AM129 HW 3 Deliverable

Sarosh Sopariwalla
November 5, 2020

Abstract

This two-part assignment concludes AM 129's section on Fortran. For the first part, we completely write our own code to approximate π using the fact that $\pi \approx \sum_{n=0}^{\infty} 16^{-n}(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6})$. (Note that this sum converges VERY quickly to 3.14159...) We created three files to help us: pi.f90, pimod.f90, makefile. For the second part of the code, we received a file that solves a three dimensional system of equations using Gaussian elimination. However, this file was riddled with syntax errors and logical mistakes; our task was to fix it so that the file could run and give the correct answer.

Approximating π

How the Program Works

First, we examine the main file, pi.f90. This file is quite simple; we define a parameter N_Max that defines the number of maximum summands. (Obviously we will not evaluate the sum for infinite n, so we need to define a limit.) We also define some double precision reals: thresh, pi_appx,diff. Thresh represents the minimum absolute error we will accept. So if the error in our approximation is less than thresh we stop summing terms (meaning we won't always do fifty sums if the thresh is large enough). pi_appx holds our approximated value of pi. diff holds the absolute difference between pi_appx and and the true value of pi. (The true value of pi was declared as a double precision real equal to the $\arccos -1$) in our pimod.f90 file. We then define an array of various thresholds and iterate through that array. For each threshold, we print the threshold, the approximated value of pi, the number of summands used, and the absolute error. This is all done by calling a subroutine ApproxPi define in pimod.f90.

As one may expect, the purpose of pimod.f90 is really to compute everything for a given thresh hold. To that end, we define thresh and N_Max to have intent(in). We then initialize pi_appx to be zero and $N = 0$. We will use N to record the number of summands. We then use a do loop to add the i^{th} term of the series to our pi_appx. Every time we add on a term we set $N = N + 1$. We then check if the diff is less than or equal to to the threshold: if true, we exit and if false, we continue the loop. Given this subroutine, all we really need to do is call it for various values of thresh. We do this by iterating through our array storing the thresholds.

Speed of Convergence

The sum converges to 3.1415872645447962 within three summands. This is an absolute error of approximately $5.4 * 10^{-6}$. For most cases, this approximation is more than reasonable. However, if we limit ourselves to 51 summands, the most we can minimize error is to $1.3 * 10^{-7}$. Even if we let N_max=500, the error in the approximation is still about $1.3 * 10^{-7}$. Thus, the sum converges very quickly to 3.1415925277860772 (in 51 summands). Even for 50001 summands, the error does not decrease.

Compiler Flags and the Makefile

The makefile is quite simple to understand, especially when we look at HW2. The makefile serves the exact same purpose; we use the same flags to aid in debugging. Specifically we define FFLAGS like so: FFLAGS = -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1 -g3 -fcheck=all -fbacktrace -fdefault-real-8 -fdefault-double-8. The makefile just makes it easier to compile everything all at once rather than having to manually connect pimod to pi each time we need to compile.

Debugging Gaussian Elimination

How we Approached Debugging

Firstly, I moved the subroutines to the end of the program. It was also easy to see that A and b had been defined as allocatable arrays but were never allocated. Thus, we allocated the appropriate memory/dimensions right after defining our variables. Next, I commented out large sections of code. So first, I commented out everything after printing out the augmented matrix. Running the code until there, it was clear that the do loop had wrong bounds and that A was defined incorrectly. (Fortran defines arrays by going down each column not each row.) I also noticed that all of the print statements had syntax errors so I fixed those quickly. Next, we had to examine what was happening with subroutine gaussian_elimination.

First, I made A and b have intent(inout) because we wanted them to be our outputs as well as our inputs. I then walked through the given for loops and saw they were done correctly. Then, I fixed the printing of the echelon form. (Same errors with do loop and print statement). I then moved on to checking the backsubstitution subroutine. This one turned out to be mostly correct except for the do loop defining the solution vector. The bounds of this method had to be adjusted as well.

Some of the remaining do loops also had the issue of improper bounds and so I fixed those. Finally, I noticed that the programs name was different at the beginning and the end so I made sure to standardize that to gauss. I also ended up removing the #ifdef statements since I didn't really need them to help with any debugging. There are probably a few small errors that I am forgetting to mention but these were the biggest and most common ones that needed to be fixed. GDB was particularly helpful since it identified specific lines in which there were syntaxual errors that I could not find looking at the raw code. When used with an IDE like Simply Fortan, these debuggers are even better because they can identify mistakes before even compiling the file.

Solution to Transposed System

The transposed system has augmented matrix $\left(\begin{array}{ccc|c} 2 & 4 & 7 & 1 \\ 3 & 7 & 10 & 3 \\ -1 & 1 & -4 & 4 \end{array}\right)$ Defining a matrix ATr, we solve this system in Fortan and get the result $x = -2.5, y = 1.5, z = 0$.