CMSC 455/655 Homework 1
Dr. Tyler Simon Fall 2024

Name: Varun Kesharaju

Date: 9/19/24

Please upload this document with your solutions as a PDF to blackboard.

Also upload any code that you wrote. Copy and paste code within the problem to show your methodology. Please comment appropriately. For instance, if you write two lines of code to calculate n!, then copy and paste those lines of code. Highlight and show the answer.

If you used source code (and you can for problems where it does not specify to write code explicitly), please cite the source code according to best practices.

Example: If you used the SPICEYPY library from NAIF,

Acton, C.H.; "Ancillary Data Services of NASA's Navigation and Ancillary Information Facility;" Planetary and Space Science, Vol. 44, No. 1, pp. 65-70, 1996.
DOI 10.1016/0032-0633(95)00107-7.

If you use my code, reference the website where I posted it.

---

Question 1: (10 points) Not all numbers are represented in base 2, and that is why we need to worry about floating point representation. Write the following programs and provide source code examples in your answer.

a) (3 points) Write a program to output the base 2 binary string of a decimal number with n digits using division and multiplication methods. Do not use built-in conversion functions. Ex: Input 3.14 will produce the binary string 11.0100011110

b) (2 points) Convert 38.1 out to 10 digits. Print the output.

c) (3 points) Write another program to convert a string binary number to floating point without using built-in functions. Using the 10-digit binary value for 0.1, convert back to base 10. What is the new value? Use as many digits as possible. What is the value using 52 digits in the conversion from base 10 to base 2?

d) (2 points) What is the relative and absolute error of the true value, 0.1, with the 10-digit rounded approximation? Repeat for 20 digits.

Ans:

a) Base 2 binary string of a decimal number with n digits using divide and multiple method, up-to 6 digit precision:
Code:

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

void wholeToBinary(int n, int precision){
   int binary[precision], i=0;
   while(n>0){
       binary[i] = n%2;
       i+=1;
       n = n/2;
   }
   for(int j=i-1; j>=0; j--){
       printf("%d", binary[j]);
   }
}

void DeciToBinary(double n, int precision){
   if (n>0){
       printf(".");
   }
   int p = precision;
   while(n>0 && p--){
       printf("%d", (int)(n*2));
       n = n*2 - (int)(n*2);
   }
   printf("\n");
}

void floatToBinary(int precision){
   double a=0,w=0,deci=0;
   /*
   Double Vs Float:
   A variable of type float only has 7 digits of precision whereas a
variable of type double has 15 digits of precision.
   double precision numbers are represented by 64 bits, while float
precision numbers are represented by 32 bits.
```

```
   */
   printf("Enter a decimal number with no more than 6 decimal
digits: \n");
   scanf("%lf", &a);

   deci = modf(a, &w); // modf returns the fractional part of a in d
upto six decimals and the integer part in w
   int intpart = (int)w;

   wholeToBinary(intpart, precision);
   DeciToBinary(deci, precision);
}

int main(int argc, char *argv[])
{
   int precision = argc > 1 ? atoi(argv[1]) : 52;
   printf("Precision: %d\n", precision);
   floatToBinary(precision);
   return 0;

}
```

Output:

```
(base) ➜  assignment1 gcc FloatToBinary.c -o ./executables/FloatToBinary -lm && ./executables/FloatToBinary
Precision: 52
Enter a decimal number with no more than 6 decimal digits:
3.14
11.00100011110101110000101000111101011100001010001111
```

b) Output:

```
(base) ➜  assignment1 gcc FloatToBinary.c -o ./executables/FloatToBinary -lm && ./executables/FloatToBinary 10
Precision: 10
Enter a decimal number with no more than 6 decimal digits:
38.1
100110.0001100110
```

c) Program to convert a string binary number to floating point:

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

void binaryToBase10(){
   char binary[104];
   int i=0;
   double intPart=0;

   printf("Enter a binary number: \n");
```

```
    scanf("%s", binary);

    while(binary[i]!='\0' && binary[i]!='.'){
        i+=1;
    }

    for (int j=0; j<i; j++){
        intPart += (((int)binary[j]-48)==1 ? pow(2, i-j-1): 0);
    }

    double deciPart=0;
    for (int j=i+1; binary[j]!='\0'; j++){
        deciPart += ((int)binary[j]-48==1 ? 1/pow(2, j-i-1): 0);
    }

    printf("The decimal equivalent is: \n%lf\n", intPart+deciPart);
}

int main()
{

    binaryToBase10();

    return 0;
}
```

Bash script to use the above code executables and convert 0.1 base 10, to 10 digit binary value, and back to base 10 number.

```
#!/bin/bash

set -e

gcc FloatToBinary.c -o ./executables/FloatToBinary -lm
gcc BinaryStringToBase10.c -o ./executables/BinaryStringToBase10 -lm
gcc Errors.c -o ./executables/Errors -lm

precision=${1:-54}
echo "Precision digits: $precision"
```

```
./executables/FloatToBinary "$precision" < FloatToBinaryinput.txt >
FloatToBinaryOutput.txt
head -n 3 FloatToBinaryOutput.txt | tail -n 1 |
./executables/BinaryStringToBase10 > BinaryStringToBase10Output.txt

input1=$(head -n 3 FloatToBinaryOutput.txt | tail -n 1)
input2=$(head -n 3 BinaryStringToBase10Output.txt | tail -n 1)

(echo "$input1"; echo "$input2") | ./executables/Errors
```

Input file content:
0.1

Output:

```
(base) ➜  assignment1 bash errors.sh 10
Precision digits: 10
The binary equivalent of 0.1 is: .0001100110
The decimal equivalent of .0001100110 is: 0.199219
```

```
(base) ➜  assignment1 bash errors.sh
Precision digits: 54
The binary equivalent of 0.1 is: .000110011001100110011001100110011001100110011001100110
The decimal equivalent of .000110011001100110011001100110011001100110011001100110 is: 0.200000
```

d) Code to calculate absolute and relative errors:

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

int main(){

   double num = 0.1 , decimal_num;
   char binary_num[104];

   scanf("%s", binary_num);
   scanf("%lf", &decimal_num);

   printf("The binary equivalent of %0.1lf is: %s\n", num,
binary_num);
   printf("The decimal equivalent of %s is: %lf\n", binary_num,
decimal_num);

   printf("True value of the number in context is: %0.1lf\n", 0.1);
   printf("Approximate value of the number in context is: %lf\n",
decimal_num);
```

```
   printf("Absolute error: %lf\n", (decimal_num- 0.1)<0 ?
(-1.0)*(decimal_num- 0.1): (decimal_num- 0.1));
   printf("Relative error: %lf\n", ((decimal_num- 0.1)<0 ?
(-1.0)*(decimal_num- 0.1): (decimal_num- 0.1))/0.1);


   return 0;
}
```

Using the bash script from part c,
Output:

```
(base) ➜  assignment1 bash errors.sh 10
Precision digits: 10
The binary equivalent of 0.1 is: .0001100110
The decimal equivalent of .0001100110 is: 0.199219
True value of the number in context is: 0.1
Approximate value of the number in context is: 0.199219
Absolute error: 0.099219
Relative error: 0.992190
```

```
(base) ➜  assignment1 bash errors.sh 20
Precision digits: 20
The binary equivalent of 0.1 is: .00011001100110011001
The decimal equivalent of .00011001100110011001 is: 0.199999
True value of the number in context is: 0.1
Approximate value of the number in context is: 0.199999
Absolute error: 0.099999
Relative error: 0.999990
```

Question 2: (10 points) Write a program to add 0.001 to itself 1000 times. The exact analytical answer would be 1.0. What is the:

a) (2 points) Absolute error in your result with the true value?

b) (2 points) The relative error in your result with the true value?

c) (2 points) Why does the computer differ from the exact value?

d) (2 points) Using the code you developed in question 1, convert 0.1|10 to base-2 using 5 digits. Convert back, and then add that value to itself 10 times. Adding 0.1 to itself 10 times should equal the value of 1. What is the actual value? The absolute error?

e) (2 points) For the number of digits ranging from 5 to 100, plot the relative error in the calculated value and the true value. Place the plot below.

Ans:

Code to add 0.001 to itself for 1000 times:

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

int main(int argc, char *argv[]){
   double num = argc > 1 ? atof(argv[1]) : 0.001, sum = 0.0;

   int numIterations = argc > 2 ? atoi(argv[2]) : 1000;
   for(int i=0;i<numIterations;i++){
       sum += num;
   }

   double trueValue = 1.0;

   printf("Approximate Value: %2.64g\n", sum);
   printf("Absolute Error: %2.64g\n", (trueValue - sum)<0 ?
(-1.0)*(trueValue - sum): (trueValue - sum));
   printf("Relative Error: %2.64g\n", ((trueValue - sum)<0 ?
(-1.0)*(trueValue - sum): (trueValue - sum))/trueValue);

   /*
   Why does the computer differ from the true value?
   */

   return 0;
}
```

a) Output:

```
(base) ➜  assignment1 gcc contAddition.c -o executables/contAddition -lm && ./executables/contAddition
Approximate Value: 1.0000000000000006661338147750939242541790008544921875
Absolute Error: 6.6613381477509392425417900085449218750e-16
Relative Error: 6.6613381477509392425417900085449218750e-16
```

b) Above output contains both absolute and relative errors.

c) The computer uses binary representation to store the numbers. The number 0.001 is not exactly representable in binary. The computer stores the number as 0.001000000000000000020816681711721685132943093776702880859375. When we add 0.001 1000 times, the error accumulates and the number stored in the computer is different from the true value.

