos(x) = \sin(x - \pi/2)$ , and thus this is exactly what the program computes.

The tangent function uses the identity  $\tan(x) = \frac{\sin(x)}{\cos(x)}$  for computation. With tangent though,

there is one trick used for efficiency, in an effort to reduce redundancy and unnecessary computations, before the sine and cosine functions are computed,  $x$  is standardized into the interval  $[0, 2\pi)$ . Furthermore, after standardization, if  $x$  is less than  $\pi/2$ , then the program adds  $2\pi$  to the value in the cosine function to prevent an unintended second standardization. This eliminates the need to ever standardize  $x$  twice. With this cosine and tangent are complete and as accurate as the sine function.

### **Arc Sine (ArcSin(x))**

The arc sine function is only real valued for the interval  $[-1, 1]$ . To exploit symmetry for the interval  $(-1, 0)$ , the identity  $\arcsin(x) = -\arcsin(-x)$  can be used to rely on the interval of  $(0, 1)$ .

---

<sup>9</sup> Larger than approximately  $10^{10}$ . This is the same restriction as standard handheld calculator.

The values  $\{-1, 0, 1\}$  are all hard coded into the program, and thus it only remains to compute the interval  $(0, 1)$ . To do this, the interval is reduced further by the identity

$$(\arcsin(x) = -\arcsin(\sqrt{1-x^2}) + \pi/2).^{10}$$

Using this, the estimation interval  $(0, 1)$  can be further reduced to the estimation interval  $(0, \frac{\sqrt{2}}{2})$ . This interval is then broken down into three different intervals for accuracy reasons, and estimated using polynomial approximations.

### **Arc Cosine (ArcCos(x))**

The arc cosine function is real valued for the interval  $[-1, 1]$ . The values  $\{-1, 1\}$  are hard coded into the program for efficiency, and using the identity

$$\arccos(x) = \arcsin(-x) + \pi/2$$

the remaining interval is computed.

### **Arc Tangent (ArcTan(x))**

Arc tangent is computed using an identity. Before the identity can be used efficiently, input values less than  $-10^{10}$  return is  $-\pi/2$  and similarly if the input value is greater than  $10^{10}$  then the returned value is  $\pi/2$ . Following this, the identity

$$\arctan(x) = \arcsin\left(\frac{x}{\sqrt{x^2 + 1}}\right)$$

is used to determine the arc tangent function on the interval  $(-10^{10}, 10^{10})$ .

### **Exponential Function – (Exp(x))**

The exponential function is only defined, based on the constraints of this calculator, for values of  $x$  in the set  $\{0\} \cup (10^{-100}, 230.25850929940) \cup (-230.25850929940, -10^{-100})$ . If  $x$  is equal to zero, it is hard coded that the result is one. If  $x$  is in the interval less than zero, the identity:

$$e^x = \frac{1}{e^{-x}}$$

---

<sup>10</sup> Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables edited by Milton Abramowitz and Irene A. Stegun, United States Department of Commerce. (Washington D.C.: 1970) Page 82

is used to leave the only interval of estimation necessary to compute being  $(10^{-100}, 230.25850929940)$ . Using the idea that

$$e^{x+y} = e^x e^y,$$

this interval can be broken down even further in a similar manner as the square root function. It was determined that the optimal time to stop this breaking down process is for values of  $x$  is in the interval  $(0, 0.125)$ . To calculate this smaller interval, an estimating polynomial is used of degree 4.

### **Logarithmic Functions – (Ln(x), Log(x) [base 10])**

The Log(x) [base 10] is based upon the natural log function. If  $x$  is in the interval  $(10^{-100}, 10^{100})$  then the equation:

$$\text{Log}(x) [\text{Base } 10] = \text{Ln}(x) / 2.3025850929940$$

is used. Therefore it is only necessary to calculate the natural log function.

The natural log function is defined for  $(10^{-100}, 10^{100})$ . Hard coding that  $\ln(1) = 0$  and using the identity:

$$\ln(x) = -\ln\left(\frac{1}{x}\right)$$

the estimation interval can be reduced to the interval  $(1, 10^{100})$ . To estimate this interval the identity

$$\ln(x * y) = \ln(x) + \ln(y)$$

is used to break down this interval in a similar method as the square root function. From this it is determined that it is ideal to stop breaking the intervals down is for values of  $x$  in the interval  $(1, 1.0578255745)$ . To estimate values of  $x$  is in this interval, approximating polynomials are used to obtain an estimation of the function. Therefore, using identities, and then breaking down the function, it is possible to compute this function on the entire interval.

