4G04 Assignment 1

Shayaan Siddiqui
400247500
siddis41

September 27,2021

Abstract

This first assignment has induced a study in a wide variety of content. The initial concept learnt deals with how to convert different number systems using different bases. The next concept learnt involved a unique differential function called the Bessel function. Through this assignment multiple methods of recursion have been implemented to preform approximations. Next, three methods were used to evaluate the square root function; the Taylor series, Newton's method and Continued fractions. The final concept studied was the implementation of the Trapezoidal and Romberg method of integration.

1 Introduction

The coding language that is being used to construct the following tasks is C++. To truly get a grasp of how computations occur within a computer, it is essential to understand how values are stored within a system. A computer will understand the polarity of a numerical value through a method called either ones or twos compliment. To extract decimal values, a 32 bit computer will store 1 bit to determine the polarity, 8 to determine the exponent and 23 bits in a region called the mantissa to determine the decimal points. Section 1.1 will go in depth as to how the code implements this system of conversion to convert a hexadecimal to binary to a floating point.

1.1 Hexadecimal to Binary Converter

Hexadecimals is a number system that work in base 16. The hexadecimal system incorporates letters into the system to include terms 10-16. Each term starting from the right most side of the hexadecimal has a base of 16 and a power of n.

The beginning of the hexadecimal, n = 0, and as you continue to the left, value of n increases by 1.

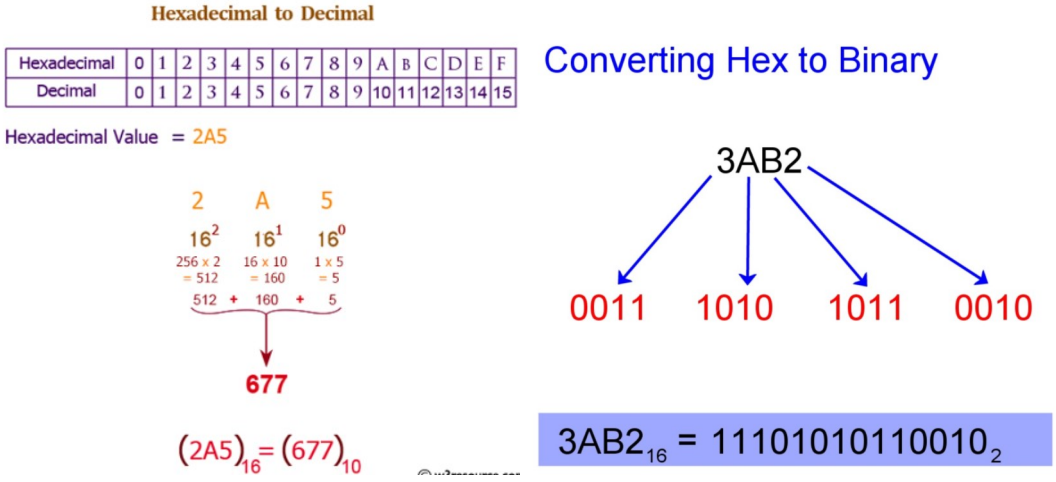

Figure 1: How to convert hexadecimal to decimal and binary.

The term in specific index is the constant multiplied to the exponent. To convert a hexadecimal to binary, it requires 4 bits in base 2. Each term in the hexadecimal can be converted directly to binary and the binary term should consist of 4 times the amount of terms the hexadecimal contains. Figure 1 shows an example of how to change a binary term. The code created use the switch and case function to read a string and output the binary code related to that hexadecimal. The variable type is initially stored as a string. The characters of the string are stored within a vector and converted into a integer.

```
int main()
{

    int i = 0; // init for index
    string value; // stores binary string content from each case
    string bin_value; //adds the string contents saved in value in the loop
    char hexDecNum[100]; // user input that accepts a large amount of characters
    cout << "Enter the 8 digit Hexadecimal Number: ";
    cin >> hexDecNum;
    cout << "\nEquivalent Binary Value = ";
    while (hexDecNum[i]) // loop that runs till index of the user input
    {
        switch (hexDecNum[i]) // read user input at each index and replace with binary term for each specific case
        {
        case '0':
            value = "0000";
            break;
        case '1':
            value = "0001";
            break;
        case '2':
            value = "0010";
            break;
        case '3':
            value = "0011";
            break;
```

Figure 2: Code that switches hexadecimal to the string values

```
        bin_value = bin_value + value; // summing into full binary term
        i++;
    }

    cout << bin_value; // outputs string of binary code
    cout << endl;
    int j = bin_value.length(); // making integer length of the binary term

    cout << endl;
    vector<int> bin_vals_list; // vector to include each term of binary string
    cout << endl;
    int num; //stores numerical values

    for (int p = 0; p < j; p++) { // for loop to fill vector of integer values for each binary term
        stringstream ss;// allows conversion from string to integer
        ss << bin_value[p];
        ss >> num; // stores integer value converted from string at index of for loop
        bin_vals_list.push_back(num); // adds value to end of the the vector
    }
    cout << endl;
```

Figure 3: Code that stores each bit of binary into a vector

When running the codes provided and entering the hexadecimal "B370B000" given in the assignment, the following is the binary output.

```
Enter the 8 digit Hexadecimal Number: b370b000

Equivalent Binary Value = 10110011011100001011000000000000
```

Figure 4: Binary output from hexadecimal input

1.2 Binary to floating point

Section 1.1 describes how to convert a hexadecimal into binary format. Now that the binary format has been found, this can be converted into a floating point. For the purpose of this value, the binary will be a 32 bit value. All this means is the system runs on 32 bits to preform operations. The introduction briefly discussed the polarity, exponent and the mantissa. The following figures will show the format of how a 32 bit binary code will be converted into a floating point value.