d) Bash script to simulate the process:

```bash
#!/bin/bash

set -e
```

```
gcc FloatToBinary.c -o ./executables/FloatToBinary -lm
gcc BinaryStringToBase10.c -o ./executables/BinaryStringToBase10 -lm
gcc contAddition.c -o ./executables/contAddition -lm

precision=${1:-54}
echo "Precision digits: $precision"

numItr=${2:-10}

echo "nu of iterations : $numItr"

./executables/FloatToBinary "$precision" < FloatToBinaryinput.txt >
FloatToBinaryOutput.txt
head -n 3 FloatToBinaryOutput.txt | tail -n 1 |
./executables/BinaryStringToBase10 > BinaryStringToBase10Output.txt

num=$(head -n 3 BinaryStringToBase10Output.txt | tail -n 1)

./executables/contAddition "$num" "$numItr"
```

Input file content:
0.1

Output:

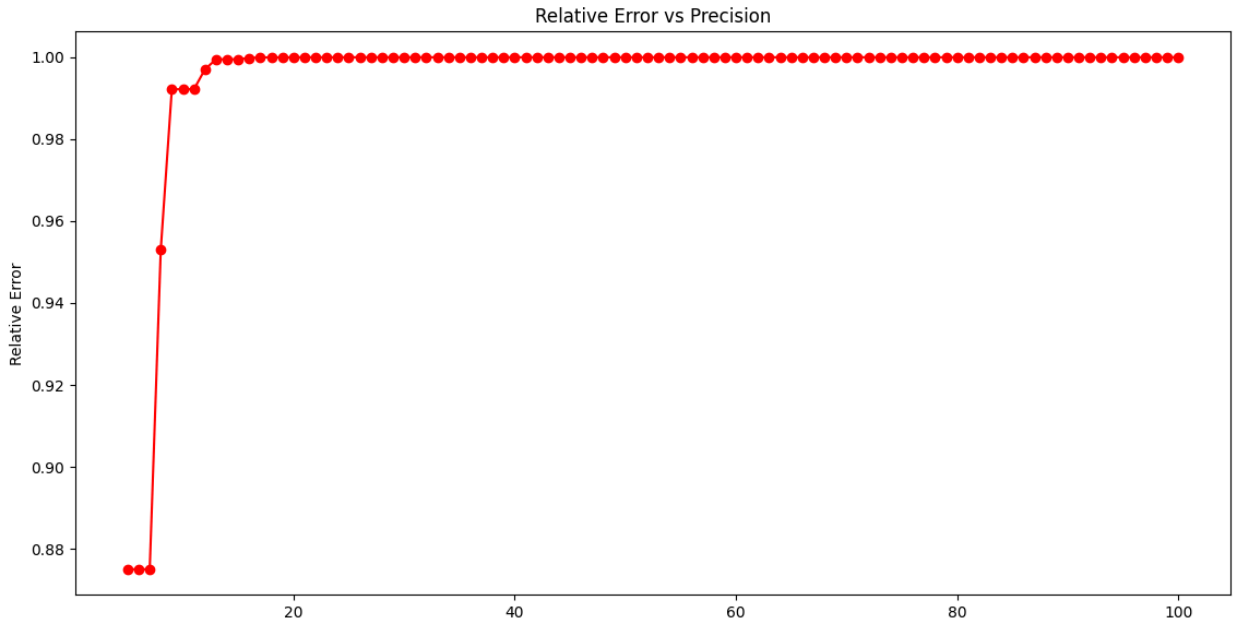
```
(base) ➜  assignment1 bash additionErrors.sh 5
Precision digits: 5
nu of iterations : 10
Approximate Value: 1.875000
Absolute Error: 0.875000
Relative Error: 0.875000
```

e) Python script to plot the relative error in the calculated value and the true value for the number of digits ranging from 5 to 100:

```
import subprocess
import matplotlib.pyplot as plt

relative_error = []

for precision in range(5, 101):
   output = subprocess.run(['./additionErrors.sh' , f'{precision}'],
capture_output=True, text=True)
```

```
    relative_error.append(float(
        (output.stdout.split('\n')[4]).split(' ')[-1]
    ))

plt.plot(range(5, 101), relative_error, 'ro-')
plt.xlabel('Precision')
plt.ylabel('Relative Error')
plt.title('Relative Error vs Precision')
plt.show()
```

Output:




Question 3: (10 points) Horner's method reduces the number of operations necessary for evaluating n-degree polynomials. At large values of n, Horner's method, while fast, can produce large absolute errors. Write a program that uses Horner's method to evaluate any polynomial. Then for the following:

a) (3 points) Evaluate the polynomial 4x5 + 7x4 + 8x3 -20x2 -5x + 10 normally and with Horner's method over the interval of [-2,2]. Plot the results using both. Label all results and axes.

b) (4 points) Evaluate the polynomial x9 -7x7 +12x5 + 90x4 + 4x3 -2x2 +8x + 10 normally and with Horner's method. Plot the results using both. Label all results and axes.

c) (3 points) Given the two polynomials, is there any difference between their output using Horner's method vs the normal method? If so, what is the absolute error? And the relative error? Plot the results using both. Label all results and axes.

Ans:

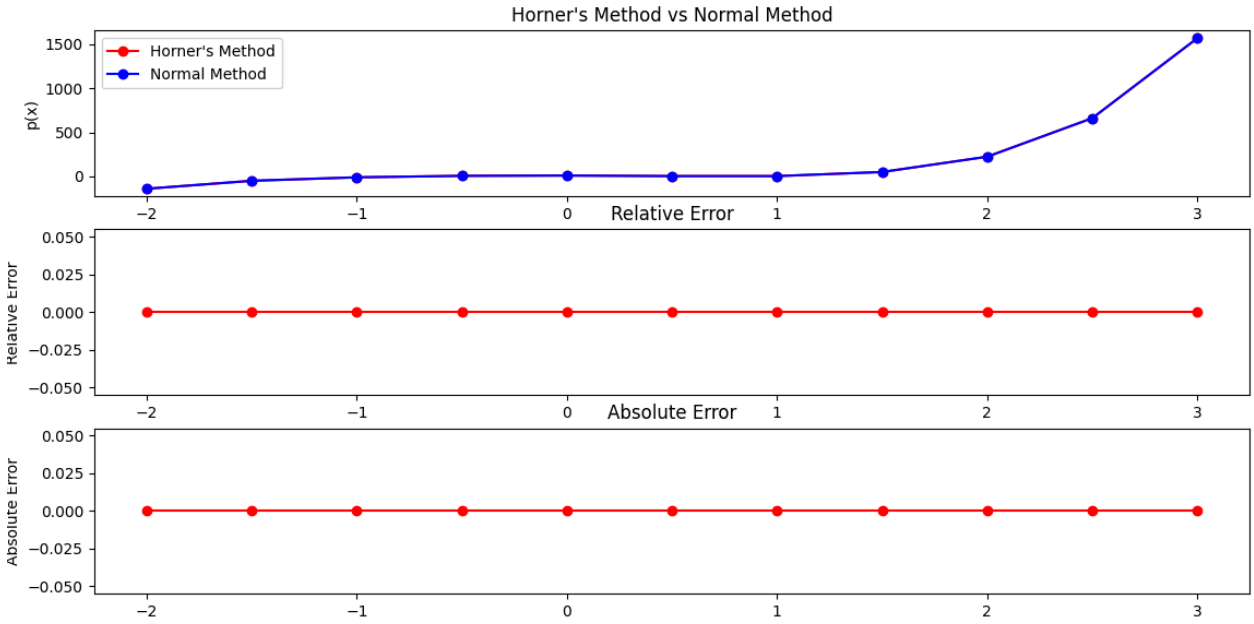
a) Code to evaluate the polynomial normally and using Horners Method:

```
import matplotlib.pyplot as plt
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-N', '--degrees', type=str, help='Degrees of
the polynomial')
parser.add_argument('-P', '--poly', type=str, help='List of
coefficients of the polynomial')
parser.add_argument('-Xs', '--xstart', type=float, help='Value of x
start')
parser.add_argument('-Xe', '--xend', type=float, help='Value of x
end')

args = parser.parse_args()

poly_dict = {}
if args.poly and args.degrees:
   for key,val in zip(args.degrees.split(' '), args.poly.split('
')):
       poly_dict[int(key)] = int(val)
else:
   poly_dict = {5: 4, 4: 7, 3: 8, 2: -20, 1: -5, 0: 10}

def horner(poly_dict, x, poly_degrees):

   # Initialize result
   result = poly_dict[poly_degrees[0]]
   cur_exp = poly_degrees[0]
    # Evaluate value of polynomial
   # using Horner's method
   for i in poly_degrees[1:]:
       gap = cur_exp - i
       result = result * (x**gap) + poly_dict[i]
       cur_exp = i
```

