Assignment 2 – Slice of Pi

Kevin Gallegos

CSE 13S – Spring 2023

Purpose

PI is a significant and important number in the world of mathematics that is applied in many aspects and corners of the math world, useful to engineers, programmers and in many other fields. The purpose of this program is to celebrate the upbringings of pie and e, showing off a few formulas by geniuses and mathematicians alike. This program shares the outputs of various formulas, shares the number of iterations it took to the attempt at finding PI accurately and the programs accuracy of PI and E compared to an accurate representation of PI itself, courtesy of the math.h PI functions.

How to Use the Program

To simulate the program you must direct yourself to the mathlib-test file to which you will have various options that allow you to test them against PI representations from the math.h functions. Here are the options you will use:

"-e", will test the programs e approximation against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers.

"-b", will test the programs PI approximation using the Bailey-Borwein-Plouffe formula against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers.

"-m", will test the programs PI approximation using the Madhava formula against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers.

"-r", will test the programs PI approximation using the Euler formula against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers.

"-v", will test the programs PI approximation using the Viete formula against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers.

"-w", will test the programs PI approximation using the Wallis formula against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers.

"-n", will test the programs Square root approximation using the Newton-Raphson formula against math.h's approximation, outputting the number, amount of iterations it took to reach the approximation and its difference between the math.h numbers. This will use numerous inputs as a comparison.

"-a" will run all of the test mentioned above at once.

"-s" will enable printing of statistics to see computed terms and factors for all tested functions.

"-h" will display all available commands that can be entered to simulate the program, displaying the information above.

[1]

Program Design

This program uses various files containing various loops in order for everything to work as intended. Each approximation of PI, e, and square root use a similar algorithm consisting of variable declaration, the main while loop and algorithm within the main loop, this part will break down each and every one to communicate the process. In all files excluding mathlib-test a variable 'itrk' will take charge in iterating through the main while loop, adding 1 to itself for every iteration and stopping the loop when a certain number of iterations is met. Each file will also have a unique function counting the amount of iterations done to reach PI. With this in mind now each file will be explained in simple terms to reflect and explain its design.

- "newton.c" finds the square root of any given double variable 'x', It copies the value of x to another variable, 'n' and stores it for the use in the function: i = (n + x/n)/2 where i becomes stored n this time around and iterates a set amount of times. Every iteration allows the approximation of n to get smaller to accuracy of the square root of given variable x.

- "e.c" This approximates the number e "2.71828". The function will add 2 to the final approximation as the first two iterations of the function will give 1. Due to this the iteration will start at 2 instead at 0. The program simulates the function 1/k! which expand to (0)1, (1)1/1*1, (2)1/1*2, (3)1/1*2*3..... The variable x will share the current iteration and use it to calculate the factorial (k!) slowly counting down to zero to know when to stop. For example if the iteration is at 3, it will count down allowing for the while loop to determine 3*2*1, the factorial of 3 (3!). From there 1 will be divided by this calculation at added to a variable adding each of this iterations to approximate e.

- "bbp.c" Finds the approximation of pi using the Bailey-Borwein-Plouffe formula. It first starts with 16 to the power of 'k' based on the iteration it will multiply 16 by itself k amount of times while k is determined by the current iteration of the program, this is shared with variable z and counts down to determine the amount of time to multiply 16 by itself using k as reference. This answer will be divided by 1 and from there using 'k' (current iteration) it will multiply itself by a function containing a series of fractions. This all together is added into the pi variable and iterates again adding to pi until pi is approximated.

- "madhava.c" Finds the approximation by using the Madhava equation. The program first references 3 to i and z to the current iteration this is used to calculate the -3 to the power -k. The program simplifies the part by instead calculating 1/-3 to the power of k. Before the iteration of powers similar to the ones used in the programs above. Counting down z until the correct amount of time 3 is multiplied to itself. This will be saved as variable i From there based on the current iteration to determine k's value, 2k+1 is calculated and saved as variable l. In each iteration i/l will be added until reaching a set amount of iterations will be multiplied by sqr(12) this calculation is courtesy of the square root function created earlier.

- "euler.c" Finds the approximation of PI using Euler's method. The iteration is defined to start at "1" for this formula. The equation has the current iteration to the power of 2 making the first task as multiplying k by itself. From there it is divided as 1/k to the power of 2.This answer will be added to the variable pi and so will the rest of the iterations. This will iterate a set amount of times and then all of the collected data in PI will be multiplied by 6 and squared (using the square root function from earlier) to approximate PI

- "Viete.c" This will find the approximation of PI using Viete's formula. The program will use the square root function from earlier to find the square root of 2 and save it as t. From then in each iteration after the first will as the approximation to 2 and find the square root of the product. This will be saved as the new t for the next iteration to solve. Within each iteration t will be divided by 2 and saved to pi through multiplication to approximate PI.

- "wallis.c"This function uses wallis' function to iterate and and approximate PI. The program first starts by multiplying the current iteration by itself and multiplying by 4, this is saved as i. The program will repeat this process again but will subtract 1 and save it as l. The program will will then save this to

PI by multiplying i/l into pi and iterate a set amount of times. Once iteration is complete PI will be multiplied by 2 to approximate the value of PI

- "mathlib-test" This will compile all of the previous files into an easy to use command line prompt that will allow the user to test each function compared to its math.h counterpart one by one.