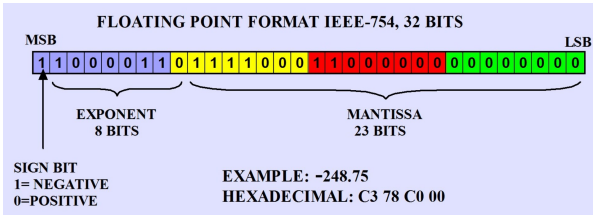

Figure 5: Location of all 32 bits in float format

$$(-1)^{b_1} \times 2^{(b_2 \ldots b_9)_2} \times 2^{-127} \times (1.b_{10}b_{11} \ldots b_{32})_2$$

Figure 6: How to manipulate bits to form floating point values

The left most bit is the first bit, this is used to determine the polarity of the float value. If the term is 1, then the float is negative, if 0 the opposite. Then the first term of the exponent section has the power of 7. This is the second bit. The following bits decrease in power by 1 till it reaches a power of -32 at the last bit which is in the mantissa. Both the mantissa and exponent will be converted to base 2, as shown in Figure 6. Then the base 2 value of the exponent will be used as an exponent and the decimal value of the mantissa will be added to 1. In the code made, a function was made to recreate these transformations.

```
void What_is_float_val(vector <int> g)
{
    double float_val//final float output
        , power = 0//initial power
        , holder//used for manipulations
        , sign//polarity of float
        , count = 7//power that targets 8th bit and lower
        , count_2 = -1;//power that targets 10th bit and higher
    double mantissa;// stores base 10 value of mantissa
    if (g[0] == 1) {//if statment that determines if the float is positive or negative
        sign = -1;
    }
    else {
        sign = 1;
    }
    for (int i = 1; i < 9; i++) {//for loop that goes from the second bit to the 9th bit
        holder = g[i] * pow(2, count);//multiply binary term by 2^count starting at 7 then decrease count. Conversion to base 10 value
        power = power + holder;//add place holder to exponential term
        count--;//decrease count
    }
    if (power > 0) {//if the exponential term exists then the mantissa starts at 1
        mantissa = 1;
    }
    else {
        mantissa = 0;//if exponential term doesnt exist then mantissa starts at 0
        power = 1;//since the exponent would be -126 instead of -127
    }
    for (int k = 9; k < 32; k++) {//for loop that starts at 10th bit and goes to the 32rd bit
        holder = g[k] * pow(2, count_2);//conversion to base 10 value
        mantissa = mantissa + holder;
        count_2--;
    }

    float_val = sign * pow(2, power - 127) * mantissa; //final formula given to equate floating point
    cout << "The floating point of the hexadecimal given is:" << float_val;
```

Figure 7: Function produced that takes any 32 bit binary code and output floating point value

The code in Figure 7 converts the binary terms into base 10 such as the formula requires and then preforms the required operations. Keep in mind that the first bit index is zero so each bit index is one less than which bit it is. When inputting the previous hexadecimal code, the following is the output.

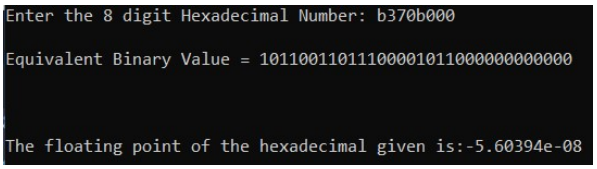

Figure 8: Float value for the hexadecimal "B370B000"

This floating point value is indeed correct for the hexadecimal given and when inputting other hexadecimal values, the correcting floating point decimal is outputted.

2 Introduction to Bessel functions

The Bessel function is a unique function that includes a value of n which causes the differential function to have an order. The solution of the function can be denoted as $J_n(x)$. The following equation is what the Bessel function is denoted as.

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - n^2)y = 0 \tag{1}$$

There are many applications of the Bessel function. Essentially the purpose is to solve the wave function in 3D at specific frequencies. The Bessel function can be used to solve for physics problems in the cylindrical or spherical coordinates. These values will what the order of the function is. This can relate to the amount of nodes or order in a physics problem. Some examples of where a Bessel function would be applied is in radial Schrödinger equations, Diffusion on a lattice, Heat conduction on a cylinder, Electromagnetic waves in cylindrical wave-guide and more.

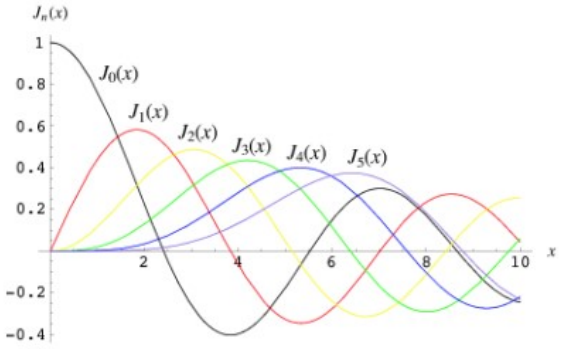

Figure 9: First few Bessel functions plot

2.1 Bessel Function Ascending Recurrence Relation

The first part of this problem requires a program built that solves the Bessel function in an ascending manner for each order. For this part of the problem, the x value will stay fixed at x = 0.5. The first two order solution are given as the following

$$J_0(0.5) = 0.938469807240813\ ,\ J_1(0.5) = 0.2422684577 \tag{2}$$

Using this formula, we can use recursion relationship to solve for the next 20 terms as asked for

$$J_{n-1}(x) = \frac{2n}{x}J_n(x) - J_{n+1}(x) \tag{3}$$

We can rearrange this formula as such :
$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x)$ To implement this in terms of code, a vector is formed to dynamically store the new order solution for the recursion. After solving for the new value, this can be reused to solve for the rest of the terms. The following figure will show the code implementation. When running

```cpp
#include <iostream>
#include<stdio.h>
#include <math.h>
#include <vector>

using namespace std;
int main()
{
    double x = 0.5;//x value held constant at 0.5

    //vector containing first 2 terms of bessel function at x=0.5 and order=index
    vector<double> J_of = { 0.938469807240813, 0.2422684577 };

    //print out order of 0 and 1 of bessel function at x=0.5
    cout << endl << "The 0 term: " << J_of[0] << endl << "The 1 term: " << J_of[1] << endl;
    for (int n = 0; n < 19; n++) {//for loop to find the next 18 order of bessel function
        int m = n + 1;//variable to get next index
        double next_term = J_of[m] * 2 * (m) / x - J_of[n];

        //recursion relationship to get next order function
        J_of.push_back(next_term);
        cout << "The " << n + 2 << " term: " << next_term << endl;
        //output the next terms
    }
}
```

Figure 10: Code following ascending recursion

this code, a enormous number is output. This is known to be false as when looking at the plot shown in Figure 9, as the order increases the y value at x=0.5 decreases. The next 20 terms at each order is shown in Figure 11.

```
The 0 term: 0.93847
The 1 term: 0.242268
The 2 term: 0.030604
The 3 term: 0.00256373
The 4 term: 0.000160746
The 5 term: 8.20079e-06
The 6 term: 3.27009e-06
The 7 term: 7.02814e-05
The 8 term: 0.00196461
The 9 term: 0.0627972
The 10 term: 2.25874
The 11 term: 90.2866
The 12 term: 3970.35
The 13 term: 190487
The 14 term: 9.90134e+06
The 15 term: 5.54284e+08
The 16 term: 3.32472e+10
The 17 term: 2.12726e+12
The 18 term: 1.44621e+14
The 19 term: 1.04106e+16
The 20 term: 7.91058e+17
```

Figure 11: First 20 Ascending order Bessel Function output

The data shows that initially the terms would reach a small magnitude and then bounces back up.