### **Testing Functions**

Once the functions were created, they needed to be tested for accuracy and efficiency. As a basis to work with, the native C++ functions are used for comparison purposes. To make

this work well, there needs to be a large amount of numbers available to test for accuracy and to get efficiency data.

A random number generator was created that, based upon a user's inputs, creates a specified number of random numbers on a given range. To do this, a vector is created with random integral values between in the interval  $(1, 2^{15})$ . These values are then divided by  $2^{15}$  and thus randomly distributed in the in the interval  $(0, 1)$ . Following this, the numbers, under the same randomization pattern, are spread throughout the user defined interval.

The error estimation function's purpose is to determine how accurate a given function is based upon the respective C++ built in function. To do this, a vector is created containing a user's defined number of terms, upper bound, lower bound and a function. Given this, the error estimator function calculates the percent error of each value in the vector by the equation:

$$\text{Percent Error}(x) = \frac{(\text{Created\_Function})(x) - (\text{C++Function})(x)}{(\text{C++Function})(x)}$$

After each value is tested, the percent error of the function at that value is tested to determine if the absolute value of the percent error at that point is the maximum percent error of the values tested thus far. Once all values in the vector have been tested, the maximum percent error is outputted and if the result is less than or equal to  $10^{-10}$  then it is determined that the accuracy is sufficient and complies with the ten significant figure goal.

In theory and through all my testing I have found that the calculations that are outputted are accurate to the ten significant figure requirement. The times that the program would not compute to the required accuracy, similar to a calculator, an error should be outputted. An example of this is  $\sin((1e50) + 1)$ . This in theory is calculable, but the large number of significant figures prevents this for happening realistically. In this program there is an error that is triggered and thus the program is stopped.

The efficiency estimator calculates how efficient a created function is compared to respective C++ native function. To do this, a vector of random numbers is created based upon the user defined variables of the number of terms, upper bound, lower bound and a function. Once this is done, the created function is timed to see how many ticks<sup>11</sup> it takes to compute all of the values in the vector. Following this, a C++ native function is tested to see how long, in ticks, it takes to compute all the values in the vector.

---

<sup>11</sup> 1 tick = 1/1000 second.

Once the timing is complete, the time of the created function is divided by the time of the C++ native function to determine an estimate of how much slower one was compared to the other. Note though, that this number is not an exact estimate of difference in efficiencies because the method for calling the functions distorts the results. The intermediate step of calling the proper function (“CallFunction”) utilizes time, but this is identical for both the C++ native function and the created function, so it only skews the data. An example of this occurrence is the sine function. It is approximately 1.7 times slower than the C++ built in function, but with this method of calling reports differences of times in the 1.25 – 1.35 multiples range.

Based on the outputted efficiency results, which again are skewed, the results are very good. The square root function, powers function, arc sine function and arc cosine function are very good consistently returning times in the 1.01 to 1.1 range compared to the C++ respective function. The arc tangent function and the exponential function are also very good returning times in the 1.1 to 1.2 range compared to their respective C++ function. The natural logarithmic and logarithmic function (base 10) returned times in the 1.18 to 1.28 range. Finally, the sine, cosine, and tangent functions returned times in the 1.275 to 1.45 range. Overall then, the program is relatively efficient and could be used in a basic handheld calculator without a noticeable difference to the user.

### **If I had more time...**

There are some issues that I am aware of that have not been dealt with or that can be resolved if there was more time. There are also issues that I have not found out and may not ever be discovered. One of the larger issues that I am aware of is within the modulo function. It is unable compute accurately for large values of x and relatively small values of y. This is due to the round-off error involved when dividing large numbers. This is important because the function is unable to compute sine or the power function<sup>12</sup> accurately. While this problem also is found in popular calculators, and within this program’s requirements is almost certainly a lost cause, it would be beneficial to have.

Another issue is throughout the entire program concerns round off error, but I have been unable to figure out the reasoning for it. There are many round off errors happening where I did

---

<sup>12</sup> The powers function is only effected if x is a large negative value because the modulo function is used to determine if it is an odd or even value, and thus return the appropriate sign.

not expect them to and thus making functions that in theory work, but in practice do not obtain the required accuracy. An example of these unforeseen errors was within a previous random number generator. Because of these errors I was forced to change the random number generator, for it was unable to give results less than  $10^{-10}$  for some unknown reason, rather automatically round them off to “1.#INF”.