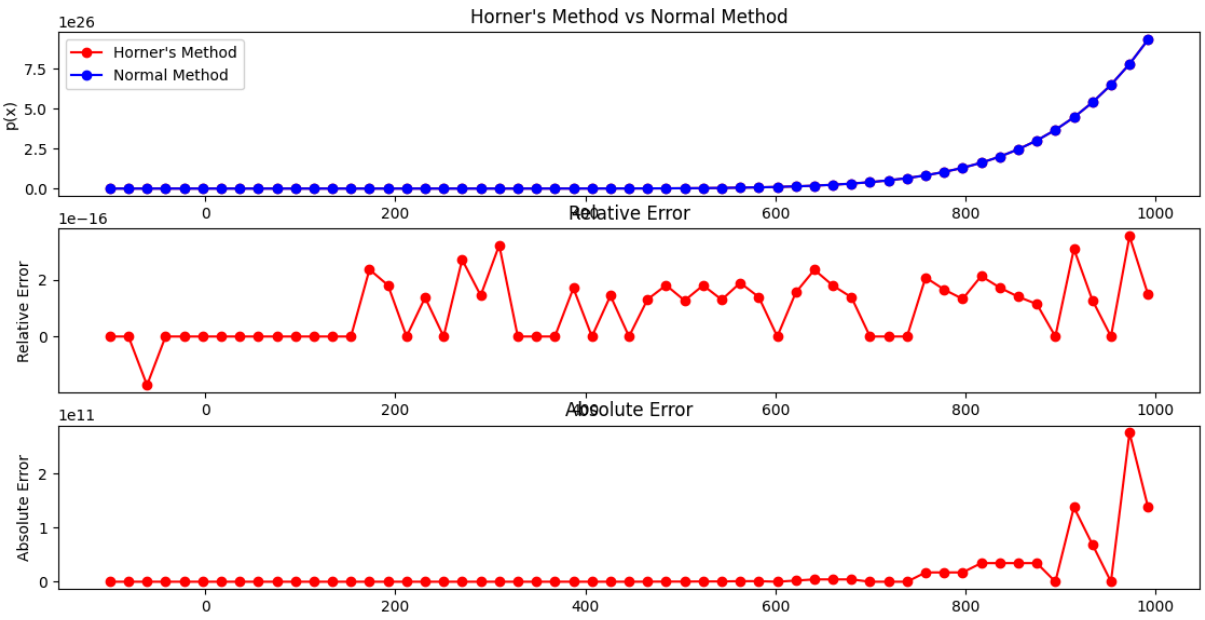
```
    return result

def poly_eval(poly_dict, x, poly_degrees):
   # Declaring the result
   result = 0

   # Running a for loop to traverse through the list
   for i in poly_degrees:

       # Declaring the variable Sum
       Sum = poly_dict[i]

       # Running a for loop to multiply x (n-i-1)
       # times to the current coefficient
       for j in range(0,i):
           Sum = Sum * x

       # Adding the sum to the result
       result = result + Sum

   return result

n = len(poly_dict)
poly_degrees =  sorted(poly_dict.keys(), reverse=True)
x_range, normal_arr, horner_arr = [], [], []
x = -2 if args.xstart is None else args.xstart
x_end = 3 if args.xend is None else args.xend
while x<=x_end:
   x_range.append(x)
   horner_arr.append(horner(poly_dict, x, poly_degrees))
   normal_arr.append(poly_eval(poly_dict, x, poly_degrees))
   x+=(0.5 if args.xend is None else 19.5)

abs_error = [abs(horner_arr[i]-normal_arr[i]) for i in
range(len(horner_arr))]
relative_error = [abs_error[i]/normal_arr[i] for i in
range(len(horner_arr))]

fig, axs = plt.subplots(3,1 , figsize=(10, 10))
axs[0].plot(x_range, horner_arr, 'ro-', label='Horner\'s Method')
```

```
axs[0].plot(x_range, normal_arr, 'bo-', label='Normal Method')
axs[0].set_xlabel('x')
axs[0].set_ylabel('p(x)')
axs[0].set_title('Horner\'s Method vs Normal Method')
axs[0].legend()

axs[1].plot(x_range, relative_error, 'ro-', label='Relative Error')
axs[1].set_xlabel('x')
axs[1].set_ylabel('Relative Error')
axs[1].set_title('Relative Error')

axs[2].plot(x_range, abs_error, 'ro-', label='Absolute Error')
axs[2].set_xlabel('x')
axs[2].set_ylabel('Absolute Error')
axs[2].set_title('Absolute Error')

plt.show()
```

Output:




b) Output:




c) Errors are plotted in part a and b.

Question 4: (10 points) On your computer, what are the largest and smallest numbers for double and single precision? What are the respective gaps between the smallest numbers and zero?

Ans:

Code to calculate smallest double and single precision numbers:

```c
/* epsilon.c  study smallest number added to 1.0 that is not 1.0 */

#include <stdio.h>
double dstore(double tmp);
float store(float tmp);
void sub(float *temp); /* extreme paranoia */

int main()
{
 float eps = 1.0f;
 float temp;
 double deps = 1.0;
 int i;

 printf("epsilon.c running \n");
 while(1.0+deps>1.0) deps = deps/2.0;
 deps = 2.0*deps; /* went one too many */
 printf("no store double deps=%e \n", deps);
```

```
 deps = 1.0;
 while(dstore(1.0+deps)>1.0) deps = deps/2.0;
 deps = 2.0*deps; /* went one too many */
 printf("forced store double deps=%e \n", deps);
 printf("the difference is because of more precision when kept in
registers\n");
 printf("Size of double deps: %lu bytes\n", sizeof(deps));

 while(1.0+eps>1.0) eps = eps/2.0;
 eps = 2.0*eps; /* went one too many */
 printf("\nno store float eps=%e \n", eps);

 eps = 1.0f;
 while(store(1.0f+eps)>1.0f) eps = eps/2.0f;
 eps = 2.0f*eps; /* went one too many */
 printf("forced store float eps=%e \n", eps);
 printf("the difference is because of more precision when kept in
registers\n");
 printf("Size of float eps: %lu bytes\n", sizeof(eps));

 printf("\nNow float eps, note double when kept in registers \n");
 for(i=1; i<130; i++){
   eps = eps/2.0f;
   temp = (1.0f+eps)-1.0f;
   printf("eps=2^-%d= %e, (1+eps)-1= %e \n", i, eps, temp);
 }
 printf("\nforce store, break optimization \n");
 eps = 1.0f;
 for(i=1; i<155; i++){
   eps = eps/2.0f;
   temp = 1.0f+eps;
   sub(&temp);
   printf("eps=2^-%d= %e, (1+eps)-1= %e \n", i, eps, temp);
 }
 printf("\nend epsilon.c\n");
 return 0;
}

double dstore(double tmp)
{
```

```
 return tmp;
}

float store(float tmp)
{
 return tmp;
}

void sub(float *temp)
{
 *temp = *temp - 1.0f;
}
```

Output:
Smallest number for single precision is:
forced store float eps=1.192093e-07
Smallest number for double precision is:
2^-126= 1.401298e-45

Source Code Reference:
Dr. Tyler Simon, https://userpages.cs.umbc.edu/tsimo1/CMSC455/code/epsilon.c

Question 5: (5 points) Copy and paste output showing the epsilon of your machine. State the coding language you are using.