This is due to the factor of n and how this changes the magnitude of the current term to become much larger than the previous term. It is obvious that this is not the method to follow.

2.2 Bessel Function Descending Recurrence Relation

To solve the issue brought by using the Ascending method, we can use the Descending method. To implement this method, the are a few techniques that have to be followed.

$$J_N = 1, \quad J_m = 0 \, \forall \, m > N \tag{4}$$

First find a reasonable value of N that would be the largest order solution that has a non-zero value. Then all solutions of larger order would be zero. Then the descending recurrence relation
$J_{n-1}(x) = \frac{2n}{x} J_n(x) - J_{n+1}(x)$
can be used until $J_0(x)$ is found. After this term is found, another relationship can be made to solve for a constant that all terms can be divided by to completely for all values of $J_n(x)$.

$$J_0(x) + 2\sum_{n=1}^{\infty} J_{2n}(x) = 1 \tag{5}$$

When using the descending method, equation 5 can be used as a re-normalization, the output of the equation will be a constant that all solutions of $J_n(x)$ are divided with to give an accurate value. The solution done by hand will be the following calculations.

$$x = 0.5 \;\; J_5(0.5) = 0 \quad J_4(0.5) = 1 \tag{6}$$

$$J_3(0.5) = \tfrac{(2)(4)}{0.5}(1) - 0 = 16$$

$$J_2(0.5) = \tfrac{(2)(3)}{0.5}(16) - 1 = 191$$

$$J_1(0.5) = \tfrac{(2)(2)}{0.5}(191) - 16 = 1512$$

$$J_0(0.5) = \tfrac{(2)(1)}{0.5}(1512) - 191 = 5857$$

The normalization constant is $5857 + 2(191 + 1) = 6241$

$J_0(0.5) \approx 0.938, \;\; J_1(0.5) \approx 0.242, \;\; J_2(0.5) \approx 0.031$

$J_3(0.5) \approx 2.56(10)^{-3}, \;\; J_4(0.5) \approx 1.60(10)^{-4}, \;\; J_5(0.5) = 0$

Now that the solution has been solved by hand, this can be confirmed by creating a program.

2.3 C++ Bessel Function Descending Recurrence Relation

The code for the descending recurrence relationship is essentially the same as the ascending code except for the fact that there needs to be another for loop included to output the summation included in the normalization formula.The code was created in a way that the solutions of n, n+1 and n-1 are output at each iteration. I can use this to ensure the hand solution is indeed correct. By increasing the highest order of the solution, the outputs will become more accurate since the normalization constant will become more defined. This can be seen in the following figures. As shown by the image above, with more iterations, the code ends up outputting values of higher precision.

```
0  5 term
1  4 term
16  3 term

1  4 term
16  3 term
191  2 term

16  3 term
191  2 term
1512  1 term

191  2 term
1512  1 term
5857  0 term


the normalization term is: 6241

Renormalized bessel function output at each order:
0 Order: 0.938471398814293
1 Order: 0.242268867168723
2 Order: 0.0306040698605993
3 Order: 0.00256369171607114
4 Order: 0.000160230732254446
5 Order: 0
```

Figure 12: First 5 terms confirmation

```
2.93156e+09  2 term
2.32069e+10  1 term
8.9896e+10  0 term


the normalization term is: 9.579e+10

Renormalized bessel function output at each order:
0 Order: 0.938469807241304
1 Order: 0.242268457675001
2 Order: 0.0306040234586987
3 Order: 0.00256372999458859
4 Order: 0.000160736476364372
5 Order: 8.05362724136167e-06
6 Order: 3.36068462861721e-07
7 Order: 1.20158673196397e-08
8 Order: 3.75822088190295e-10
9 Order: 1.04395024497304e-11
10 Order: 0
```

Figure 13: First 10 terms

3 Zeros of a Bessel Function

The solutions of a Bessel function are closely intertwined with sin and cos functions. This means that the roots are related to pi in some fashion. In the case of a Bessel function, for a specific solution $J_n(x)$ the distance between each root approaches π. Since the solution of the Bessel differential function is complicated, the recurrence relation solved in Section 2 can be used in the descending fashion. The C++ implementation would require a function built that would solve an approximation for the desired nth solution to the Bessel function. For the purposes of this question, the order of focus is the zeroth order. Thus the function built in c++ will output the zeroth order of the Bessel function at a specific value of x using the value of x as the parameter. To then find the roots, the bisection method can be applied. The concept of this method is to essentially bracket around a point by picking 2 points, a and b respectively, that have opposite polarity. Then find the mid point. If the midpoint and point a are on opposite sides, change the midpoint into b and redo this process till there is a very small gap between

the 2 points. The code implementation can be seen by the following function built. We can now use

```
/*function that preforms bisection method
given 2 points that bracket the root*/
double bisection(double a, double b) {
    double c;//midpoint
    if (bessel(a) * bessel(b) < 0) {//make sure y values of points different polarity
        for (int i = 0; i < 50; i++) {
            c = (a + b) / 2;//reseting midpoint
            if (bessel(c) == 0) {
                //if midpoint is root return value
                return c;
            }
            //if bessel at c and a are opposite sides, move b to c
            else if (bessel(c) * bessel(a) < 0) {
                b = c;
            }//if bessel at c and b are opposite sides, move a to c
            else {
                a = c;
            }
        }//output closest found value to root
        return c;
    }
    else {
        cout << "Invalid brackets";
        return 0;
    }
}
```

Figure 14: C++ implementation of Bisection method

Figure 9 to get an idea of the brackets to place for the first 2 roots of $J_0(x)$. The first bracket chosen is 2,3 the second is 5,6 and the third is 8,9. Now by running the code and measuring the distance between the output of the bisection method we can confirm if the distance is in fact approaching π. The figure shows that the consecutive distance between the roots is approaching the value pi.

```
The first root is: 2.40483
The secont root is: 5.52008
The third root is: 8.65373

The distance between the first 2 roots are: 3.11525
The distance between the second and third roots are: 3.13365
```

