Simon Lee
AM 129

HW 3 Report

Approximating Pi

Approximating Pi on computers has been a pretty popular problem for as long as I can remember. Outside the material from this class, I remember hearing that computers can provide nothing less than an approximation. No matter how close the program is to the exact number, it is always deemed an approximation. So when we were implementing the approximation program, we were asked to leave some threshold of the number. Denoted in scientific notation if we had a number (i.e `1e-16`) we would have to use a built in number and subtract the estimated number until their value was less than the threshold. The provided table is an example of if the threshold was equal to `1e-16.`

```
1    3.1333333333333333         8.2593202564598123E-003
2    3.1414224664224664         1.7018716732675188E-004
3    3.1415873903465816         5.2632432114840810E-006
4    3.1415924575674357         1.9602235745708185E-007
5    3.1415926454603365         8.1294566633971499E-009
6    3.1415926532280878         3.6170533235235780E-010
7    3.1415926535728809         1.6912249378719935E-011
8    3.1415926535889729         8.2023277059306565E-013
9    3.1415926535897523         4.0856207306205761E-014
10    3.1415926535897913         1.7763568394002505E-015
11    3.1415926535897931         0.0000000000000000
```

As you can see, the code is looping until the difference is less than the threshold `1e-16`. But something important to note is that because we are working with double precision numbers, if you go anything beyond `1e-16` (i.e `1e-17, 1e-18`,....) the computer actually cannot tell the difference between them. So you will be given the same numbers as the example.

Converging Quickly

Another important relationship to point out is that this equation below converges very quickly to the

$$\pi = \sum_{n=0}^{\infty} 16^{-n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right).$$

solution. If we look at the outputs from my program, we can see that the first approximation is `3.1333333333333333.` And although it isn't 3.14 yet, it takes 2 iterations total to achieve the notorious pi sequence. From the third iteration and beyond, it appears to just be getting more precise and resulting in an approximation more like pi. Therefore this equation is mighty strong in approximating pi.

Makefile flags

For this particular assignment I decided to use the following flags:

```
FFLAGS = -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1 -g -fcheck=all
-fbacktrace
```

Here are what these following flags do:

- `-Wall` - "Warn about all", provides warnings about many common sources of bugs like naming conventions and subroutine and functional errors.
- `-Wextra` - checks for more compilation errors beyond the `-Wall` flag.
- `-Wimplicit-interface` - this flag will warn you about undefined symbols which is important because some characters are not displayable in the ASCII table.
- `-fPIC` - This flag makes our code suitable for a library. It is able to be relocated from a destination into a memory address of another location. It allows your code to be included into an address.
- `-fmax-errors=1` - This flag stops at the first error identified.
- `-g` - This flag generates useful debugging information usable by GDB, which is a debugging tool.
- `-fcheck=all` - This flag turns on all debugging checks.
- `-fbacktrace` - This flag turns on backtrace printing when something fails to run at runtime.

A Lot of these compiler flags are simply from the assignment 2 Makefile. But because of their usefulness for debugging and warnings, I have decided to keep primarily the same Makefile flags. The one exception was deleting the `-Wno-surprising` flag because I have no idea what it does. But other than that these flags help allow me to practice writing good code, pointing out errors and warnings that I should fix before submitting this assignment.

Gaussian Elimination Debugging

Gaussian Elimination, which is one of the most fundamental methodologies in linear algebra, was quite the hard program to debug. In practice writing out the elimination on paper is easy, but because we are required to manipulate matrices and vectors, it does get a little tricky having to manage the i and j variables of a matrix. However this also does show you the power of scientific computing, and being able to construct calculators for these tedious mathematical tasks.

My first objective when debugging is always to fix the errors before proceeding with fixing the warnings. After compiling for the first time, I was overwhelmed with the amount of warnings and errors that came up. With subroutines missing the `implicit none` commands, to having silly errors like misspelled subroutines within the program. After completing all the errors, I was able to compile our executable which had another array of problems. For example we indexed a 3x3 matrix but our loops were trying to access memories from 0 to 10. Before I knew about these memory leaks I had to use `lldb` which is essentially the `gdb` tool for Mac users. Using the three following commands I was able to isolate where the errors were occurring, those being: `lldb a.out, run, bt.`

After fixing the memory leaks the last thing I had to do was fix all the warnings. One of the harder warnings I kept getting was "`Warning: 'a.dim[0].ubound' may be used`

`uninitialized [-Wmaybe-uninitialized]`". To get rid of this warning I had to use the `allocate(A(3,3))`. Other than that, alot of the warnings and errors were quite silly and it was pretty easy to go about fixing it. What made this portion of the assignment also manageable was that you had the freedom to change the code however you wanted. So overall through this portion of the assignment I was able to familiarize myself with the `lldb` tools. This portion I believe will play a bigger role in the future when we work with more Fortran or even the C programming language where dynamic allocation of memories will result in many memory leaks. I am not excited for the pointers and the dereferencing of pointers that are bound to come.

NxN Gaussian Elimination Hypothetical Implementation

So if we wanted to change this for the implementation for generic N x N systems, there would be a few parameters that we should change permanently. It is probably best to initialize some N variable and have it be a fixed number of N. Then we could have our A matrix to have dimensions (N,N) and have the solutions vectors be (N,1). After then we could have the loop iterators not be hard coded and have it go until this new variable N. Doing so will not change how the program runs so that I believe is the easiest way to implement this gaussian elimination for some generic NxN systems. A visual representation will be provided below:

```
program gauss
  implicit none
  Real :: N = 4 !any integer can go there for some value N
  real, dimension(N,N) :: A, At
  real, dimension(N,1) :: b, b2
  integer :: i
```

```
 ! When printing any augmented matrix
  do i = 1, N            ! i is row
     print*, A(i,:), "|", b(i)
  end do
```

Conclusion

Overall this assignment allowed us to make our very own modules in Fortran and it allowed us to practice debugging broken programs. These two things are quite important in the field of programming and it is important moving forward to try to understand how to read compiler flags to help you debug. I thought the approximation of pi program was actually quite cool in practice and it was also cool to see the implementation of the gaussian elimination in code form. I am excited for the future assignments for this class that are to come.