As mentioned before, I would have also liked to use a different interface for the calculator. A visual interface would be much more natural for user inputs and outputs, as well as give more satisfaction in creating a “physical” traditional calculator. Finally, I would like to research and determine other ways for computing functions more efficiently.

## Appendix I: Maple program to compute estimation polynomials via Chebyshev's Method.

```
restart:
Digits:= 25:
FunctionToEstimate:= x-> sqrt(x): FunctionToEstimate(x);
LowerBound:=evalf( 1);
UpperBound:=evalf( 1.0578255745);
LowerDegreeOfFunction:= 0;
UpperDegreeOfFunction:= 4;

DifferenceOfBounds:= UpperBound - LowerBound:
NewUpperBound:= UpperBound*2/DifferenceOfBounds:
Slide:= NewUpperBound - 1:
Function:= x->
FunctionToEstimate((x+Slide)*DifferenceOfBounds/2):
inverse:= (x*2/DifferenceOfBounds)-Slide:

T:= proc(k, x);
    if (k = 0) then return 1 end if;
    if (k = 1) then return x end if;
    if (k >=2) then return simplify(2*x*T(k-1, x)-T(k-2,
x))end if;
end:

t[0]:= evalf(1/Pi* int(Function(x)/sqrt(1-x^2), x= -1..1));
for k from LowerDegreeOfFunction + 1 to UpperDegreeOfFunction do
t[k]:= evalf(2/Pi * int(Function(x)*T(k, x)/sqrt(1-x^2), x= -
1..1)) end do;
Digits:=14:
F:= simplify(add(t[i]*T(i, inverse),
i=LowerDegreeOfFunction..UpperDegreeOfFunction)):
plot({((F - FunctionToEstimate(x)) /
FunctionToEstimate(x))*1000000000}, x=LowerBound..UpperBound);
convert(F, horner, x);
```

$$\sqrt{x}$$

$$\text{LowerBound} := 1.$$

$$\text{UpperBound} := 1.0578255745$$

$$\text{LowerDegreeOfFunction} := 0$$

$$\text{UpperDegreeOfFunction} := 4$$

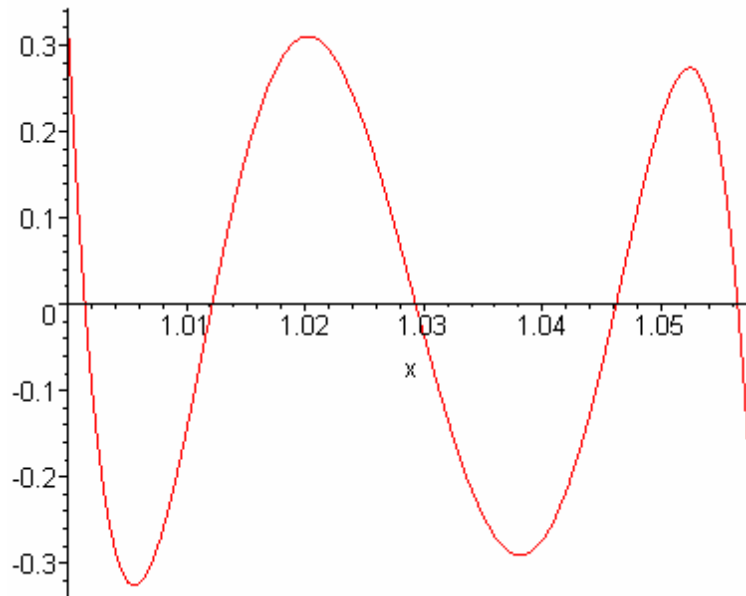
$$t_0 := 1.01430331440014$$

$$t_1 := .0142528869495101$$

$$t_2 := -.0000500725012392203$$

$$t_3 := .35182861748069410^{-6}$$

$$t_4 := -.30876058959827710^{-8}$$



The graph of the curve is less than 1, therefore the required accuracy was achieved.

$$.27734241640171 + (1.0783235997544 + (-.52402137550741 + (.20370231433693 - .035346954952493 x) x) x) x$$