Figure 15: C++ output of Bessel Function roots

4 Evaluate Square Root Function using Different Approximations

4.1 How to change the document language and spell check settings

The $\sqrt{x}$ function is an easy enough function to evaluate but it is useful tool to use to really understand how approximation methods work. Since the function is simple to manipulate the mathematics involved is not too complicated. The 3 approximation methods that are going to be used are the Taylor series, Newton's method and repeated fraction.
First to implement the Taylor series in C++ a function that requires 2 parameters is created, requiring the amount of iterations and the x value that is going to be calculated. Next a function that can preform factorials needs to be made since C++ does not have a built in function for this.

```
double fact(double x) {
    double outcome=x;//set value as parameter
    if (x == 0) {
        outcome = 1;//factorial of 0
        return outcome;
    }
    else {
        /*for loop that starts
        @ value of parameter
        and continues to
        multiply by index minus 1
        till finally multiplied by 1*/
        for (double i = x; i > 1; i--) {
            outcome = outcome * (i - 1);
        }
        return outcome;//return factorial outcome
    }

}
```

Figure 16: Factorial Function

Taylor series heavily relies on derivatives. The format is the following
$\sum_{n=0}^{\infty} \frac{f^n(a)}{n!}(x-a)^n$
In this situation, using an approximation for a derivative would be tedious and produce a large amount of error. Instead consider the mathematical relationship between a function and its derivative.

$$f(x) = ax^n, f'(x) = nax^{(n-1)} = na\frac{x^n}{x^1}$$
$$f''(x) = n(n-1)(a)\frac{x^n}{x^2}, f'''n(n-1)(n-2)(a)\frac{x^n}{x^3}, f''''...$$

Using this relationship, the code implemented finds the correct constant and correct exponent in terms of the for loop creating the sum to correctly output the derivative of the function. The center of the Taylor series is always 1 less than the x value of the square root function and this will let the Taylor series approximation be accurate. The code to preform this approximation is show below.

```
/*taylor series function using amount of iterations and x value as a parameter*/
double Sq_Tlr(double x, double N) {
    //set value of power
    double n=0.5

        //multiplication factor cause by derivative
        ,constant=0.5

        //center of taylor series
        , a=x-1;
    double series=0;//initial series sum
    double function1//function holder
        ,function2= sqrt(a);//initial function

    /*loop that runs for amount of iterations
    applying derivative then adding to taylor series*/
    for (double k = 1; k <= N; k++) {
        function1 = function2 / (pow(a, k));//derivative
        //taylor series
        series=series+(constant*function1*pow((x-a),k))/fact(k);
        //new constant due to derivative
        constant = constant * (n - k);
    }
    //adding T0 to the series
    series = series + function2;
    return series;//outputs approx solution
}
```

Figure 17: C++ Taylor Series Approximation

The next method is Newtons method. Newtons method is normally used to solve for a root but since it is used in this situation to solve for the solution of the function. Some simple algebra can be

preformed to solve this problem. For example if the $sqrt(3)$ is taken, the solution to this is $x^2 = 3$. This means that $f(x) = x^2 - 3$. In terms of C++, since recursion is used, a vector can be made, the x value that is to be input into the square root is now going to used as a constant for a new function that uses an arbitrary starting value in the vector constructed. The function will look as the following $f(x_1) = x_1^2 - x$. Newtons method is actually a simple rearrangement of the Taylor series.

$0 = f(x) + f'(x)(x_2 - x_1)$
$x_1 - \frac{f(x)}{f'(x)} = x_2$

Since the x_1 term is arbitrary, the code will set it to 0 and let the algorithm refine and solve for the actual value. Since only a first derivative is used in this method, an analytical approach can be used. The three point stencil is standard practice when applying a derivative. This has an error that is a function in order of h^2. The formula can be described as $f'(x) = \frac{f(a+h)-f(a-h)}{2h}$. The code built in C++ is the following.

```
double Sq_Nwt(double x,double N)
{

    vector<double> xk;//vector for recursion relation

    //fills vector with amount of index
    for (double i = 0; i <= N; i++) {
        xk.push_back(i + 1);
    }
    //derivative x point deviation value
    double h = 0.00000001;

    //for loop preforming newton approx relationships
    for (double k = 1; k <= N; k++) {
        double m = k - 1;
        //rearrangement of formula to solve for input of x
        double func = pow(xk[m], 2) - x;
        //derivative of function
        double diff = ((pow(xk[m] + h, 2) - x) - (pow(xk[m] - h, 2) - x)) / (2 * h);
        //newtons recursion relationship
        xk[k] = xk[m] - func / diff;
    }
    //output approx solution
    return xk[N];
}
```

Figure 18: C++ Newton's Approximation

The final method used to approximate the square root function is the method of repeated fractions. This method is non-unique and there are multiple ways of implementing this method. The first method would be to describe the square root function as such: $\sqrt(x) = a + \sqrt(x) - a$ The next step would be to multiply by the conjugate of $\sqrt(x) - a$ but as a fraction such that you are multiplying the equation by 1. The next steps would look like:
$a + \frac{(\sqrt(x)-a)(\sqrt(x)+a)}{(\sqrt(x)+a)} = a + \frac{(x-a^2)}{(\sqrt(x)+a)} = a + \frac{(x-a^2)}{(\sqrt(x)+2a-a)}$
It is known from previous evaluations that:
$\sqrt(x) - a = \frac{(x-a^2)}{(\sqrt(x)+a)}$
Thus anytime this term is recreated, this substitution can be made, some 0 addition and 1 multiplication can rearrange the formula to continue this fraction. For the C++ implementation, an easier continuous fraction was used. The following continued fraction was used in the C++ code but another could have been used. $\lfloor\sqrt(x)\rfloor + \frac{x-(\lfloor\sqrt(x)\rfloor)^2}{2\lfloor\sqrt(x)\rfloor + \frac{x-(\lfloor\sqrt(x)\rfloor)^2}{2\lfloor\sqrt(x)\rfloor \ldots}}$
To formulate this function, rather than starting at the numerator, it is better to start at the Nth iteration fraction and consistently make that the denominator for the following iterations. The code construct is as shown below:

```
//repeated fractions function that uses function as pointer
//repeated fractions relation is not unique
//this function is absolute and requires less algebra
double Sq_Frc(double (*f)(double x), double x, double N)
{
    double a0, approx,num;
    //initial term
    a0 = floor(f(x));

    //numerator of repeated fraction
    num = x - pow(a0, 2);

    //bottom fraction
    approx = num/(2*a0);

    //for loop that continues the fraction
    for (int i = 0; i <= N; i++) {
        /*takes bottom fraction, adds 2*a0 and
        makes it denominator to numerator repeatedly*/
        approx = pow(approx + 2 * a0, -1) * num;
    }
    approx = approx + a0;//adds a0 term
    return approx;//outputs approx solution
}
```