Ans:
Output:

```
Unset
(base) ➜  assignment1 gcc machineEpsilon.c -o executables/machineEpsilon -lm &&
executables/machineEpsilon
epsilon.c running
no store double deps=2.220446e-16
forced store double deps=2.220446e-16
the difference is because of more precision when kept in registers
Size of double deps: 8 bytes

no store float eps=2.220446e-16
forced store float eps=1.192093e-07
the difference is because of more precision when kept in registers
Size of float eps: 4 bytes
```

```
Now float eps, note double when kept in registers
eps=2^-1= 5.960464e-08, (1+eps)-1= 0.000000e+00
eps=2^-2= 2.980232e-08, (1+eps)-1= 0.000000e+00
eps=2^-3= 1.490116e-08, (1+eps)-1= 0.000000e+00
eps=2^-4= 7.450581e-09, (1+eps)-1= 0.000000e+00
eps=2^-5= 3.725290e-09, (1+eps)-1= 0.000000e+00
eps=2^-6= 1.862645e-09, (1+eps)-1= 0.000000e+00
eps=2^-7= 9.313226e-10, (1+eps)-1= 0.000000e+00
eps=2^-8= 4.656613e-10, (1+eps)-1= 0.000000e+00
eps=2^-9= 2.328306e-10, (1+eps)-1= 0.000000e+00
eps=2^-10= 1.164153e-10, (1+eps)-1= 0.000000e+00
eps=2^-11= 5.820766e-11, (1+eps)-1= 0.000000e+00
eps=2^-12= 2.910383e-11, (1+eps)-1= 0.000000e+00
eps=2^-13= 1.455192e-11, (1+eps)-1= 0.000000e+00
eps=2^-14= 7.275958e-12, (1+eps)-1= 0.000000e+00
eps=2^-15= 3.637979e-12, (1+eps)-1= 0.000000e+00
eps=2^-16= 1.818989e-12, (1+eps)-1= 0.000000e+00
eps=2^-17= 9.094947e-13, (1+eps)-1= 0.000000e+00
eps=2^-18= 4.547474e-13, (1+eps)-1= 0.000000e+00
eps=2^-19= 2.273737e-13, (1+eps)-1= 0.000000e+00
eps=2^-20= 1.136868e-13, (1+eps)-1= 0.000000e+00
eps=2^-21= 5.684342e-14, (1+eps)-1= 0.000000e+00
eps=2^-22= 2.842171e-14, (1+eps)-1= 0.000000e+00
eps=2^-23= 1.421085e-14, (1+eps)-1= 0.000000e+00
eps=2^-24= 7.105427e-15, (1+eps)-1= 0.000000e+00
eps=2^-25= 3.552714e-15, (1+eps)-1= 0.000000e+00
eps=2^-26= 1.776357e-15, (1+eps)-1= 0.000000e+00
eps=2^-27= 8.881784e-16, (1+eps)-1= 0.000000e+00
eps=2^-28= 4.440892e-16, (1+eps)-1= 0.000000e+00
eps=2^-29= 2.220446e-16, (1+eps)-1= 0.000000e+00
eps=2^-30= 1.110223e-16, (1+eps)-1= 0.000000e+00
eps=2^-31= 5.551115e-17, (1+eps)-1= 0.000000e+00
eps=2^-32= 2.775558e-17, (1+eps)-1= 0.000000e+00
eps=2^-33= 1.387779e-17, (1+eps)-1= 0.000000e+00
eps=2^-34= 6.938894e-18, (1+eps)-1= 0.000000e+00
eps=2^-35= 3.469447e-18, (1+eps)-1= 0.000000e+00
eps=2^-36= 1.734723e-18, (1+eps)-1= 0.000000e+00
eps=2^-37= 8.673617e-19, (1+eps)-1= 0.000000e+00
eps=2^-38= 4.336809e-19, (1+eps)-1= 0.000000e+00
eps=2^-39= 2.168404e-19, (1+eps)-1= 0.000000e+00
eps=2^-40= 1.084202e-19, (1+eps)-1= 0.000000e+00
eps=2^-41= 5.421011e-20, (1+eps)-1= 0.000000e+00
eps=2^-42= 2.710505e-20, (1+eps)-1= 0.000000e+00
```

```
eps=2^-43= 1.355253e-20, (1+eps)-1= 0.000000e+00
eps=2^-44= 6.776264e-21, (1+eps)-1= 0.000000e+00
eps=2^-45= 3.388132e-21, (1+eps)-1= 0.000000e+00
eps=2^-46= 1.694066e-21, (1+eps)-1= 0.000000e+00
eps=2^-47= 8.470329e-22, (1+eps)-1= 0.000000e+00
eps=2^-48= 4.235165e-22, (1+eps)-1= 0.000000e+00
eps=2^-49= 2.117582e-22, (1+eps)-1= 0.000000e+00
eps=2^-50= 1.058791e-22, (1+eps)-1= 0.000000e+00
eps=2^-51= 5.293956e-23, (1+eps)-1= 0.000000e+00
eps=2^-52= 2.646978e-23, (1+eps)-1= 0.000000e+00
eps=2^-53= 1.323489e-23, (1+eps)-1= 0.000000e+00
eps=2^-54= 6.617445e-24, (1+eps)-1= 0.000000e+00
eps=2^-55= 3.308722e-24, (1+eps)-1= 0.000000e+00
eps=2^-56= 1.654361e-24, (1+eps)-1= 0.000000e+00
eps=2^-57= 8.271806e-25, (1+eps)-1= 0.000000e+00
eps=2^-58= 4.135903e-25, (1+eps)-1= 0.000000e+00
eps=2^-59= 2.067952e-25, (1+eps)-1= 0.000000e+00
eps=2^-60= 1.033976e-25, (1+eps)-1= 0.000000e+00
eps=2^-61= 5.169879e-26, (1+eps)-1= 0.000000e+00
eps=2^-62= 2.584939e-26, (1+eps)-1= 0.000000e+00
eps=2^-63= 1.292470e-26, (1+eps)-1= 0.000000e+00
eps=2^-64= 6.462349e-27, (1+eps)-1= 0.000000e+00
eps=2^-65= 3.231174e-27, (1+eps)-1= 0.000000e+00
eps=2^-66= 1.615587e-27, (1+eps)-1= 0.000000e+00
eps=2^-67= 8.077936e-28, (1+eps)-1= 0.000000e+00
eps=2^-68= 4.038968e-28, (1+eps)-1= 0.000000e+00
eps=2^-69= 2.019484e-28, (1+eps)-1= 0.000000e+00
eps=2^-70= 1.009742e-28, (1+eps)-1= 0.000000e+00
eps=2^-71= 5.048710e-29, (1+eps)-1= 0.000000e+00
eps=2^-72= 2.524355e-29, (1+eps)-1= 0.000000e+00
eps=2^-73= 1.262177e-29, (1+eps)-1= 0.000000e+00
eps=2^-74= 6.310887e-30, (1+eps)-1= 0.000000e+00
eps=2^-75= 3.155444e-30, (1+eps)-1= 0.000000e+00
eps=2^-76= 1.577722e-30, (1+eps)-1= 0.000000e+00
eps=2^-77= 7.888609e-31, (1+eps)-1= 0.000000e+00
eps=2^-78= 3.944305e-31, (1+eps)-1= 0.000000e+00
eps=2^-79= 1.972152e-31, (1+eps)-1= 0.000000e+00
eps=2^-80= 9.860761e-32, (1+eps)-1= 0.000000e+00
eps=2^-81= 4.930381e-32, (1+eps)-1= 0.000000e+00
eps=2^-82= 2.465190e-32, (1+eps)-1= 0.000000e+00
eps=2^-83= 1.232595e-32, (1+eps)-1= 0.000000e+00
eps=2^-84= 6.162976e-33, (1+eps)-1= 0.000000e+00
eps=2^-85= 3.081488e-33, (1+eps)-1= 0.000000e+00
eps=2^-86= 1.540744e-33, (1+eps)-1= 0.000000e+00
```

```
eps=2^-87= 7.703720e-34, (1+eps)-1= 0.000000e+00
eps=2^-88= 3.851860e-34, (1+eps)-1= 0.000000e+00
eps=2^-89= 1.925930e-34, (1+eps)-1= 0.000000e+00
eps=2^-90= 9.629650e-35, (1+eps)-1= 0.000000e+00
eps=2^-91= 4.814825e-35, (1+eps)-1= 0.000000e+00
eps=2^-92= 2.407412e-35, (1+eps)-1= 0.000000e+00
eps=2^-93= 1.203706e-35, (1+eps)-1= 0.000000e+00
eps=2^-94= 6.018531e-36, (1+eps)-1= 0.000000e+00
eps=2^-95= 3.009266e-36, (1+eps)-1= 0.000000e+00
eps=2^-96= 1.504633e-36, (1+eps)-1= 0.000000e+00
eps=2^-97= 7.523164e-37, (1+eps)-1= 0.000000e+00
eps=2^-98= 3.761582e-37, (1+eps)-1= 0.000000e+00
eps=2^-99= 1.880791e-37, (1+eps)-1= 0.000000e+00
eps=2^-100= 9.403955e-38, (1+eps)-1= 0.000000e+00
eps=2^-101= 4.701977e-38, (1+eps)-1= 0.000000e+00
eps=2^-102= 2.350989e-38, (1+eps)-1= 0.000000e+00
eps=2^-103= 1.175494e-38, (1+eps)-1= 0.000000e+00
eps=2^-104= 5.877472e-39, (1+eps)-1= 0.000000e+00
eps=2^-105= 2.938736e-39, (1+eps)-1= 0.000000e+00
eps=2^-106= 1.469368e-39, (1+eps)-1= 0.000000e+00
eps=2^-107= 7.346840e-40, (1+eps)-1= 0.000000e+00
eps=2^-108= 3.673420e-40, (1+eps)-1= 0.000000e+00
eps=2^-109= 1.836710e-40, (1+eps)-1= 0.000000e+00
eps=2^-110= 9.183550e-41, (1+eps)-1= 0.000000e+00
eps=2^-111= 4.591775e-41, (1+eps)-1= 0.000000e+00
eps=2^-112= 2.295887e-41, (1+eps)-1= 0.000000e+00
eps=2^-113= 1.147944e-41, (1+eps)-1= 0.000000e+00
eps=2^-114= 5.739719e-42, (1+eps)-1= 0.000000e+00
eps=2^-115= 2.869859e-42, (1+eps)-1= 0.000000e+00
eps=2^-116= 1.434930e-42, (1+eps)-1= 0.000000e+00
eps=2^-117= 7.174648e-43, (1+eps)-1= 0.000000e+00
eps=2^-118= 3.587324e-43, (1+eps)-1= 0.000000e+00
eps=2^-119= 1.793662e-43, (1+eps)-1= 0.000000e+00
eps=2^-120= 8.968310e-44, (1+eps)-1= 0.000000e+00
eps=2^-121= 4.484155e-44, (1+eps)-1= 0.000000e+00
eps=2^-122= 2.242078e-44, (1+eps)-1= 0.000000e+00
eps=2^-123= 1.121039e-44, (1+eps)-1= 0.000000e+00
eps=2^-124= 5.605194e-45, (1+eps)-1= 0.000000e+00
eps=2^-125= 2.802597e-45, (1+eps)-1= 0.000000e+00
eps=2^-126= 1.401298e-45, (1+eps)-1= 0.000000e+00
eps=2^-127= 0.000000e+00, (1+eps)-1= 0.000000e+00
eps=2^-128= 0.000000e+00, (1+eps)-1= 0.000000e+00
eps=2^-129= 0.000000e+00, (1+eps)-1= 0.000000e+00
```

```
force store, break optimization
eps=2^-1= 5.000000e-01, (1+eps)-1= 5.000000e-01
eps=2^-2= 2.500000e-01, (1+eps)-1= 2.500000e-01
eps=2^-3= 1.250000e-01, (1+eps)-1= 1.250000e-01
eps=2^-4= 6.250000e-02, (1+eps)-1= 6.250000e-02
eps=2^-5= 3.125000e-02, (1+eps)-1= 3.125000e-02
eps=2^-6= 1.562500e-02, (1+eps)-1= 1.562500e-02
eps=2^-7= 7.812500e-03, (1+eps)-1= 7.812500e-03
eps=2^-8= 3.906250e-03, (1+eps)-1= 3.906250e-03
eps=2^-9= 1.953125e-03, (1+eps)-1= 1.953125e-03
eps=2^-10= 9.765625e-04, (1+eps)-1= 9.765625e-04
eps=2^-11= 4.882812e-04, (1+eps)-1= 4.882812e-04
eps=2^-12= 2.441406e-04, (1+eps)-1= 2.441406e-04
eps=2^-13= 1.220703e-04, (1+eps)-1= 1.220703e-04
eps=2^-14= 6.103516e-05, (1+eps)-1= 6.103516e-05
eps=2^-15= 3.051758e-05, (1+eps)-1= 3.051758e-05
eps=2^-16= 1.525879e-05, (1+eps)-1= 1.525879e-05
eps=2^-17= 7.629395e-06, (1+eps)-1= 7.629395e-06
eps=2^-18= 3.814697e-06, (1+eps)-1= 3.814697e-06
eps=2^-19= 1.907349e-06, (1+eps)-1= 1.907349e-06
eps=2^-20= 9.536743e-07, (1+eps)-1= 9.536743e-07
eps=2^-21= 4.768372e-07, (1+eps)-1= 4.768372e-07
eps=2^-22= 2.384186e-07, (1+eps)-1= 2.384186e-07
eps=2^-23= 1.192093e-07, (1+eps)-1= 1.192093e-07
eps=2^-24= 5.960464e-08, (1+eps)-1= 0.000000e+00
eps=2^-25= 2.980232e-08, (1+eps)-1= 0.000000e+00
eps=2^-26= 1.490116e-08, (1+eps)-1= 0.000000e+00
eps=2^-27= 7.450581e-09, (1+eps)-1= 0.000000e+00
eps=2^-28= 3.725290e-09, (1+eps)-1= 0.000000e+00
eps=2^-29= 1.862645e-09, (1+eps)-1= 0.000000e+00
eps=2^-30= 9.313226e-10, (1+eps)-1= 0.000000e+00
eps=2^-31= 4.656613e-10, (1+eps)-1= 0.000000e+00
eps=2^-32= 2.328306e-10, (1+eps)-1= 0.000000e+00
eps=2^-33= 1.164153e-10, (1+eps)-1= 0.000000e+00
eps=2^-34= 5.820766e-11, (1+eps)-1= 0.000000e+00
eps=2^-35= 2.910383e-11, (1+eps)-1= 0.000000e+00
eps=2^-36= 1.455192e-11, (1+eps)-1= 0.000000e+00
eps=2^-37= 7.275958e-12, (1+eps)-1= 0.000000e+00
eps=2^-38= 3.637979e-12, (1+eps)-1= 0.000000e+00
eps=2^-39= 1.818989e-12, (1+eps)-1= 0.000000e+00
eps=2^-40= 9.094947e-13, (1+eps)-1= 0.000000e+00
eps=2^-41= 4.547474e-13, (1+eps)-1= 0.000000e+00
eps=2^-42= 2.273737e-13, (1+eps)-1= 0.000000e+00
eps=2^-43= 1.136868e-13, (1+eps)-1= 0.000000e+00
```