## Appendix II: C++ Code used to create the calculator.

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <time.h>
#include <vector>
#include <string>
#include <iomanip>
#include <windows.h>
using namespace std;
const double HALFPi = 1.5707963267949;
const double Pi = 3.14159265358979;
const double THREEHALFPi = 4.7123889803847;
const double TWOPi = 6.2831853071796;
struct ReturnError{int i; ReturnError(int ii) {i=ii;}};
void Error();
double Exp(double x);
double Ln(double x);
double ArcTan(double x);
double CallFunction(int Choice, double x, double y);
double CallCplusplusFunction(int choice, double x, double y);
int Prompt(int Reason);

double Addition(double x, double y) {return x + y;}
double Subtraction(double x, double y) {return x - y;}
double Multiplication(double x, double y) {return x * y;}
double Division(double x, double y) {return x / y;}
double ModEstimator(double x, double y) // x mod(y) && y must be positive.
{
    if (y <= 0) Error();
    double estimator = x / y;
    estimator = estimator - (int)estimator;
    if (estimator < 0) return (estimator*y + y) ;
    return estimator * y;
}
double Mod()
{
    cout << "Given: x (mod y)" << "\nPlease Enter 'x': ";
    double x;
    cin >> x;
    if (cin.fail()) Error();
    cout << "Please Enter 'y': ";
    double y;
    cin >> y;
    if (cin.fail()) Error();
    return ModEstimator(x, y);
    Error();
}
double Sqrt(double x) // Note: Sqrt(x*y) = Sqrt(x)*Sqrt(y);
{
    if (x < 0) Error();
    double Answer = 1;
    double Adjuster = 1;
    if(x > 0 && x < 1)
    {
```

```

Adjuster = x;
x = 1 / x;
}
if (x < 1e100 && x > 1)
{
    if ( x > 1e50)
    {
        Answer = 1e25;
        x = x / 1e50;
    }
    if ( x > 1e25)
    {
        Answer = Answer * 3.162277660e12;
        x = x / 1e25;
    }
    if ( x > 3.162277660e12)
    {
        Answer = Answer * 1.778279410e6;
        x = x / 3.162277660e12;
    }
    if ( x > 1.778279410e6)
    {
        Answer = Answer * 1333.521432;
        x = x / 1.778279410e6;
    }
    if ( x > 1333.521432)
    {
        Answer = Answer * 36.51741272;
        x = x / 1333.521432;
    }
    if ( x > 36.51741272)
    {
        Answer = Answer * 6.042963902;
        x = x / 36.51741272;
    }
    if ( x > 6.042963902)
    {
        Answer = Answer * 2.458244069;
        x = x / 6.042963902;
    }
    if ( x > 2.458244069)
    {
        Answer = Answer * 1.5678788438;
        x = x / 2.458244069;
    }
    if ( x > 1.5678788438)
    {
        Answer = Answer * 1.2521496891;
        x = x / 1.5678788438;
    }
    if ( x > 1.2521496891)
    {
        Answer = Answer * 1.1189949460;
        x = x / 1.2521496891;
    }
    if ( x > 1.1189949460)
    {

```

```

        Answer = Answer * 1.0578255745;
        x = x / 1.1189949460;
    }
    if ( x > 1.0578255745)
    {
        Answer = Answer * 1.0285064775;
        x = x / 1.0578255745;
    }
    if (x > 1)
    {
        Answer = Answer * (.27731022259508+(1.0784488163225+(-
.52420398582544+ (.20382065911444-0.35375712175953e-1*x)*x)*x)*x);
    }
    return Answer * Adjuster;
}
if (x == 0) return 0;
if (x == 1) return 1;
Error();
}
double Power(double x, double y)
{
    double Adjuster = 1;
    if (x < 0)
    {
        if (y - (int)y != 0) Error(); //nth root of negative #, where n
is a not an integer.
        if (ModEstimator(y, 2) == 1)
            Adjuster = Adjuster * -1;
        x = -1 * x;
    }
    if (x == 0)
    {
        if (y == 0) Error();
        return 0;
    }
    return Adjuster * Exp(y*Ln(x));
}
double SinEstimator(double x) // x - > [0, Pi/2]
{
    if (x > 0.1 && x < HALFPi)
        return -0.846700000000000e-10+(1.0000000023612+(-
0.266455200000000e-7+(-.16666650472215+(-0.59590816893210e-
6+(0.83347449028899e-2+(-0.22231169620330e-5+(-0.19605809547659e-3+(-
0.16622686387982e-5+(0.35102425487219e-5-0.20185741637521e-
6*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x;
    if (x <= 0.1 && x > .0003107341499)
        return -0.100000000000000e-14+(1.0000000000024+(-
0.38089751810000e-9+(-.16666664385070+(-0.65132477470000e-
6+(0.83428817050840e-2-0.69430869016437e-4*x)*x)*x)*x)*x)*x;
    if (x >= 0 && x < .0003107341499)
        return x;
}
double Sin(double x)
{
    if (x < 0 || x >= TWOPi)
        x = ModEstimator(x, TWOPi);
    if(x >= 0 && x < HALFPi)

```