Figure 19: C++ Repeated fraction Code

To test the effectiveness of this code, the relative error of each approximation and how long it took to compute the algorithm is outputted to give a good concept of when to apply which method. The

```
The x term for the functions is 3 and amount of iterations is 10

Square roots in taylors methods are: +-1.73204695621271   Exact value: 1.73205080756888
Relative error of taylor series is: -2.22358151892003e-06
Time: 0.006 s

Square roots in newtons methods are: +-1.73205080756888   Exact value: 1.73205080756888
Relative error of Newtons method is: 1.28197512425571e-16
Time: 0.004 s

Square roots in repeated fractions are: +-1.73205068043172   Exact value: 1.73205080756888
Relative error of Newtons method is: -7.34026706782593e-08
Time: 0.004 s
```

Figure 20: Effectiveness of each method

Taylor series took the longest and had the largest error. The repeated fractions took the same amount of time as newtons method but had a larger error. Newtons method is the best as it produces the smallest error and takes the least amount of time.

5 Trapezoidal and Romberg Integration

The trapezoidal integration uses a similar technique as the Riemann sum except rather than making a rectangle, a trapezoid would be made to approximate the area in a specific interval under the curve.

The Romberg integration uses a method called Richardson extrapolation to continuously produce new functions that will cancel out the error in the previous function using recursion thus minimizing the final error in the output. It is used with the trapezoidal rule to remove errors.

5.1 Trapezoidal rule

The trapezoidal function can be equated to the sum:

$$\sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k \tag{7}$$

Δx_k is solved by taking two x values subtracting them and dividing by how many intervals of the sum exist. The fraction essentially creates an average x value and multiplying by Δx_k gives the approx. area the curve within the specific interval. Notice the sum will only contain $f(x_0)$ and $f(x_N)$ once but every other term within the sum twice. To implement into a code, first solve for the interval size using 2 points as parameters for the function and 1 parameter for amount of intervals. Then a for loop is built where the index of the loop multiplied by the interval size and added on the lower bound. This gives a specific x value. Then find the y value at this point, calculate 2 times the product and create a summation of that. Finally after the loop is complete add the function value at the lower and upper bounds. The code has the following format.

```
//trapezoidal function to solve integral
//function, bounds and interval amount are parameters
double Trapezoidal(double (*f)(double x), double xl, double xh, int inl)
{
    double each_interval = (xh - xl) / inl;//deviance for each interval
    double integration = f(xl) + f(xh);//first and last term of sum
    double c;//each x point for interval

    for (double i = 1; i < inl; i++)
    //run through loop and add 2 times the y value at each interval for summation
    {
        c = xl + i * each_interval;
        integration = integration + 2 * (f(c));
    }
    //area under the curve using found values
    integration = integration * each_interval / 2;
    return integration;
}
```

Figure 21: C++ function for Trapezoidal integration

5.2 Romberg's method

The final topic explored is Romberg's method. Romberg's method is applied to both trapezoidal rule and midpoint rule. By start at an index or Romberg function R(0,0) it gives an initial reference point in respect to which integration approximation method used. Then by running thorough each R(n,0) value error is slowly being cancelled out.Then all R(n,m) values would be found by solving for every n at each m index, effectively removing all error. To preform this a nested for loop will have to be built. The recursion relationships are as followed

$$R(n,m) = R(n,m-1) + \frac{1}{4^m - 1}\left[R(n,m-1) - R(n-1,m-1)\right].$$

$$R(n,0) = R(n-1,0)/2 + h\sum_{k=1}^{2^{n-1}} f(x_l + (2k-1)h).$$

Figure 22: Romberg Recursion Relationships

This is also known as Richardson extrapolation and the reason the error is minimized is due to the fact that the error becomes a function of $\frac{h}{2^n}$. This becomes so minuscule as the N increases.The value of h would be $\frac{xh-xl}{2^n}$.The code to implement this presented in the following image.

```
double Romberg(double (*f)(double x), double xl, double xh, double N)
{
    vector<vector<double>> r;//2D vector
    vector<double> h;//vector for varying h values

    double a = (N) + 1;
    //for loop to fill vectors
    for (double q = 0; q < a; q++) {
        h.push_back(1);
        vector<double>v;
        for (double o = 0; o < a; o++) {
            v.push_back(1);
        }
        r.push_back(v);
    }
    //solution for each h term
    for (int i = 1; i <= N; i++)
    {
        h[i] = (xh - xl) / pow(2, i - 1);
    }
    //initial romberg value
    /*keep in mind initialized index starts
    at 1,1 and not 0,0 so remember for final plug in*/
    r[1][1] = h[1] / 2 * (f(xl) + f(xh));
    //for loop to solve for first variable of romberg fuction
    //ie. all R(i,0)
    for (int i = 2; i <= N; i++)
    {
        double coeff = 0;
        // summation for loop for recursion relationship
        for (double k = 1; k <= pow(2, i - 2); ++k)
        {
            coeff += f(xl + (2 * k - 1) * h[i]);
        }
        //solution to all R(i,0) for each index of for loop
        r[i][1] = 0.5 * (r[i - 1][1]) + h[i] * coeff;
    }

    //final recusion relationship using previous R(i,0) values
    /*solving for R(i,j), first running through
    all j vaues for each i and then increase i and repeat
    til i=j*/
    for (int i = 2; i <= N; i++)
    {
        for (int j = 2; j <= i; j++)
        {
            r[i][j] = r[i][j - 1] + (r[i][j - 1] - r[i - 1][j - 1]) / (pow(4, j - 1) - 1);
        }
    }
    //output solution to integral
    return r[N][N];
}
```

Figure 23: Romberg C++ implementation

When changing the n value from 6-8 the amount of relative error drastically decreases but when changing it from 8-10 the magnitude does not change too significantly. My conclusion is there must be a level of precision the computer can detect before the computations become too hard for the computer to preform

