Approximations and Representing Real Numbers in C

Isabella Phung

January 2023

1 Introduction

Due to the nature of computers and the way they represent values in binary, from a mathematics standpoint it's not possible to represent every real number with some floating point value. Fractions of the following format can be represented in binary:

$$\frac{1}{2^n}$$

But fractions such as 1/3, 2/5 all technically cannot be represented even if a computer seems to be able to. When working with floating point numbers, oftentimes the accuracy begins to dwindle as more and more fractions are piled on. This can be detrimental when working with small values due to relative error. While the error may seem miniscule, when compared to the values being computed with, the error is actually quite significant. The values our computers spit out are likely a close estimation of the fraction, rather than a true decimal point. Although our computers seem capable of extreme mathematical acrobatics with no effort, on a fundamental level, a processor is only truly capable of addition, subtraction, multiplication, division, and shifting binary numbers. Beyond that, it's not directly capable of spitting out sin graphs, performing derivatives, or algebra. This is where, yet again, more estimation comes into play. Rather than perform the computation in a more "human" way, we can use mathematical methods to estimate certain values, rather than store them as constants. It's important that these methods are fast and efficient for quick loading.

2 Initial Approach and Pseudocode

For the newton.c file, a square root function located in initial.c and the instructors have allowed us to borrow it with credit. I plan to tweak it to accept the absolute value function provided in the mathlib.h header file. The way this function works is that it performs a binary search to locate the square root. If the value is larger than 4, the function will remove any extra factors then

multiply those factors back in later. The function guess where the square root is, squares that value, then compares it to the value to be squared. The value is adjusted until the difference between the original value and the approximate square is within epsilon. Epsilon has been provided by the instructors to be

$$10^{-14}$$

to serve as an acceptable error between our estimations and the accepted values from the math.h library. To create a mathlib-text.c file, I plan on studying the monte_carlo.c file from last assignment to better understand how to accept op codes. The Makefile for this assignment will also be closely modeled after the provided Makefile for the previous assignment. For the e.c file, my pseudocode and thinking is elaborated below: As described in page 3 of the spec, we'll be using the following equation by Jacob Bernoulli to estimate e.

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

With observation we can not that when

$$k = 0, \frac{1}{0!} = 1$$

$$k = 1, \frac{1}{1!} = 1$$

$$k = 2, \frac{1}{2!} = \frac{1}{2}$$

This will require a for loop to perform the summation.

- Initialize total variable to 1 to account for 0!
- Initialize iteration variable i to 1, value will be static
- Start denominator at 1 since we've initialized total with 1
- Some loop:
  - Multiply denominator by i
  - factor = 1/denominator
  - Add factor to total
  - iterate i
- If factor is greater than epsilon, loop

For e_terms, return i

For pi_madhava.c and madhava_terms.c, we'll be using the following equation to compute the madhava estimate:

$$\sqrt{12} \sum_{k=0}^{n} \frac{-3^{-k}}{2k+1}$$

- Initialize iteration variable i to 0, value will be static
- Start total value at 1
- Since total variable starts at 1, then start numerator from -1/3 and denominator to 3
- Add 1 to iteration variable since total starts at 1
- some loop:
  - Let's separate the numerator and denominator values, to keep things organised.
  - Let's observe the patterns of the numerator:

$$(-3)^{-k} = \frac{1}{(-3)^k}$$

$$(-3)^0 = 1$$

$$(-3)^{-1} = \frac{-1}{3}$$

$$(-3)^{-2} = \frac{1}{9}$$

  - Looks like we multiply numerator by a factor of -1/3.
  - Let's observe the patterns of the denominator:
  - 2k + 1 equivalent to 2(i) + 1

$$2(1) + 1 = 3$$

$$2(2) + 1 = 5$$

$$2(3) + 1 = 7$$

  - So, we'll add 2 each time to the denominator
  - Then we iterate i by 1
  - factor = numerator divided by denominator
  - add factor to total
  - iterate i
  - while factor is greater than epsilon (checked indirect.c, be sure to absolute value factor while checking against epsilon), loop
- multiply factor by sqrt(12)

For madhava_terms, return iteration variable i

For pi_euler and euler_terms.c: We'll be using the following equation to perform the euler approximation:

$$\sqrt{6\sum_{k^2}^{n}\frac{1}{k^2}}$$

- Initialize total variable to 0.
- Initialize factor variable to 1
- Initialize iteration variable i to 0, make static.
- some loop:
  - Observation shows:
    $$\frac{1}{k^2} = \frac{1}{k*k}$$
  - Therefore, if we substitute our iterative variable i, we get:
    $$factor = \frac{1}{i*i}$$
  - So all we have to do is compute factor = 1/i*i each time we loop.
  - Add factor to total
  - Iterate i
  - While factor is greater than epsilon, loop
- Multiply total with 6, then sqrt.

For euler_terms return variable i

For pi_bbp.c and bbp_terms.c, we'll be using the following equation to approximate pi.

$$\sum_{k=0}^{n} 16^{-k}(\frac{(k(120k+151)+41)}{k(k(k(512k+1024)+712)+194)+15})$$

- Initialize iteration variable i to 0, make static
- Initialize total value, numerator, and denominator to 0
- Initialize some sixteenFactor to 1
- loop:
  - numerator = i*(120*i + 151)+47
  - denominator = i*(i*(i*(512*i + 1024) + 712) + 194) + 15

  - With observation, we can note that:

$$16^{-k} = \frac{1}{16^k}$$

$$16^0 = 1$$

$$16^{-1} = \frac{1}{16}$$

$$16^{-2} = \frac{1}{16 * 16}$$

  - So, we just need to multiply sixteenFactor by 1/16 each time.
  - Factor equals numerator divided by denominator times sixteenfactor
  - Add factor to total
  - Iterate i
  - Multiply sixteenFactor by 1/16
  - While factor <epsilon, loop

For bbp_terms return variable i

For pi_viete.c and viete_terms.c, we'll be using the following equation:

$$\frac{2}{\pi} = \prod_{k=1}^{\infty} \frac{a_k}{2}$$

where

$$a_1 = \sqrt{2}$$

- Initialize iteration variable i to 0, make static
- Initialize total value to 1
  - Observation shows:

$$a_k = \sqrt{2 + a_{k-1}}$$

$$a_1 = \sqrt{2}$$

$$a_2 = \sqrt{2 + \sqrt{2}}$$

$$a_3 = \sqrt{2 + \sqrt{2 + \sqrt{2}}}$$

  - So we initialize numerator as sqrt(2)
  - some loop:
  - Each time we set numerator = sqrt of 2+numerator
  - factor = divide numerator by 2
  - multiply total by factor

  - iterate i
  - check if factor is >epsilon, if true, loop
- this total equals 2/pi, so to get pi, divide by 2, then divide 1 by that value

for viete_terms, return variable i