```

        return SinEstimator(x);
    if(x >= HALFPI && x < Pi)
        return SinEstimator(Pi - x);
    if(x >= Pi && x < THREEHALFSPI)
        return -SinEstimator(x - Pi);
    if(x >= THREEHALFSPI && x < TWOPI)
        return -SinEstimator(TWOPI - x);
    Error();
}
double Cos(double x)
{
    return Sin(x - HALFPI);
}

double Tan(double x)
{
    x = ModEstimator(x, TWOPI);
    if (x < HALFPI) return Sin(x) / Cos(x + TWOPI);
    //Eliminates the need for the modulo function to be computed
    within cosine.
    if (x >= HALFPI) return Sin(x) / Cos(x);
}
double ArcSinEstimator(double x) // x -> [0, 1]
{
    if (x <= .70710678118655 && x >= 0.15) // .70710678118655 = (.5 *
sqrt(2))
        return 0.44048609600000e-4+(.99807907244156+(0.38012728398690e-
1+(-.28590881250036+(3.6236119542291+(-20.578529307156+(86.489755056376+(-
270.53863165212+(636.17866292572+(-1119.9093129553+(1455.0351791982+(-
1354.8081850293+(856.12545216732+(-
329.29871045847+58.355732347635*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x)*x;
        // 9 Sig. Figures accuracy.
    if (x < 1 && x > .70710678118655) // .70710678118655 = (.5 * sqrt(2))
        return -ArcSinEstimator(Sqrt(1-(x*x))) + HALFPI;
    if (x <.15 && x > 0.00003107341499)
        return 0.40000000000000e-14+(1.00000000000003+(-
0.62786854300000e-10+(.16666667191523+(-0.22181691775660e-
6+(0.75005342290432e-1+(-0.77277293773000e-4+(0.45317459415328e-1+(-
0.33655927030790e-2+0.38189409351506e-1*x)*x)*x)*x)*x)*x)*x)*x)*x;
    if (x >= 0 && x <= 0.00003107341499)
        return x;
    Error();
}
double ArcSin(double x) // x -> [-1, 1]
{
    if (x < 1 && x > 0) return ArcSinEstimator(x);
    if (x < 0 && x > -1) return -ArcSinEstimator(-x);
    if (x == 1) return HALFPI;
    if (x == 0) return 0;
    if (x == -1) return -HALFPI;
    Error();
}
double ArcCos(double x) // x -> [-1, 1]
{
    if (x < 1 && x > -1) return ArcSin(-x)+HALFPI;
    if (x == 1) return 0;
    if (x == -1) return Pi;
}

```

```

    Error();
}
double ArcTan(double x)
{
    if(x < 1e100 && x > 1e10) return HALFPI;
    if(x < -1e10 && x > -1e100) return -HALFPI;
    if(x >= -1e10 && x <= 1e10) return ArcSin(x/sqrt(x*x+1));
    Error();
}
double ExpEstimator(double x) // x -> [0, 0.125]
{
    if (x > 0 && x <= 0.125)
        return
1.00000000005260+(.99999979004637+(.50001339110801+(.16636841193070+.044362599
37134*x)*x)*x)*x;
    if (x == 0)
        return 1;
    Error();
}
double Exp(double x)
{
    if (x > 230.25850929940 || x < -230.25850929940) Error();
    bool LessThanZero = false;
    if (x < 0)
    {
        x = -x;
        LessThanZero = true;
    }
    double result = 1;
    if (x > 0)
    {
        if (x >= 128)
        {
            result = .38877084059946e56;
            x = x - 128;
        }
        if (x >= 64)
        {
            result = result * .62351490808116e28;
            x = x - 64;
        }
        if (x >= 32)
        {
            result = result * 78962960182681;
            x = x - 32;
        }
        if (x >= 16)
        {
            result = result * 8886110.5205079;
            x = x - 16;
        }
        if (x >= 8)
        {
            result = result * 2980.9579870417;
            x = x - 8;
        }
        if (x >= 4)

```