5.3 Comparison Between Trapezoidal and Romberg Methods

These two methods have very different levels of precision. The Romberg function has a nested for loop so even though the n value is quite now, the amount of calculations the computer is preforming is much

larger than the trapezoidal function when n=10 for Romberg and inl =64 for trapezoidal. The first comparison output is: The Romberg function is much more accurate and the deviation is much lower

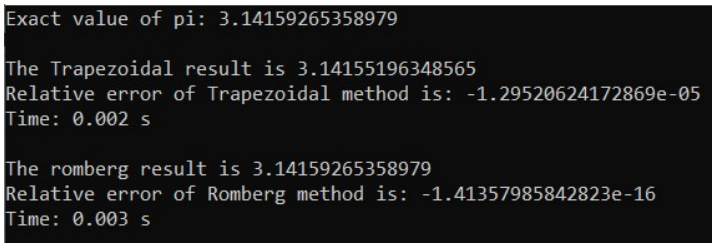

Figure 24: Comparison of Romberg and Trapezoidal functions with given parameters

in comparison to the trapezoidal function. A tradeoff of being so accurate is it took a small amount of time longer than the trapezoidal function to compute. This is really no problem but as the n value for the Romberg function explored into the double digits, the code would not be able to output whereas the trapezoidal function would output values for very large values of intervals. The romberg function is definetly more precise but takes longer to compute in comparison to the trapezoidal function.

6 Conclusion

In doing this assignment, a variety of content has been absorbed, many being how to efficiently approximate functions. Another skill gained is the evaluation of which approximation is beneficial per each unique scenario given and how to compare which function is better to use.

7 Code Appendix

7.1 Code Q1

```
//4G04 Q1
#include <iostream>
#include<stdio.h>
#include <vector>
#include <math.h>
#include <sstream>



using namespace std;
void What_is_float_val(vector <int> g)
{
    double float_val//final float output
        , power = 0//initial power
        , holder//used for manipulations
        , sign//polarity of float
        , count = 7//power that targets 8th bit and lower
        , count_2 = -1;//power that targets 10th bit and higher
    double mantissa;// stores base 10 value of mantissa
    if (g[0] == 1) {//if statment that determines if the float is positive or negative
        sign = -1;
    }
    else {
        sign = 1;
    }
    for (int i = 1; i < 9; i++) {//for loop that goes from the second bit to the 9th bit
        holder = g[i] * pow(2, count);//multiply binary term by 2^count starting at 7 then decrease c
        power = power + holder;//add place holder to exponential term
        count--;//decrease count
    }
    if (power > 0) {//if the exponential term exists then the mantissa starts at 1
        mantissa = 1;
    }
    else {
        mantissa = 0;//if exponential term doesnt exist then mantissa starts at 0
        power = 1;//since the exponent would be -126 instead of -127
    }
    for (int k = 9; k < 32; k++) {//for loop that starts at 10th bit and goes to the 32rd bit
        holder = g[k] * pow(2, count_2);//conversion to base 10 value
        mantissa = mantissa + holder;
        count_2--;
    }

    float_val = sign * pow(2, power - 127) * mantissa; //final formula given to equate floating point
    cout << "The floating point of the hexadecimal given is:" << float_val;

}
int main()
{

    int i = 0; // init for index
    string value; // stores binary string content from each case
    string bin_value; //adds the string contents saved in value in the loop
```

```
char hexDecNum[100]; // user input that accepts a large amount of characters
cout << "Enter the 8 digit Hexadecimal Number: ";
cin >> hexDecNum;
cout << "\nEquivalent Binary Value = ";
while (hexDecNum[i]) // loop that runs till index of the user input
{
    switch (hexDecNum[i]) // read user input at each index and replace with binary term for each
    {
    case '0':
        value = "0000";
        break;
    case '1':
        value = "0001";
        break;
    case '2':
        value = "0010";
        break;
    case '3':
        value = "0011";
        break;
    case '4':
        value = "0100";
        break;
    case '5':
        value = "0101";
        break;
    case '6':
        value = "0110";
        break;
    case '7':
        value = "0111";
        break;
    case '8':
        value = "1000";
        break;
    case '9':
        value = "1001";
        break;
    case 'A':
    case 'a':
        value = "1010";
        break;
    case 'B':
    case 'b':
        value = "1011";
        break;
    case 'C':
    case 'c':
        value = "1100";
        break;
    case 'D':
    case 'd':
        value = "1101";
        break;
    case 'E':
    case 'e':
```

```
            value = "1110";
            break;
        case 'F':
        case 'f':
            value = "1111";
            break;
        default:
            cout << "--Invalid Hex Digit (" << hexDecNum[i] << ")--";
        }
        bin_value = bin_value + value; // summing into full binary term
        i++;
    }

    cout << bin_value; // outputs string of binary code
    cout << endl;
    int j = bin_value.length(); // making integer length of the binary term

    cout << endl;
    vector<int> bin_vals_list; // vector to include each term of binary string
    cout << endl;
    int num; //stores numerical values

    for (int p = 0; p < j; p++) { // for loop to fill vector of integer values for each binary term
        stringstream ss;// allows conversion from string to integer
        ss << bin_value[p];
        ss >> num; // stores integer value converted from string at index of for loop
        bin_vals_list.push_back(num); // adds value to end of the the vector
    }
    cout << endl;

    What_is_float_val(bin_vals_list);
    cout << endl;
    return 0;

}
```

