Acer Cristea
Cruzid: acristea
Winter 2023 - CSE 13S
Dr. Veenstra

Assignment 2 DESIGN.pdf

Description of Program

We need to implement a small number of mathematical functions (e^x and sqrt (x), mimicking, and using them to compute the fundamental constants e and π. You will also need to write a dedicated test harness comparing your implemented functions with that of the math library's, then analyze and present your findings in a writeup. The test harness should be a program named mathlib-test. The interface for your math library will be given in mathlib.h. You may not modify this file. You are strictly forbidden to use any functions from your own math library. You are also forbidden to write a factorial() function. Each of the functions you will write must halt computation using an ϵ = 10−14, which will be defined in mathlib.h.

The Deliverables
(files that need to be included in the directory "asgn2")

1. e.c: This file contains two functions: e() and e_terms(). e() approximates the value of e using Taylor series and tracks the number of computed terms using a static variable local to the file. e_terms() returns the number of computed terms.

2. madhava.c: This file contains two functions: pi_madhava() and pi_madhava_terms(). pi_madhava() approximates the value of π using the Madhava series and tracks the number of computed terms with a static variable, exactly like in e.c. pi_madhava_terms() returns the number of computed terms.

3. euler.c: This file contains two functions: pi_euler() and pi_euler_terms(). pi_euler() approximates the value of π using the formula derived from Euler's solution to the Basel problem. It should also track the number of computed terms. pi_euler_terms() returns the number of computed terms.

4. bbp.c: This file contains two functions: pi_bbp() and pi_bbp_terms(). pi_bbp() approximates the value of π using the Bailey-Borwein-Plouffe formula and tracks the number of computed terms. pi_bbp_terms() returns the number of computed terms.

5. viete.c: This file contains two functions: pi_viete() and pi_viete_factors(). pi_viete() approximates the value of π using Viète's formula and tracks the number of computed factors. pi_viete() returns the number of computed factors.

6. newton.c: This file contains two functions: sqrt_newton() and sqrt_newton_iters(). sqrt_newton() approximates the square root of the argument passed to it using the Newton-Raphson method and tracks the number of iterations taken. sqrt_newton_iters() returns the number of iterations.

7. mathlib-test.c: This file will contain the main test harness for your implemented math library. NEED A DIFFERENCE from one approximation to another. Newline should appear after the difference. It should support the following command-line options:
    - -a: Runs all tests.
    - -e: Runs e approximation test.
    - -b: Runs Bailey-Borwein-Plouffe π approximation test.
    - -m: Runs Madhava π approximation test.
    - -r: Runs Euler sequence π approximation test.
    - -v: Runs Viète π approximation test.
    - -n: Runs Newton-Raphson square root approximation tests.
    - -s: Enable printing of statistics to see computed terms and factors for each tested function.
    - -h: Display a help message detailing program usage

8. Makefile: Not provided, and must include:
    - CC = clang must be specified
    - CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified
    - make must build the mathlib-test executable, as should make all and make mathlib-test.
    - make clean must remove all files that are compiler generated.
    - make format should format all your source code, including the header files.

9. README.md: This must use proper Markdown syntax. It must describe how to use your script and Makefile.

10. DESIGN.pdf: What I am currently writing, a description of the programs and the design process including supporting pseudocode.

11. WRITEUP.pdf: This document must be a proper PDF. This writeup must include graphs displaying the difference between the values reported by your implemented functions and that of the math library's. Use a UNIX tool — not some website — to produce these graphs. gnuplot is recommended. It must also include analysis and explanations for any discrepancies and findings that you glean from your testing. the plots that you produced using your bash script, as well as discussion on which UNIX commands you used to

produce each plot and why you chose to use them. You should use LATEX to produce this document.