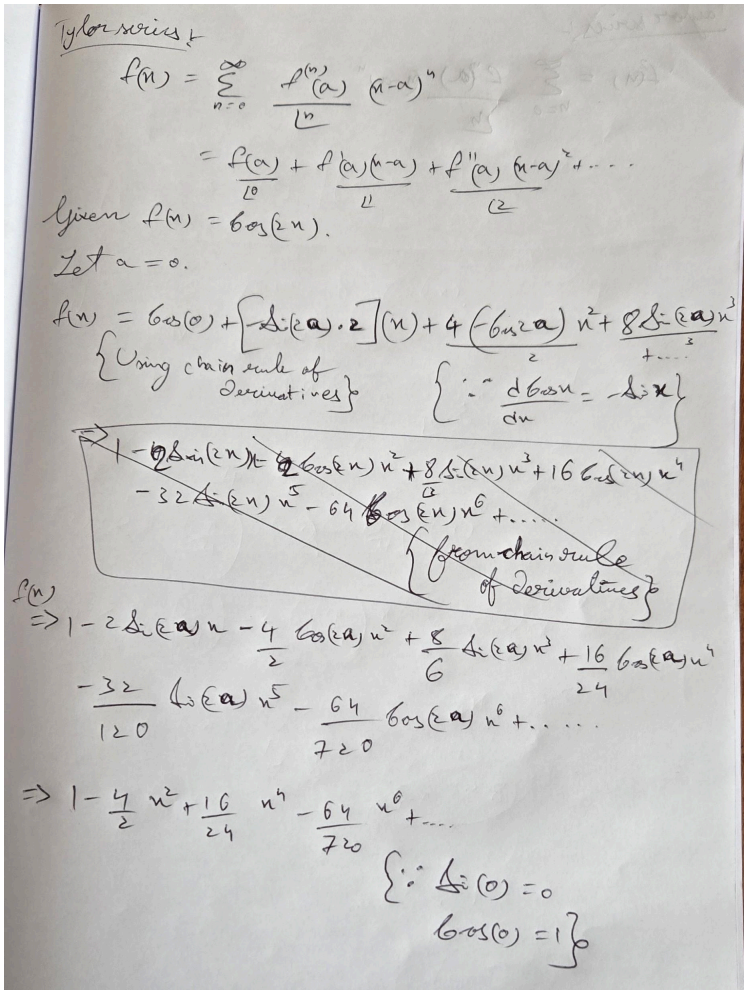
```
eps=2^-44= 5.684342e-14, (1+eps)-1= 0.000000e+00
eps=2^-45= 2.842171e-14, (1+eps)-1= 0.000000e+00
eps=2^-46= 1.421085e-14, (1+eps)-1= 0.000000e+00
eps=2^-47= 7.105427e-15, (1+eps)-1= 0.000000e+00
eps=2^-48= 3.552714e-15, (1+eps)-1= 0.000000e+00
eps=2^-49= 1.776357e-15, (1+eps)-1= 0.000000e+00
eps=2^-50= 8.881784e-16, (1+eps)-1= 0.000000e+00
eps=2^-51= 4.440892e-16, (1+eps)-1= 0.000000e+00
eps=2^-52= 2.220446e-16, (1+eps)-1= 0.000000e+00
eps=2^-53= 1.110223e-16, (1+eps)-1= 0.000000e+00
eps=2^-54= 5.551115e-17, (1+eps)-1= 0.000000e+00
eps=2^-55= 2.775558e-17, (1+eps)-1= 0.000000e+00
eps=2^-56= 1.387779e-17, (1+eps)-1= 0.000000e+00
eps=2^-57= 6.938894e-18, (1+eps)-1= 0.000000e+00
eps=2^-58= 3.469447e-18, (1+eps)-1= 0.000000e+00
eps=2^-59= 1.734723e-18, (1+eps)-1= 0.000000e+00
eps=2^-60= 8.673617e-19, (1+eps)-1= 0.000000e+00
eps=2^-61= 4.336809e-19, (1+eps)-1= 0.000000e+00
eps=2^-62= 2.168404e-19, (1+eps)-1= 0.000000e+00
eps=2^-63= 1.084202e-19, (1+eps)-1= 0.000000e+00
eps=2^-64= 5.421011e-20, (1+eps)-1= 0.000000e+00
eps=2^-65= 2.710505e-20, (1+eps)-1= 0.000000e+00
eps=2^-66= 1.355253e-20, (1+eps)-1= 0.000000e+00
eps=2^-67= 6.776264e-21, (1+eps)-1= 0.000000e+00
eps=2^-68= 3.388132e-21, (1+eps)-1= 0.000000e+00
eps=2^-69= 1.694066e-21, (1+eps)-1= 0.000000e+00
eps=2^-70= 8.470329e-22, (1+eps)-1= 0.000000e+00
eps=2^-71= 4.235165e-22, (1+eps)-1= 0.000000e+00
eps=2^-72= 2.117582e-22, (1+eps)-1= 0.000000e+00
eps=2^-73= 1.058791e-22, (1+eps)-1= 0.000000e+00
eps=2^-74= 5.293956e-23, (1+eps)-1= 0.000000e+00
eps=2^-75= 2.646978e-23, (1+eps)-1= 0.000000e+00
eps=2^-76= 1.323489e-23, (1+eps)-1= 0.000000e+00
eps=2^-77= 6.617445e-24, (1+eps)-1= 0.000000e+00
eps=2^-78= 3.308722e-24, (1+eps)-1= 0.000000e+00
eps=2^-79= 1.654361e-24, (1+eps)-1= 0.000000e+00
eps=2^-80= 8.271806e-25, (1+eps)-1= 0.000000e+00
eps=2^-81= 4.135903e-25, (1+eps)-1= 0.000000e+00
eps=2^-82= 2.067952e-25, (1+eps)-1= 0.000000e+00
eps=2^-83= 1.033976e-25, (1+eps)-1= 0.000000e+00
eps=2^-84= 5.169879e-26, (1+eps)-1= 0.000000e+00
eps=2^-85= 2.584939e-26, (1+eps)-1= 0.000000e+00
eps=2^-86= 1.292470e-26, (1+eps)-1= 0.000000e+00
eps=2^-87= 6.462349e-27, (1+eps)-1= 0.000000e+00
```

```
eps=2^-88= 3.231174e-27, (1+eps)-1= 0.000000e+00
eps=2^-89= 1.615587e-27, (1+eps)-1= 0.000000e+00
eps=2^-90= 8.077936e-28, (1+eps)-1= 0.000000e+00
eps=2^-91= 4.038968e-28, (1+eps)-1= 0.000000e+00
eps=2^-92= 2.019484e-28, (1+eps)-1= 0.000000e+00
eps=2^-93= 1.009742e-28, (1+eps)-1= 0.000000e+00
eps=2^-94= 5.048710e-29, (1+eps)-1= 0.000000e+00
eps=2^-95= 2.524355e-29, (1+eps)-1= 0.000000e+00
eps=2^-96= 1.262177e-29, (1+eps)-1= 0.000000e+00
eps=2^-97= 6.310887e-30, (1+eps)-1= 0.000000e+00
eps=2^-98= 3.155444e-30, (1+eps)-1= 0.000000e+00
eps=2^-99= 1.577722e-30, (1+eps)-1= 0.000000e+00
eps=2^-100= 7.888609e-31, (1+eps)-1= 0.000000e+00
eps=2^-101= 3.944305e-31, (1+eps)-1= 0.000000e+00
eps=2^-102= 1.972152e-31, (1+eps)-1= 0.000000e+00
eps=2^-103= 9.860761e-32, (1+eps)-1= 0.000000e+00
eps=2^-104= 4.930381e-32, (1+eps)-1= 0.000000e+00
eps=2^-105= 2.465190e-32, (1+eps)-1= 0.000000e+00
eps=2^-106= 1.232595e-32, (1+eps)-1= 0.000000e+00
eps=2^-107= 6.162976e-33, (1+eps)-1= 0.000000e+00
eps=2^-108= 3.081488e-33, (1+eps)-1= 0.000000e+00
eps=2^-109= 1.540744e-33, (1+eps)-1= 0.000000e+00
eps=2^-110= 7.703720e-34, (1+eps)-1= 0.000000e+00
eps=2^-111= 3.851860e-34, (1+eps)-1= 0.000000e+00
eps=2^-112= 1.925930e-34, (1+eps)-1= 0.000000e+00
eps=2^-113= 9.629650e-35, (1+eps)-1= 0.000000e+00
eps=2^-114= 4.814825e-35, (1+eps)-1= 0.000000e+00
eps=2^-115= 2.407412e-35, (1+eps)-1= 0.000000e+00
eps=2^-116= 1.203706e-35, (1+eps)-1= 0.000000e+00
eps=2^-117= 6.018531e-36, (1+eps)-1= 0.000000e+00
eps=2^-118= 3.009266e-36, (1+eps)-1= 0.000000e+00
eps=2^-119= 1.504633e-36, (1+eps)-1= 0.000000e+00
eps=2^-120= 7.523164e-37, (1+eps)-1= 0.000000e+00
eps=2^-121= 3.761582e-37, (1+eps)-1= 0.000000e+00
eps=2^-122= 1.880791e-37, (1+eps)-1= 0.000000e+00
eps=2^-123= 9.403955e-38, (1+eps)-1= 0.000000e+00
eps=2^-124= 4.701977e-38, (1+eps)-1= 0.000000e+00
eps=2^-125= 2.350989e-38, (1+eps)-1= 0.000000e+00
eps=2^-126= 1.175494e-38, (1+eps)-1= 0.000000e+00
eps=2^-127= 5.877472e-39, (1+eps)-1= 0.000000e+00
eps=2^-128= 2.938736e-39, (1+eps)-1= 0.000000e+00
eps=2^-129= 1.469368e-39, (1+eps)-1= 0.000000e+00
eps=2^-130= 7.346840e-40, (1+eps)-1= 0.000000e+00
eps=2^-131= 3.673420e-40, (1+eps)-1= 0.000000e+00
```