7.2 Code Q2

```
// 4G03 Q2
#include <iostream>
#include<stdio.h>
#include <math.h>
#include <vector>

using namespace std;
int main()
{
    double x = 0.5;//x value held constant at 0.5

    //vector containing first 2 terms of bessel function at x=0.5 and order=index
    vector<double> J_of = { 0.938469807240813, 0.2422684577 };

    //print out order of 0 and 1 of bessel function at x=0.5
    cout << endl << "The 0 term: " << J_of[0] << endl << "The 1 term: " << J_of[1] << endl;
    for (int n = 0; n < 19; n++) {//for loop to find the next 18 order of bessel function
        int m = n + 1;//variable to get next index
        double next_term = J_of[m] * 2 * (m) / x - J_of[n];

        //recursion relationship to get next order function
        J_of.push_back(next_term);
        cout << "The " << n + 2 << " term: " << next_term << endl;
        //output the next terms
    }
}
// 4G03 Q2.3

#include <iostream>
#include<stdio.h>
#include <math.h>
#include<iomanip>
#include <vector>

using namespace std;
int main()
{
    double x = 0.5;//at x value 0.5
    int N=10;//intervals
    vector<double> J_of;//vector containing values for recursion
    for (int i = 0; i <= N; i++) {//fill vector for amount of intervals
        J_of.push_back(1);
    }
    J_of[N] = 0;//end of vector equates to 0
    J_of[N-1] = 1;//second last term of vector equates to 2

    for (int n =N-1; n>0; n--) {//descending order for loop
        int m = n - 1, p = n+1;
        cout << J_of[p] <<"  "<<p <<" term"<<endl << J_of[n]<<"  "<<n << " term" <<endl;
        J_of[m] =  n*J_of[n]*2/x - J_of[p];//descending recursion relationship
        cout << J_of[m] <<"  "<< m << " term" << endl<<endl;

    }
    cout << endl;
```

```
    double c=0;//normalization constant

    for (int q = 2; q <= N; q=q+2) {//summation for normalization constant
        c =c + 2 * J_of[q];
    }
    c = c + J_of[0];//final addition for normalization constant
    cout <<"the normalization term is: "<< c<<endl<<endl;
    for (int p = 0; p <= N; p++) {
        J_of[p]= J_of[p]/c;//divide all element of vector by normalization constant
    }
    //output bessel function at each output
    cout << "Renormalized bessel function output at each order: "<<endl;
    for (int u = 0; u <= N; u++) {
        cout << u << " Order: " <<setprecision(15)<< J_of[u] << endl;
    }
}
```

7.3 Code Q3

```
// 4G03 Q3
#include <iostream>
#include<stdio.h>
#include <math.h>
#include <vector>

using namespace std;
/*function that preforms recursion relationship
to output 0th order of bessel function for given value of x.
Essentially the same code from part 2 except
x is a paramter of function*/
double bessel(double x){
    int N = 25;
    vector<double> J_of;
    for (int i = 0; i <= N; i++) {
        J_of.push_back(1);
    }
    J_of[N] = 0;
    J_of[N - 1] = 1;

    for (int n = N - 1; n > 0; n--) {
        int m = n - 1, p = n + 1;
        J_of[m] = n * J_of[n] * 2 / x - J_of[p];
    }
    double c = 0;

    for (int q = 2; q <= N; q = q + 2) {
        c = c + 2 * J_of[q];
    }
    c = c + J_of[0];
    for (int p = 0; p <= N; p++) {
        J_of[p] = J_of[p] / c;
    }



    return J_of[0];
}
/*function that preforms bisection method
given 2 points that bracket the root*/
double bisection(double a, double b) {
    double c;//midpoint
    if (bessel(a) * bessel(b) < 0) {//make sure y values of points different polarity
        for (int i = 0; i < 50; i++) {
            c = (a + b) / 2;//reseting midpoint
            if (bessel(c) == 0) {
                //if midpoint is root return value
                return c;
            }
            //if bessel at c and a are opposite sides, move b to c
            else if (bessel(c) * bessel(a) < 0) {
                b = c;
            }//if bessel at c and b are opposite sides, move a to c
            else {
                a = c;
```

```
            }
        }//output closest found value to root
        return c;
    }
    else {
        cout << "Invalid brackets";
        return 0;
    }
}

int main()
{
    //init variables to bisection function at found roots
    double root_1 = bisection(2, 3),
        root_2 = bisection(5, 6),
        root_3 = bisection(8, 9);
    //output difference of the roots
    cout <<"The first root is: "<< root_1<<endl;
    cout << "The secont root is: " << root_2 << endl;
    cout << "The third root is: " << root_3 << endl<<endl;
    cout << "The distance between the first 2 roots are: " << root_2 - root_1 << endl;
    cout << "The distance between the second and third roots are: " << root_3 -root_2 << endl;
}
```

7.4 Code Q4

```
//4G03 Q4
#include <iostream>
#include<stdio.h>
#include <math.h>
#include <ctime>
#include <vector>
#include<iomanip>

using namespace std;
//create function for sqrt(x) used for repeated fraction
double myfunc(double x)
{

    double r;
    r = sqrt(x);
    return r;
}
//create function to preform factorial for taylor series
double fact(double x) {
    double outcome=x;//set value as parameter
    if (x == 0) {
        outcome = 1;//factorial of 0
        return outcome;
    }
    else {
        /*for loop that starts
        @ value of parameter
        and continues to
        multiply by index minus 1
        till finally multiplied by 1*/
        for (double i = x; i > 1; i--) {
            outcome = outcome * (i - 1);
        }
        return outcome;//return factorial outcome
    }

}
/*taylor series function using amount of iterations and x value as a parameter*/
double Sq_Tlr(double x, double N) {
    //set value of power
    double n=0.5

        //multiplication factor cause by derivative
        ,constant=0.5

        //center of taylor series
        , a=x-1;
    double series=0;//initial series sum
    double function1//function holder
        ,function2= sqrt(a);//initial function

    /*loop that runs for amount of iterations
    applying derivative then adding to taylor series*/
    for (double k = 1; k <= N; k++) {
        function1 = function2 / (pow(a, k));//derivative
```

```
        //taylor series
        series=series+(constant*function1*pow((x-a),k))/fact(k);
        //new constant due to derivative
        constant = constant * (n - k);
    }
    //adding TO to the series
    series = series + function2;
    return series;//outputs approx solution
}
double Sq_Nwt(double x,double N)
{

    vector<double> xk;//vector for recursion relation

    //fills vector with amount of index
    for (double i = 0; i <= N; i++) {
        xk.push_back(i + 1);
    }
    //derivative x point deviation value
    double h = 0.00000001;

    //for loop preforming newton approx relationships
    for (double k = 1; k <= N; k++) {
        double m = k - 1;
        //rearrangement of formula to solve for input of x
        double func = pow(xk[m], 2) - x;
        //derivative of function
        double diff = ((pow(xk[m] + h, 2) - x) - (pow(xk[m] - h, 2) - x)) / (2 * h);
        //newtons recursion relationship
        xk[k] = xk[m] - func / diff;
    }
    //output approx solution
    return xk[N];
}
//repeated fractions function that uses function as pointer
//repeated fractions relation is not unique
//this function is absolute and requires less algebra
double Sq_Frc(double (*f)(double x), double x, double N)
{
    double a0, approx,num;
    //initial term
    a0 = floor(f(x));

    //numerator of repeated fraction
    num = x - pow(a0, 2);

    //bottom fraction
    approx = num/(2*a0);

    //for loop that continues the fraction
    for (int i = 0; i <= N; i++) {
        /*takes bottom fraction, adds 2*a0 and
        makes it denominator to numerator repeatedly*/
        approx = pow(approx + 2 * a0, -1) * num;
    }
    approx = approx + a0;//adds a0 term
```

```
    return approx;//outputs approx solution
}

int main()
{
    //variables set as parameters
    double x = 5, N = 10;
    //used to measure clock time
    /*output all 3 methods of approximations
    and calculate times taken and relative error*/
    cout << "The x term for the functions is "
    << x <<" and amount of iterations is "<<N <<endl<<endl;
    clock_t begin_time = clock();

    cout << "Square roots in taylors methods are: +-"
    <<setprecision(15) << Sq_Tlr(x, N) << "   "
    << "Exact value: "
    << setprecision(15) << myfunc(x) << endl;

    cout << "Relative error of taylor series is: "
    << (Sq_Tlr(x, N) - myfunc(x)) / myfunc(x) << endl;

    clock_t end_time = clock();

    cout <<"Time: "<< setprecision(5)
    << double(end_time - begin_time)/ CLOCKS_PER_SEC << " s" <<endl<<endl;


    clock_t begin_time2 = clock();

    cout << "Square roots in newtons methods are: +-"
    <<setprecision(15) << Sq_Nwt(x,N)<<"   "
    <<"Exact value: "<< setprecision(15) << myfunc(x) <<endl;

    cout << "Relative error of Newtons method is: "
    << (Sq_Nwt(x, N) - myfunc(x)) / myfunc(x) << endl;

    clock_t end_time2 = clock();

    cout << "Time: " << setprecision(5) <<double(end_time2 - begin_time2) / CLOCKS_PER_SEC << " s" <<

    clock_t begin_time3 = clock();

    cout << "Square roots in repeated fractions are: +-" << setprecision(15) << Sq_Frc(&myfunc,x, N)
    << "Exact value: " << setprecision(15) << myfunc(x) <<endl;

    cout << "Relative error of Newtons method is: "
    << (Sq_Frc(&myfunc, x, N) - myfunc(x)) / myfunc(x) << endl;

    clock_t end_time3 = clock();
    cout << "Time: " << setprecision(5)
    << double(end_time3 - begin_time3) / CLOCKS_PER_SEC
    <<" s"<< endl << endl;
    return 0;
}
```