```

    {
        result = result * 54.598150033144;
        x = x - 4;
    }
    if (x >= 2)
    {
        result = result * 7.3890560989307;
        x = x - 2;
    }
    if (x >= 1)
    {
        result = result * 2.7182818284590;
        x = x - 1;
    }
    if (x >= 0.5)
    {
        result = result * 1.64872127070013;
        x = x - 0.5;
    }
    if (x >= 0.25)
    {
        result = result * 1.28402541668774;
        x = x - 0.25;
    }
    if (x >= 0.125)
    {
        result = result * 1.13314845306683;
        x = x - 0.125;
    }
    if (x > 0) result = result * ExpEstimator(x);
    if (LessThanZero == false)
        return result;
    else if (LessThanZero == true)
        return 1 / result;
}
else if (x == 0) return 1;
Error();
}
double Ln(double x) // Base e.
{
    double Answer = 0;
    int Adjuster = 1;
    if(x > 0 && x < 1)
    {
        x = 1 / x;
        Adjuster = -1;
    }
    if (x < 1e100 && x > 1)
    {
        if (x >= 1e50)
        {
            Answer = 115.12925465;
            x = x / 1e50;
        }
        if (x >= 1e25)
        {
            Answer = Answer + 57.56462732;

```

```

        x = x / 1e25;
    }
    if (x >= 3.162277660e12)
    {
        Answer = Answer + 28.782313662;
        x = x / 3.162277660e12;
    }
    if (x >= 1.778279410e6)
    {
        Answer = Answer + 14.391156831;
        x = x / 1.778279410e6;
    }
    if (x >= 1333.521432)
    {
        Answer = Answer + 7.195578415;
        x = x / 1333.521432;
    }
    if (x >= 36.51741272)
    {
        Answer = Answer + 3.597789208;
        x = x / 36.51741272;
    }
    if (x >= 6.042963902)
    {
        Answer = Answer + 1.798894604;
        x = x / 6.042963902;
    }
    if (x >= 2.458244069)
    {
        Answer = Answer + 0.8994473020;
        x = x / 2.458244069;
    }
    if (x >= 1.5678788438)
    {
        Answer = Answer + .44972365096;
        x = x / 1.5678788438;
    }
    if (x >= 1.2521496891)
    {
        Answer = Answer + .22486182552;
        x = x / 1.2521496891;
    }
    if (x >= 1.1189949460)
    {
        Answer = Answer + .11243091279;
        x = x / 1.1189949460;
    }
    if (x >= 1.0578255745)
    {
        Answer = Answer + 0.56215456423e-1;
        x = x / 1.0578255745;
    }
    if (x > 1.001)
    {
        Answer = Answer + (-2.4217836535974+(5.8328745435397+(-
7.0875349387807+(6.1237356880758+(-3.3479880477301+(1.0412379902940-
.14054158180134*x)*x)*x)*x)*x)*x);
    }

```

```

    }
    else if (x > 1.000000005)
    {
        Answer = Answer + (-1.4995002493334+(1.9990006859698-
.49950043662595*x)*x); //Need better estimator for this!
    }
    else if (x > 1.000000001)
    {
        Answer = Answer + x - 1;
    }
    return Answer * Adjuster;
}
if(x == 1) return 0;
Error();
}
double Log(double x) // Base 10.
{
    if (x > 0 && x < 1e100)
        return Ln(x) / 2.3025850929940;
    Error();
}
double GetX()
{
    cout << "Please enter a number:\n";
    double x;
    cin >> x;
    if (cin.fail()) Error();
    if (x >= 1e100 || x <= -1e100) Error();
    if (x >= -1e-100 && x < 0) Error();
    if (x <= 1e-100 && x > 0) Error();
    return x;
}
vector<double> GetRandomNumbers(long NumberOfTerms) //Will NOT include upper
bound or lower bound.
{
    const long Max = 32768; //2^15.

    cout << "What would you like the upper bound to be?\n";
    double UpperBound = GetX();
    cout << "What would you like the lower bound to be?\n";
    double LowerBound = GetX();
    if (UpperBound <= LowerBound) Error();

    double DifferenceOfBounds = UpperBound - LowerBound;
    double Slide = UpperBound/DifferenceOfBounds - 1;

    time_t seconds; //www.cprogramming.com/tutorial/random.html
    time(&seconds);
    srand((unsigned int) seconds);
    vector<double> RandomNumberVector(NumberOfTerms);
    int Counter = 0;
do
    {
        RandomNumberVector[Counter] = ((RandomNumberVector[Counter] /
(Max)) + Slide) * DifferenceOfBounds;
        Counter++;
    }while (Counter < NumberOfTerms);
}

```