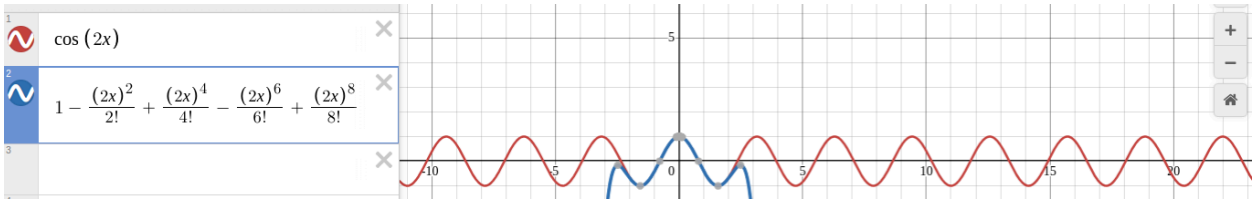
```
eps=2^-132= 1.836710e-40, (1+eps)-1= 0.000000e+00
eps=2^-133= 9.183550e-41, (1+eps)-1= 0.000000e+00
eps=2^-134= 4.591775e-41, (1+eps)-1= 0.000000e+00
eps=2^-135= 2.295887e-41, (1+eps)-1= 0.000000e+00
eps=2^-136= 1.147944e-41, (1+eps)-1= 0.000000e+00
eps=2^-137= 5.739719e-42, (1+eps)-1= 0.000000e+00
eps=2^-138= 2.869859e-42, (1+eps)-1= 0.000000e+00
eps=2^-139= 1.434930e-42, (1+eps)-1= 0.000000e+00
eps=2^-140= 7.174648e-43, (1+eps)-1= 0.000000e+00
eps=2^-141= 3.587324e-43, (1+eps)-1= 0.000000e+00
eps=2^-142= 1.793662e-43, (1+eps)-1= 0.000000e+00
eps=2^-143= 8.968310e-44, (1+eps)-1= 0.000000e+00
eps=2^-144= 4.484155e-44, (1+eps)-1= 0.000000e+00
eps=2^-145= 2.242078e-44, (1+eps)-1= 0.000000e+00
eps=2^-146= 1.121039e-44, (1+eps)-1= 0.000000e+00
eps=2^-147= 5.605194e-45, (1+eps)-1= 0.000000e+00
eps=2^-148= 2.802597e-45, (1+eps)-1= 0.000000e+00
eps=2^-149= 1.401298e-45, (1+eps)-1= 0.000000e+00
eps=2^-150= 0.000000e+00, (1+eps)-1= 0.000000e+00
eps=2^-151= 0.000000e+00, (1+eps)-1= 0.000000e+00
eps=2^-152= 0.000000e+00, (1+eps)-1= 0.000000e+00
eps=2^-153= 0.000000e+00, (1+eps)-1= 0.000000e+00
eps=2^-154= 0.000000e+00, (1+eps)-1= 0.000000e+00

end epsilon.c
```

Question 6: (15 points) Derive the Taylor series expansion for cos(2x) by hand. Show all the steps.

Ans:




Validation:




Question 7: (10 points) Write a short code without using built-in functions to calculate the Taylor series expansion of sin(x) out to n-number of terms.

a) (3 points) Plot the results of the Taylor series using 3 terms against the built-in function for sin(x) expanded around a = 0.

b) (2 points) Plot the results of the Taylor series using 10 terms against the built-in function for sin(x) expanded around a = 0.

c) (3 points) Derive a Taylor series expansion about a=1.2. Code this into a function and

plot against the analytical function. Show the derivation by hand and plot for N=4 and N=6.

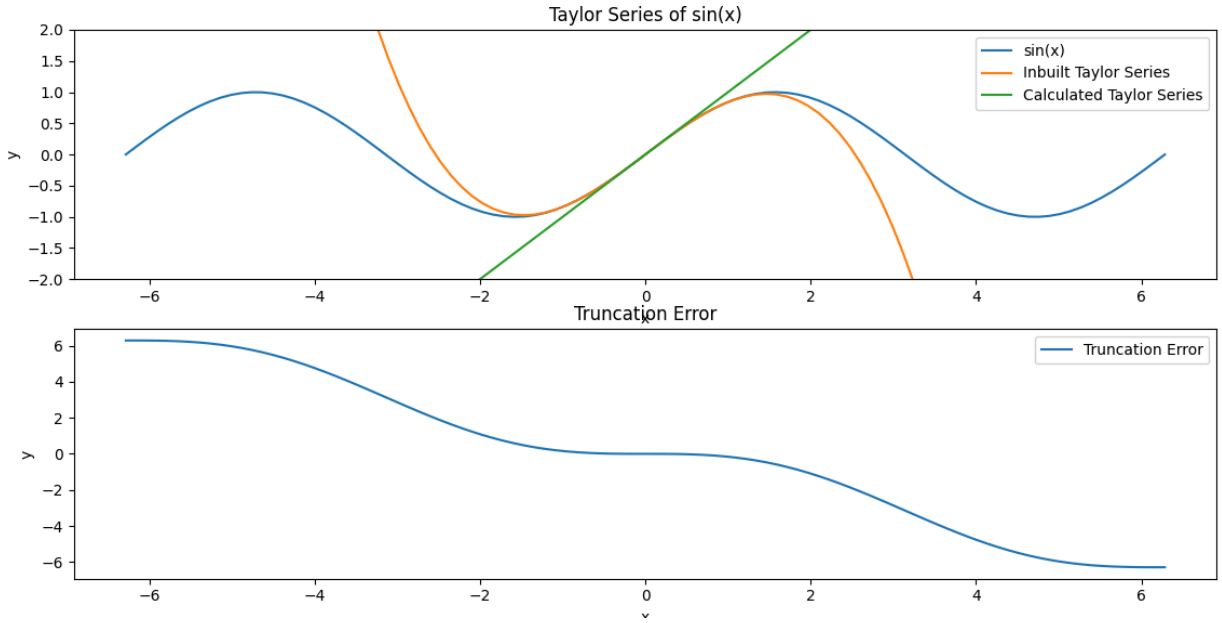
d) (2 points) What is the truncation error at a=1.2 for N=4 and N=10?