Pseudocode / Structure

*Code will be spread out across multiple files

Helpful C Documentation:	Most of these functions will need to be double type, as we are using floating point variables.
MATHLIB.C - FUNCTIONS	The header: - stdio.h (the standard library for C) - math.h (this is only used for constants like M_PI which is used for some of the functions in mathlib.c and mathlib-test.c) - mathlib.h (this file contains the function prototypes that you will be referring to in the mathlib.c file)
Step 1: Creating the Makefile	Since no Makefile is provided, we need to make it ourselves to be able to compile and run our files. 1. Make sure to use CFLAG and CLANG
Step 2: Calculating the approximation for e $e = \sum_{k=0}^{\infty} \frac{1}{k!}$	We need two functions e() and e_terms() e(void): 1. Create a static variable as a counter in our loop 2. Create a for loop that has a counter and stops when a term is less than epsilon 3. Keep track of it's previous calculation and multiply it by itself and 1/counter. (Will create the factorial) e_terms(void): 1. Return number of computed terms (static variable)
Step 3: Creating our square root function	Need to create sqrt_newton() and sqrt_newton_iters() sqrt_newton(double x): 1. Create a local static variable 2. Using the python version provided in asgn2.pdf we can replicate a C version. Uses a while loop and

	stops when the absolute of variable y-z is less than epsilon sqrt_newton_iters(void) 1. Returns the static variable keeping track of the number of iterations in the while loop
Step 4: Calculating the approximation for π using pi_madhava() $p(n) = \sqrt{12}\sum_{k=0}^{n}\frac{(-3)^{-k}}{2k+1}$	We need two functions pi_madhava() and pi_madhava_terms() pi_madhava(void): 1. Create another static variable local to file 2. Need top value and bottom value variables for fraction 3. Create a for loop like in *step 2* 4. The top side has a pow. Need to multiply by the previous step to navigate around 4. Bottom is easy: 2 * the count plus 1 5. Create a term where you divide top by bottom value 6. Another term that adds that value by itself each iteration 7. Multiply that by sqrt_netwon(12) pi_madhava_terms(void): 1. Return number of computed terms (static variable)
Step 5: Calculating the approximation for π using pi_euler() $p(n) = \sqrt{6\sum_{k=1}^{n}\frac{1}{k^2}}$	We need two functions pi_euler() and pi_euler_terms() pi_euler(void): 1. Create static local variable 2. Create a for loop same as *step 2* 3. Every term = 1/(c*c) 4. Add terms together after iteration 5. Do the sqrt_newton(6*added term after loop) pi_euler_terms(void): 1. Return static variable
Step 6: Calculating the approximation for π using pi_viete()	We need two functions pi_viete() and pi_viete_terms() pi_viete(void): 1. Create a static local variable 2. Create a for loop same as *step 2* 3. ai is newton_sqrt(2) while for every k in a is sqrt_newton(2 + a*k-1)

$$\frac{2}{\pi} = \prod_{k=1}^{\infty} \frac{a_i}{2}$$	4. Keep track of that value and return it as it slowly reaches epsilon pi_viete_terms(void): 1. Returns the static variable
Step 7: Calculating the approximation for π using pi_bbp() $\sum_{k=0}^{n} 16^{-k} \times \frac{(k(120k+151)+47)}{k(k(k(512k+1024)+712)+194)+15}.$	We need two functions pi_bbp() and pi_bbp_terms() pi_viete(void): 1. Create static variable 2. Use a similar process as *step 4* to get around the pow. This time not used on the fraction but what we multiply is by (16^-1 = 1/16) 3. Return that value once iterations done pi_viete_terms(void): 1. Return static variable
Step 8: Creating our test file	mathlib-test.c is used to test all our files and their functions 1. Create a main(void) function 2. Assign variables to our functions 3. Write lots of print statements to see what our functions return 4. Calculate the difference here (allowed to use math.h)
Step 9: Creating data from our functions and turning them into graphs	1. Similar to asgn1, we need to gather our data (what our functions return), create a bash file to run them, and make graphs using gnuplot inside. For example, we can plot the error of difference between our approximation and math.h's.