7.5 Code Q5

```
//4G03 Q5
#include <iostream>
#include <math.h>
#include <iomanip>
#include <ctime>
#include <vector>
using namespace std;
//initialize function being used
double myfunc(double x)
{
    double r;
    r = 4.0 / (1 + x * x);
    return r;
}
//trapezoidal function to solve integral
//function, bounds and interval amount are parameters
double Trapezoidal(double (*f)(double x), double xl, double xh, int inl)
{
    double each_interval = (xh - xl) / inl;//deviance for each interval
    double integration = f(xl) + f(xh);//first and last term of sum
    double c;//each x point for interval

    for (double i = 1; i < inl; i++)
    //run through loop and add 2 times the y value at each interval for summation
    {
        c = xl + i * each_interval;
        integration = integration + 2 * (f(c));
    }
    //area under the curve using found values
    integration = integration * each_interval / 2;
    return integration;
}
double Romberg(double (*f)(double x), double xl, double xh, double N)
{
    vector<vector<double>> r;//2D vector
    vector<double> h;//vector for varying h values

    double a = (N) + 1;
    //for loop to fill vectors
    for (double q = 0; q < a; q++) {
        h.push_back(1);
        vector<double>v;
        for (double o = 0; o < a; o++) {
            v.push_back(1);
        }
        r.push_back(v);
    }
    //solution for each h term
    for (int i = 1; i <= N; i++)
    {
        h[i] = (xh - xl) / pow(2, i - 1);
    }
    //initial romberg value
    /*keep in mind initialized index starts
    at 1,1 and not 0,0 so remember for final plug in*/
```

```
    r[1][1] = h[1] / 2 * (f(xl) + f(xh));
    //for loop to solve for first variable of romberg fuction
    //ie. all R(i,0)
    for (int i = 2; i <= N; i++)
    {
        double coeff = 0;
        // summation for loop for recursion relationship
        for (double k = 1; k <= pow(2, i - 2); ++k)
        {
            coeff += f(xl + (2 * k - 1) * h[i]);
        }
        //solution to all R(i,0) for each index of for loop
        r[i][1] = 0.5 * (r[i - 1][1]) + h[i] * coeff;
    }

    //final recusion relationship using previous R(i,0) values
    /*solving for R(i,j), first running through
    all j vaues for each i and then increase i and repeat
    til i=j*/
    for (int i = 2; i <= N; i++)
    {
        for (int j = 2; j <= i; j++)
        {
            r[i][j] = r[i][j - 1] + (r[i][j - 1] - r[i - 1][j - 1]) / (pow(4, j - 1) - 1);
        }
    }
    //output solution to integral
    return r[N][N];
}

int main()
{
    double pi = 2 * acos(0.0);

    //exact val of pi
    cout << "Exact value of pi: " << setprecision(15) << pi<<endl<<endl;

    //output precise value for trapzedoial integration,
    //measure time taken and relative erro
    clock_t begin_time = clock();
    cout << "The Trapezoidal result is " << setprecision(15)
    << Trapezoidal(&myfunc, 0, 1, 64) << endl;
    cout<<"Relative error of Trapezoidal method is: "
    <<setprecision(15)
    << (Trapezoidal(&myfunc, 0, 1, 64)-pi)/pi << endl;
    clock_t end_time = clock();
    cout << "Time: " << setprecision(5) << double(end_time - begin_time) / CLOCKS_PER_SEC
    << " s" << endl << endl;

    //output precise value for romberg integration and measure time taken
    clock_t begin_time2 = clock();
    cout<<"The romberg result is "<<setprecision(15)<<Romberg(&myfunc, 0, 1, 10)<<endl;
    cout << "Relative error of Romberg method is: " << setprecision(15)
    << (Romberg(&myfunc, 0, 1, 10) - pi) / pi << endl;
    clock_t end_time2 = clock();
    cout << "Time: " << setprecision(5)
```

```
    << double(end_time2 - begin_time2) / CLOCKS_PER_SEC << " s" << endl << endl;
}
```