```

        return RandomNumberVector;
    }
void MaxPercentErrorEstimator()
{
    int FunctionChoice = Prompt(1);
    cout << "How many numbers would you like to sample?\n";
    long NumberOfTerms;
    cin >> NumberOfTerms;
    if (cin.fail() || NumberOfTerms < 0) Error();

    cout << "This is the for the X values\n";
    vector<double> RandomNumbersX = GetRandomNumbers(NumberOfTerms);
    vector<double> RandomNumbersY(NumberOfTerms);
    if(FunctionChoice == 1 || FunctionChoice == 2 || FunctionChoice == 3 ||
FunctionChoice == 4 || FunctionChoice == 6 || FunctionChoice == 8)
    {
        cout << "This is the for the Y values\n";
        vector<double> RandomNumbersY = GetRandomNumbers(NumberOfTerms);
    }
    else
    {
        int Counter = 0;
        do
        {
            RandomNumbersY[Counter] = 0;
            Counter++;
        }while (Counter < NumberOfTerms);
    }

    double PercentError = 0;
    double MaxPercentError = 0;
    int Counter = 0;
    do
    {
        PercentError = (CallFunction(FunctionChoice,
RandomNumbersX[Counter], RandomNumbersY[Counter]) -
CallCplusplusFunction(FunctionChoice, RandomNumbersX[Counter],
RandomNumbersY[Counter]))/CallCplusplusFunction(FunctionChoice,
RandomNumbersX[Counter], RandomNumbersY[Counter]);
        if (PercentError < 0) PercentError = -PercentError;
        if (PercentError > MaxPercentError) MaxPercentError =
PercentError;
        Counter++;
    }while (Counter < NumberOfTerms);
    cout << "Max percent error = " << setprecision(4) << MaxPercentError <<
"\n";
}
void EfficiencyEstimator()
{
    int FunctionChoice = Prompt(1);

    cout << "How many numbers would you like to sample?\n";
    long NumberOfTerms;
    cin >> NumberOfTerms;
    if (cin.fail() || NumberOfTerms < 0) Error();

    cout << "This is the for the X values\n";

```

```

    vector<double> RandomNumbersX = GetRandomNumbers(NumberOfTerms);
    vector<double> RandomNumbersY(NumberOfTerms);
    if(FunctionChoice == 1 || FunctionChoice == 2 || FunctionChoice == 3 ||
FunctionChoice == 4 || FunctionChoice == 6 || FunctionChoice == 8)
    {
        cout << "This is the for the Y values\n";
        vector<double> RandomNumbersY = GetRandomNumbers(NumberOfTerms);
    }
else
    {
        int Counter = 0;
        do
        {
            RandomNumbersY[Counter] = 0;
            Counter++;
        }while (Counter < NumberOfTerms);

        int Counter = 0;
        DWORD StartTick, EndTick;

        StartTick = GetTickCount();
        do
        {
            CallFunction(FunctionChoice, RandomNumbersX[Counter],
RandomNumbersY[Counter]);
            Counter++;
        }while (Counter < NumberOfTerms);
        EndTick = GetTickCount();
        double ElapsedTicks1 = (EndTick - StartTick);
        cout << "Estimator Function Elapsed Ticks: " << ElapsedTicks1 << "\n";

        Counter = 0;
        StartTick = GetTickCount();
        do
        {
            CallCplusplusFunction(FunctionChoice, RandomNumbersX[Counter],
RandomNumbersY[Counter]);
            Counter++;
        }while (Counter < NumberOfTerms);
        EndTick = GetTickCount();
        double ElapsedTicks2 = (EndTick - StartTick);
        cout << "C++ Function Elapsed Ticks:" << ElapsedTicks2 << "\n";
        double MultiplesSlower = ElapsedTicks1/ElapsedTicks2;
        cout << "The funtion is " << setprecision(4) << MultiplesSlower << "
times slower than the C++ built in function.\n";
    }
double CallCplusplusFunction(int choice, double x, double y)
{
    if (choice == 1) return x + y;
    if (choice == 2) return x - y;
    if (choice == 3) return x * y;
    if (choice == 4) return x / y;
    if (choice == 5) return sqrt(x);
    if (choice == 6) return pow(x, y);
    if (choice == 7) return exp(x);
    if (choice == 10) return sin(x);
}

```