Data Structures

"newton.c"

```
double n, i, itrk, x
x: number that will be used to calculate its square root
n: copy of x for function use
i: compilation of equation, copied into n after equation
itrk: tracks current iteration
```

"e.c"

```
double k, x, e, i, itrk
k: zero used to communicate with x when to stop loop for i
x: copy of itrk to be used in function without editing current itration value
i: saves solution of the iteration to be used by final step
e: saves solution of current iteration adding the rest to it from communication with i
```

"bbp.c"

```
double i, x, z, pi, itrk
x: saved as 16 to be used in function to simulate powers using i and z.
i: saves the solution to the while loops and enters them to pi
z: copy of itrk to simulate powers based on current iteration and check value of iteration.
pi: saves the answer at the end of function fed by i.
itrk: cuurent iteration.
```

"madhava.c"

```
double z, x, i, l, pi, itrk
z: copy of itrk to be used within the function
x: set as the 3 within the madhava equation to be used in calculating in loop
i: copies x/ simulates value of x based on the iteration and uses it in loops
l: calculates and saves the denominator using current iteration to influence solution
pi: saves the compiled iteration of i/l and continues the process of calculatin pi.
itrk: tracks current iteration.
```

"euler.c"

```
double i,x, pi, itrk
i: copy of itrk/x to be used to bring it to the power of 2 and save 1/x
x: copy of itrk
pi: saves the current iteration and adds on to previous to compile pi.
itrk: tracks current iteration.
```

"viete.c"

```
double i, d, t, pi, itrk
i: keeps track of 2 to be plugged into square root
t: saves square root of 2 and its iterations
```

```
d: saves the compilaton of the current iteration that includes t
pi: saves iterations mulitplied by each other to calculate pi.
itrk: tracks iterations
```

"wallis.c"

```
double i, x, l, pi, itrk
i: keeps track of itrk and calculates powers/equation with it and saves it each iteration
x: copy of itrk
l:saves i and subtracts one
pi: saves i/l and multiplies it with pervious iteration calculate pi
itrk: keeps track of iterations
```

Algorithms

"newton.c pseudocode"

```
while(<maximum iterations)
    equation = (current eq+ (current eq/current eq))/2
        current eq = equation
            iteration + 1
```

"e.c pseudocode"

```
while( <maximum iterations)
    x = iteration value
    while (< iteration value)
        pow = pow * iteration value
    e = e + 1/pow
    +1 to iteration value
```

"bbp.c pseudocode"

```
while(<maximum iterations)
    if iteration is at position 0
        set denominator to 1
    else while loop repeats < iteration position
        pow = pow * 16
    pi is pi + 1/pow * 4/8k+1 - 2/8k+4 - 1/8k+5 - 1/8k+6
    iterate again (k)
```

"madhava.c pseudocode"

```
while(<maximum iterations)
    if iteration postion 0
        set denominator to one
    if enumerator position 1
        set denominator to -3
    else denominator = -3^iteration value
    i is 1/denominator
    l is 2 * iteration position + 1
    pi is previous iteration i/l  + i/l
    add position value by 1
pi is pi * sqr(12)
```

"euler.c pseudocode"

```
while(<maximum iterations)
    new iteration value = iteration value * iteration value
    pi = pi + 1/new iteration value
    +1 iteration value
pi * 6
sqr(pi)
```

"viete.c pseudocode"

```
while(<maximum iterations)
    if iteration position is 1
        keep current t (sqr(2))
    else: new t= sqr(2) + previous t
    fraction = t/2
    add fraction to pi variable by multiplying in
    + iteration
```

"wallis.c pseudocode"

```
while(<maximum iterations)
    x  = iterate current postion value to power of 2 then multiply by 4
    pi equals x/x-1 * tpreviosu iteration
    iteration value +1
```

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

- "sqrt-newton()" finds the square root of any given double variable x with an input of double 'x', outputs square root of x.
- "e()" This attempts approximates Euler's number e "2.71828" using a while loop returning 'e'
- "pi-bbp()" attempts to approximate pi using Bailey-Borwein-Plouffe formula. It runs 1 loop and outputs 'pi'
- "pi-madhava()" attempts to approximate pi using the madhava equation, it returns "pi" and uses sqrt-newton().
- "pi-euler()" Finds the approximation of PI using Euler's method. It returns pi and uses sqrt-newton().
- "pi-viete()" finds the approximation of PI using Viete's formula. It returns pi and uses sqrt-newton().
- "pi-wallis()" uses wallis' function to iterate and and approximate pi. It returns "pi".

.

Results

The code successfully executes and runs but the approximations are off for most of them. It is lacking in certain areas where it could use minor clean ups and clearer code. It could most definitely be improved by getting the approximations more accurately

Error Handling

approximations only go so far and do not continue on closer to more accurate approximations.

References

[1] Jess Srinivas. Tutoring hours, monday 12pm -2pm, thursday 10am -12pm.