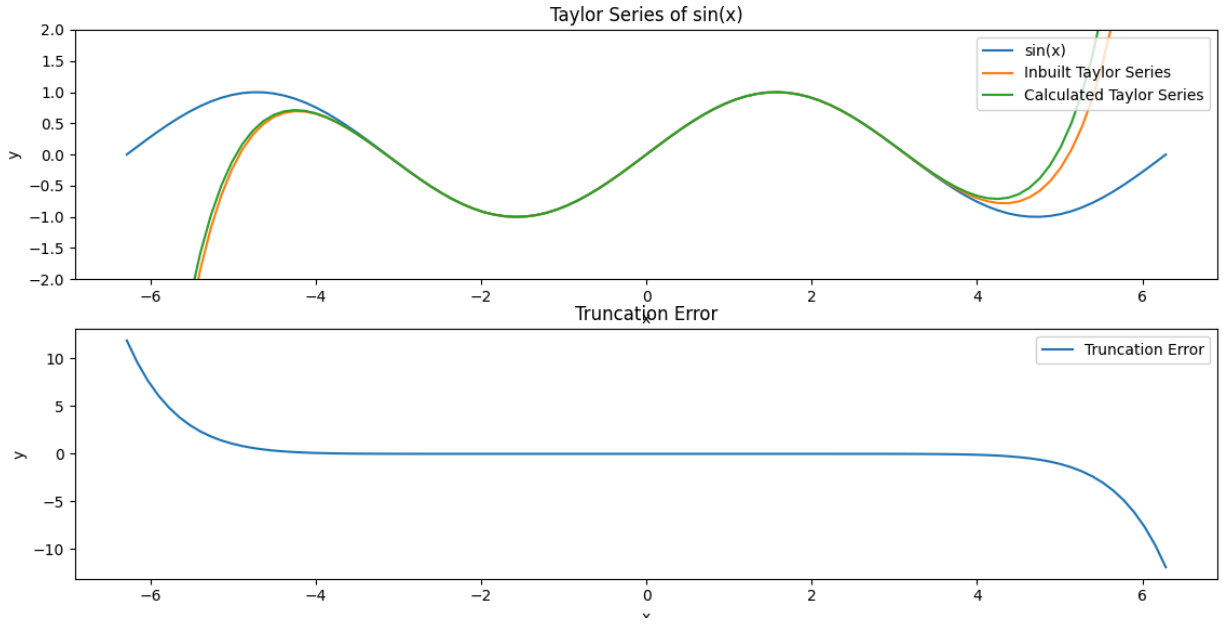
Ans:
Python code to calculate the Taylor series expansion of sin(x):

```
import numpy as np
import math
import matplotlib.pyplot as plt
import argparse
from scipy.interpolate import approximate_taylor_polynomial

parser = argparse.ArgumentParser()
parser.add_argument('-N', '--terns', type=str, help='No of terms in Taylor
Series')
parser.add_argument('-A', '--evalpoint', type=str, help='Angle in
radians')

args = parser.parse_args()

nu_terms = 7 if args.terns is None else int(args.terns)
eval_point = 0 if args.evalpoint is None else float(args.evalpoint)

inbuilt_sin_taylor = approximate_taylor_polynomial(np.sin, eval_point,
nu_terms, 1)

x = np.linspace(-2*np.pi, 2*np.pi, 100)
y = np.sin(x)
y_taylor = inbuilt_sin_taylor(x)

sin_taylor = 0
sign_counter = -1
for n in range(0,nu_terms):
   if n%2 == 0:
       sign_counter *= -1
   if n%2 == 0:
       sin_taylor += (sign_counter * ((np.sin(eval_point) *
(x-eval_point)**(n)) / math.factorial(n)))
   else:
```

```
        sin_taylor += (sign_counter * ((np.cos(eval_point) *
(x-eval_point)**(n)) / math.factorial(n)))

truncation_error = y - sin_taylor

fig, ax = plt.subplots(2, 1, figsize=(10,10))
ax[0].plot(x, y, label='sin(x)')
ax[0].plot(x, y_taylor, label='Inbuilt Taylor Series')
ax[0].plot(x, sin_taylor, label='Calculated Taylor Series')
ax[0].legend()
ax[0].set_title('Taylor Series of sin(x)')
ax[0].set_xlabel('x')
ax[0].set_ylabel('y')
ax[0].set_ylim(-2, 2)

ax[1].plot(x, truncation_error, label='Truncation Error')
ax[1].legend()
ax[1].set_title('Truncation Error')
ax[1].set_xlabel('x')
ax[1].set_ylabel('y')

plt.show()
```

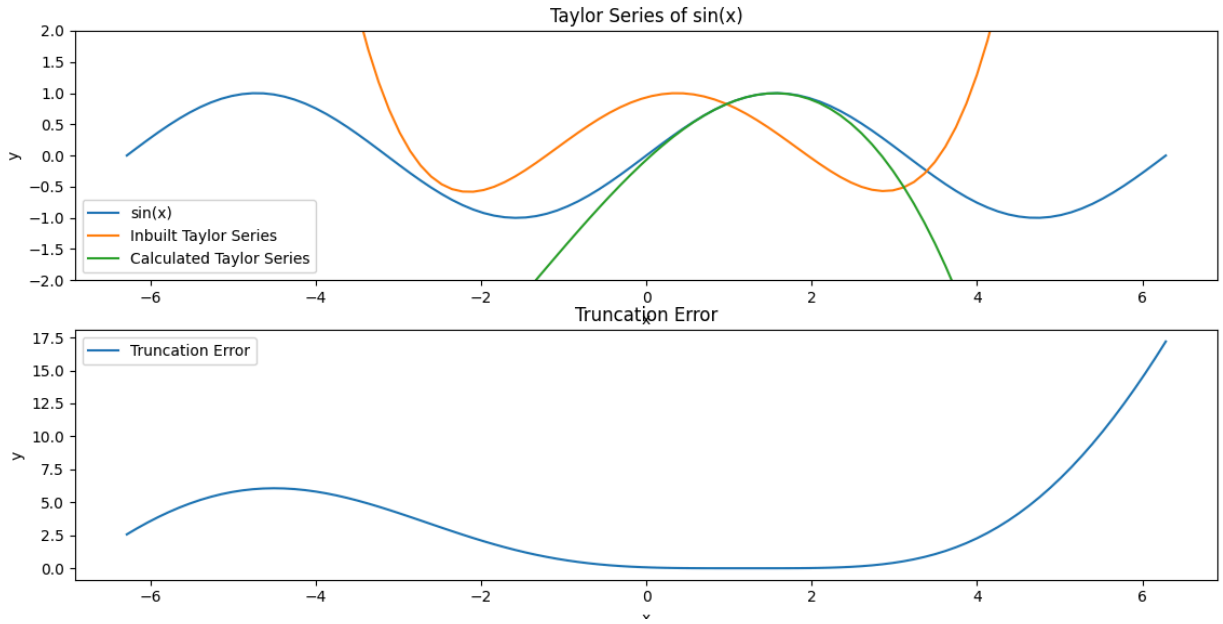
a)



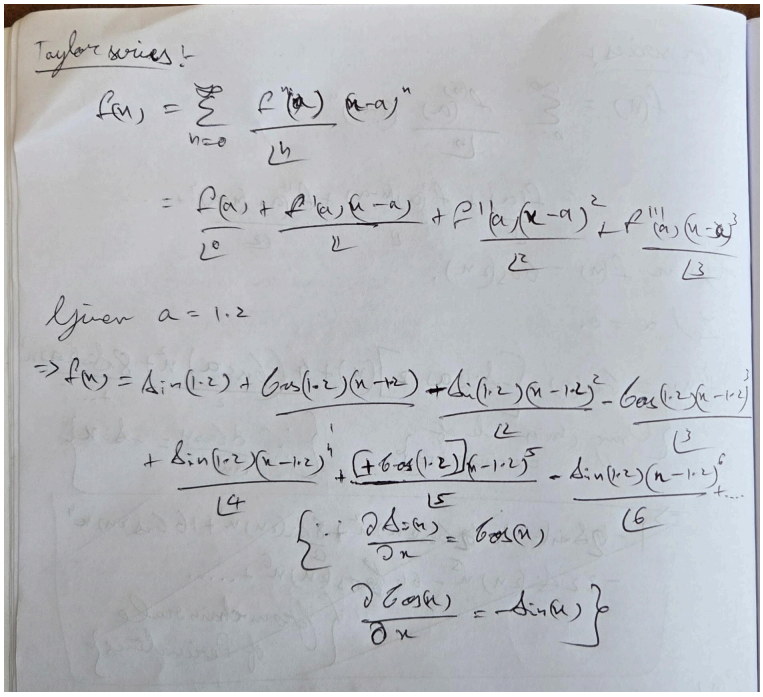


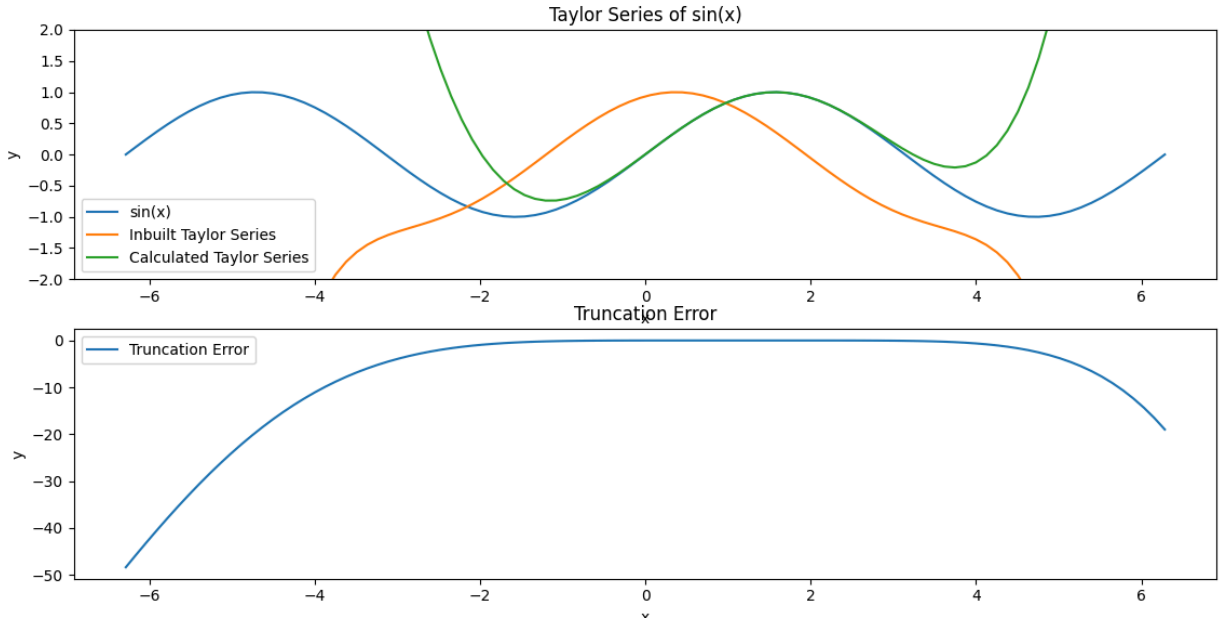

b)

c) Taylor Series expansion at 1.2 with 4 terms, along with the truncation error:

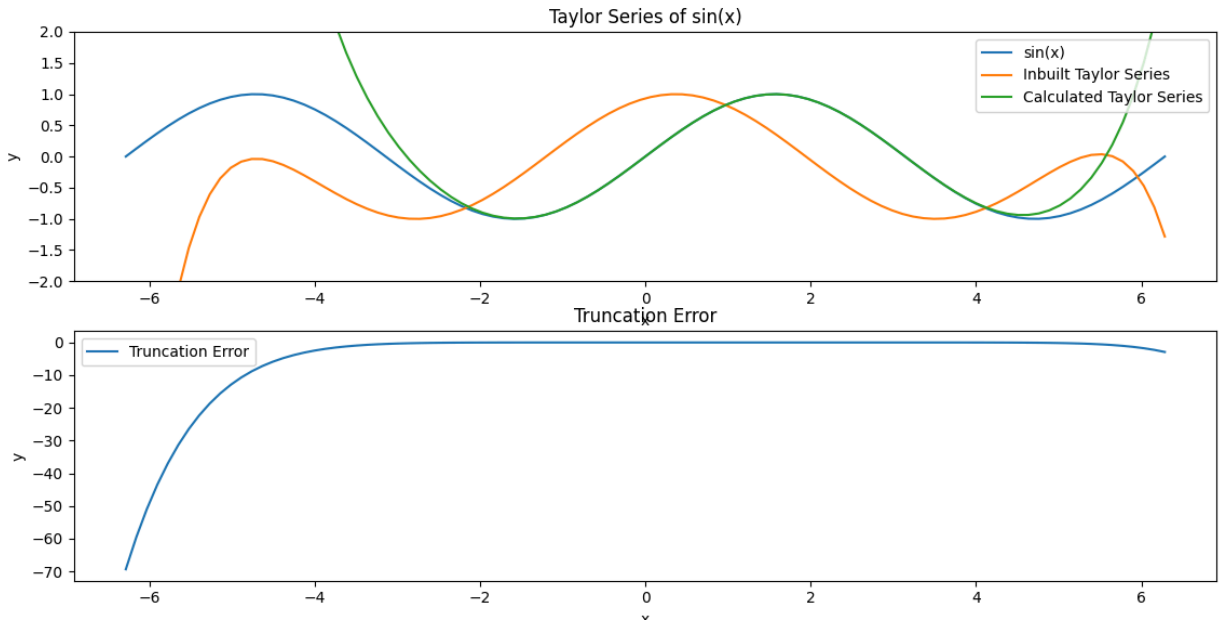



Taylor Series expansion at 1.2 with 6 terms, along with the truncation error:




d) Truncation error with 10 terms at 1.2:




Question 8: (10 points) Create a code that uses bisection to find the roots of a curve. Then create
a code that uses Newton's method. Within the code, keep track of the number of iterations used for each method to find each root.

a. The quadratic (x-0.3)*(x-0.5) has zeros at 0.3 and 0.5. Show that your bisection code can find both roots. How many iterations did your code use at each root? Be sure to mention what you used for your tolerance value.

b. Find the zeros using your code for Newton's method. How many iterations did it use for each root? Compare to the number of iterations used for bisection method.

c. Using your bisection code where do the curves of y=cos(x) and y=x3-1 intersect?

Ans:
Code for bisection and Newton's method of finding roots:

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

void root_cal_bisection(double root, double tolerance, double l, double r,
int nu_iterations){
 double func_l = pow(l,2)-0.8*l+0.15, func_r = pow(r,2)-0.8*r+0.15;
 double m = (l+r)/2;
```

```
 double func_m = pow(m,2)-0.8*m+0.15;
 double abs_error = ((m - root) > 0) ? (m - root) : (root - m);

 while(abs_error>tolerance){
   if ((func_m*func_l)<0){
     r = m;
     func_r = func_m;
   }
   else{
       l = m;
       func_l = func_m;
   }
   m = (l+r)/2;
   nu_iterations++;
   func_m = pow(m,2)-0.8*m+0.15;

   abs_error = ((m - root) > 0) ? (m - root) : (root - m);
 }

 printf("Root calculated in Bisection method with tolerance %lf after %d
iterations: %lf\n", tolerance, nu_iterations, m);
}

double func_eval(double x){
 return pow(x,2)-0.8*x+0.15;
}

double func_eval_derivative(double x){
 return 2*x-0.8;
}

void root_cal_newton(double root, double tolerance, double m, int
nu_iterations){
 double abs_error = ((m - root) > 0) ? (m - root) : (root - m);
 double func_m = func_eval(m);
 double func_m_derivative = func_eval_derivative(m);

 while(abs_error>tolerance){
   if (func_m_derivative == 0){
     printf("Derivative is zero. Exiting\n");
```

```
      exit(0);
    }
    m = m - (func_m/func_m_derivative);
    nu_iterations++;
    func_m = func_eval(m);
    func_m_derivative = func_eval_derivative(m);

    abs_error = ((m - root) > 0) ? (m - root) : (root - m);
  }

 printf("Root calculated in Newton's method with tolerance %lf after %d
iterations: %lf\n", tolerance, nu_iterations, m);
}

int main()
{
 double tolerance=0.0125, root1=0.3, root2=0.5, abs_error=0;
  int nu_iterations = 1;

 root_cal_bisection(root1, tolerance, 0.1, 0.4, nu_iterations);
 root_cal_bisection(root2, tolerance, 0.4, 1, nu_iterations);
 printf("\n");
 root_cal_newton(root1, tolerance, 0.1, nu_iterations);
 root_cal_newton(root2, tolerance, 1, nu_iterations);
 return 0;
}
```

a) Output:

```
(numerical-computations) → assignment1 gcc roots.c -o executables/roots.c -lm && ./executables/roots.c
Root calculated in Bisection method with tolerance 0.012500 after 4 iterations: 0.306250
Root calculated in Bisection method with tolerance 0.012500 after 5 iterations: 0.493750

Root calculated in Newton's method with tolerance 0.012500 after 4 iterations: 0.299216
Root calculated in Newton's method with tolerance 0.012500 after 5 iterations: 0.500923
```

b) Above output has the number of iterations for Newton's method.

c) Code to calculate the roots of $x^3 - 1 - cos(x) = 0$ using bisection:

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

void root_cal_bisection(double root, double tolerance, double l,
double r, int nu_iterations){
```

```c
 double func_l = cos(l)-pow(l,3)+1, func_r = cos(r)-pow(r,3)+1;
 double m = (l+r)/2;
 double func_m = cos(m)-pow(m,3)+1;
 double abs_error = ((m - root) > 0) ? (m - root) : (root - m);

 while(abs_error>tolerance){
   if ((func_m*func_l)<0){
     r = m;
     func_r = func_m;
   }
   else{
       l = m;
       func_l = func_m;
   }
   m = (l+r)/2;
   nu_iterations++;
   func_m = cos(m)-pow(m,3)+1;

   abs_error = ((m - root) > 0) ? (m - root) : (root - m);
 }

 printf("Root calculated in Bisection method with tolerance %lf
after %d iterations: %lf\n", tolerance, nu_iterations, m);
}

int main(){
   double tolerance=0.0125, root1=1.12656, abs_error=0;

   int nu_iterations = 1;

   root_cal_bisection(root1, tolerance, 0.1, 02, nu_iterations);
   return 0;
}
```

Output:

```
(numerical-computations) → assignment1 gcc bisection.c -o executables/bisection -lm && ./executables/bisection
Root calculated in Bisection method with tolerance 0.012500 after 7 iterations: 1.124219
```

Question 9: Calculate the FLOPS rate of your computer then write a program to measure the actual performance of your computer. Turn in your performance results and program.

Ans:
FLOPS (Floating point operations per second) can be determined by:

$CPU\ Speed\ *\ nu\ of$ CPU cores * nu of floating point operations by a core

My computer specs:

```
(base) ➜  ~ lscpu | grep "MHz"
CPU max MHz:                           3900.0000
CPU min MHz:                           400.0000
```

```
(base) ➜  ~ nproc
8
```

```
(base) ➜  ~ lscpu | grep "Flags" | grep "avx"
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
 tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm a
bm 3dnowprefetch cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 er
ms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp vnmi m
d_clear flush_l1d arch_capabilities
```

These imply:
Clock speed is 3.9GHz (max)
Nu of cores is 8
Since I have an avx2 (Advanced Vector Extensions 2) flag, my PC can perform 8 64-bit floating-point operations per clock cycle per core
Therefore, the total maximum FLOPS rate is 249.6 Gigaflops.

Program to measure the actual performance of your computer:

```c
#include <stdio.h>
#include <stdlib.h>
#include <cblas.h>
#include <time.h>

int main() {
   int N = 10000;  // Matrix size for large floating-point operations
   double *A = (double *) malloc(N * N * sizeof(double));
   double *B = (double *) malloc(N * N * sizeof(double));
   double *C = (double *) malloc(N * N * sizeof(double));

   // Initialize matrices with random values
   for (int i = 0; i < N * N; i++) {
       A[i] = (double) rand() / RAND_MAX;
       B[i] = (double) rand() / RAND_MAX;
       C[i] = 0.0;
   }

   // Start timer
```

```
    clock_t start = clock();

    // Perform matrix multiplication: C = A * B
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1.0, A,
N, B, N, 0.0, C, N);

    // End timer
    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    // Calculate FLOPS (2 * N^3 operations for matrix multiplication)
    double flops = (2.0 * N * N * N) / time_spent;

    printf("Performed %.0f floating-point operations in %.2f seconds\n",
2.0 * N * N * N, time_spent);
    printf("Estimated FLOPS: %.2e FLOPS\n", flops);

    free(A);
    free(B);
    free(C);

    return 0;
}
```

Output:

```
(numerical-computations) → assignment1 gcc -o executables/performance_flops performance_flops.c -lm -lopenblas -lpthread
 && ./executables/performance_flops
Performed 2000000000000 floating-point operations in 156.24 seconds
Estimated FLOPS: 1.28e+10 FLOPS
```

Source Code Reference:
ChatGPT, 2024