```

        if (choice == 11) return cos(x);
        if (choice == 12) return tan(x);
        if (choice == 13) return asin(x);
        if (choice == 14) return acos(x);
        if (choice == 15) return atan(x);
        if (choice == 16) return log(x); // Base e.
        if (choice == 17) return log10(x); // Base 10.
        Error();
    }
double CallFunction(int choice, double x, double y)
{
    if (choice == 1) return Addition(x, y);
    if (choice == 2) return Subtraction(x, y);
    if (choice == 3) return Multiplication(x, y);
    if (choice == 4) return Division(x, y);
    if (choice == 5) return Sqrt(x);
    if (choice == 6) return Power(x, y);
    if (choice == 7) return Exp(x);
    if (choice == 10) return Sin(x);
    if (choice == 11) return Cos(x);
    if (choice == 12) return Tan(x);
    if (choice == 13) return ArcSin(x);
    if (choice == 14) return ArcCos(x);
    if (choice == 15) return ArcTan(x);
    if (choice == 16) return Ln(x); // Base e.
    if (choice == 17) return Log(x); // Base 10.
    Error();
}
string Operator(int Choice)
{
    string temp = "";
    if (Choice == 1) temp = "\nx + y = ";
    if (Choice == 2) temp = "\nx - y = ";
    if (Choice == 3) temp = "\nx * y = ";
    if (Choice == 4) temp = "\nx / y = ";
    if (Choice == 5) temp = "\nSquare Root(x) = ";
    if (Choice == 6) temp = "\nx ^ y = ";
    if (Choice == 7) temp = "\ne^ (x) = ";
    if (Choice == 8) temp = "\nx (mod y) = ";
    if (Choice == 10) temp = "\nsin(x) = ";
    if (Choice == 11) temp = "\ncos(x) = ";
    if (Choice == 12) temp = "\ntan(x) = ";
    if (Choice == 13) temp = "\narcsin(x) = ";
    if (Choice == 14) temp = "\narccos(x) = ";
    if (Choice == 15) temp = "\narctan(x) = ";
    if (Choice == 16) temp = "\nLn(x) = ";
    if (Choice == 17) temp = "\nLog(x) [Base 10] = ";
    if (temp == "") Error();
    return temp;
}
void DisplayAnswer(int Choice)
{
    double x = GetX();
    if (Choice == 1 || Choice == 2 || Choice == 3 || Choice == 4 || Choice
== 6 || Choice == 8)
    {
        double y = GetX();

```

```

        cout << Operator(Choice) << setprecision(10) <<
CallFunction(Choice, x, y) << "\n";
    }
    else if (Choice == 5 || Choice == 7 || Choice == 10 || Choice == 11 ||
Choice == 12 || Choice == 13 || Choice == 14 || Choice == 15 || Choice == 16
|| Choice == 17)
    {
        cout << Operator(Choice) << setprecision(10) <<
CallFunction(Choice, x, 0) << "\n";
    }
    else Error();
}
int Prompt(int Reason)
{
    if (Reason != 0 && Reason != 1) Error();
    if (Reason == 0)
        cout << "What function of the following would you like to
compute? ";
    if (Reason == 1)
        cout << "What function of the following would you like estimate
with? ";

    cout
    << "\n(1) Addition (10) Sin(x)"
    << "\n(2) Subtraction (11) Cos(x)"
    << "\n(3) Multiplication (12) Tan(x)"
    << "\n(4) Division (13) ArcSin(x)"
    << "\n(5) Square Root (x) (14) ArcCos(x)"
    << "\n(6) x ^ (y) (15) ArcTan(x)"
    << "\n(7) Exp(x) (16) Ln(x)"
    << "\n(8) x (mod y) (17) Log(x) [Base 10]";
    if (Reason == 0)
    {
        cout
        << "\n(19) Maximum Percent Error of a Function(x)"
        << "\n(20) Efficiency of a Function(x)"
        << "\n(0) None, I am done";
    }

    cout << "\nPlease enter the number of the function that you
would like to compute.\n";
    int Choice;
    cin >> Choice;
    if (cin.fail() || Choice < 0 || Choice > 20) Error();
    return Choice;
}
int main()
{
    try
    {
        bool stop = false;
        int choice;
        do{
            choice = Prompt(0);
            if (choice == 0) stop = true;
            if (choice != 0 && choice != 9 && choice < 18) DisplayAnswer(choice);
            if (choice == 19) MaxPercentErrorEstimator();
            if (choice == 20) EfficiencyEstimator();
            cout << "\n";
        } while(stop == false);
    }
}

```

```
    }
    catch (ReturnError)
    {
        cin.clear();
        cout << "Error: Invalid Entry!\nProgram will be Restarted.\nPush
any key and 'Enter' to continue...\n";
        string temp;
        cin >> temp;
        cin.clear();
        main();
    }
    return 0;
}
void Error()
{
    throw ReturnError(0);
}
```