Simon Lee
AM 129

HW 2 Report - Boundary Value Problem (BVP)

Overview:

With previous experience as a programmer (Python, C), there were little to no learning curves in terms of how the logic worked in Fortran. However the language and its syntax was something that became apparent immediately. Luckily having done some MatLab, there were some parallels between the two languages. With really nice properties in how they work with matrices (2d-arrays), there were some savvy tools that made me feel clever in implementing some of the code. And although I could have utilized the row and column splicing more, I also saw some of my previous experiences in Python and C appear in this assignment with how I implemented the matrix-vector product. In terms of issues along the way, the one issue I had was trying to debug the division between two integers. Because we were working with double precision, I was confused as to why the matrix entries always had a result of 0. However after attending office hours the point was made immediately that we had to divide using this double precision (i.e. `1.d0 / N`). Other than that the problem itself was pretty straightforward and the difficulty has not scaled too much. But in terms of mathematics, it was cool to find out the origins of the boundary value problem. Being able to finally use scientific computing is exciting. Obviously most of the code was given to us but seeing how powerful these languages can be in solving problems that would take significant time by hand shows us why computation is becoming a powerful tool in not only just math but many applicational sciences.

Code:

As previously mentioned one of the main things I had to adjust to was the `real (dp)` statement which can be used across both our `dft.f90` and `dftmod.f90` modules. The reason we were able to access this precision was because we have one more file called `utility.f90`. And the purpose of this module is that it is included throughout the `dft.f90` and `dftmod.f90` modules which provides a definition for the `real (dp)` statement.

In terms of why we have `dft.f90` and `dftmod.f90` modules, it is because we want to have separate functions perform their subroutine and functions. Obviously this problem required less functions and subroutines but this easily could have split into 4 files where we had a subroutine and function be assigned to a single file. So the purpose of the `dftmod.f90` is to separate the subroutines and functions. Meanwhile the `dft.f90` module is the driver of this code bringing in all the particular parts and giving us our output.

An `external` in the `dft.f90` module did not have to be declared because of the `elemental` keyword. But before we dive into its purpose we should have a better understanding of these two variables. The `external` is when you have some additional procedure after the main block has ended. So in our case our functions are not outside of the

`dft.f90` module. The `elemental` keyword benefits a procedure because it simplifies the parallel execution of a function. This allows computers to execute code more efficiently especially since fortran 90 is a compiled language.

As mentioned in the overview, finally being able to use scientific computing to solve the boundary value problem was exciting. One of the main variables you could change was the dimensions of `N` which resembled the dimensions of the matrix. By plotting some of values of N we could see how our error changed:

`N = 21`	`Err: 9.8179492425645165E-003`
`N = 41`	`Err: 5.5800848541842640E-007`
`N = 61`	`Err: 1.0904832592473213E-011`
`N = 81`	`Err: 6.2172489379008766E-015`
`N = 101`	`Err: 5.1070259132757201E-015`

As the `N` values increase, the Error decreases. It begins to slowly converge at 5.10E-15. This indicates that the solutions don't really matter the higher the value of `N` goes and that it has an elbow curve relationship

Next we will change `dp = kind(0.d0)` to `dp = kind(0.e0)` and provide analysis as to what changes within the table. Plotting those same numbers we arrive at these results:

`N = 21`	`Err: 9.82260704E-03`
`N = 41`	`Err: 8.58306885E-06`
`N = 61`	`Err: 2.38418579E-06`
`N = 81`	`Err: 4.41074371E-06`
`N = 101`	`Err: 3.57627869E-06`

Looking at this data we can begin to see a few differences across the data. The first part being that the numbers converge much quicker than when `dp = kind(0.d0).` First we should point out that it becomes single precision. Next we see that the solutions converge to `E-06.` But instead of approaching a singular number it appears that it is increasing and decreasing, possibly even oscillating around this region.

In terms of how we linked all our files into one executable, we used a Makefile which is a handy tool for compilation of a program. What a makefile does is compile our code into machine

code which comes out to be a `*.o` object file. We then use these object files to and have the machine convert this into a running executable file. Because in the future we can have multiple files needed for a particular program, we construct these makefiles. Makefiles also are nice debugging tools to point out language specific errors. By adding makefile flags we can sort of begin to see what its minimum purpose is to compile. Refer to the below for a description of the flags.

- `-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.
- `-Wno-surprising` - This flag will raise a warning to a surprising component of the code.
- `-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.

Conclusion:

So in totality, this first assignment explored a basic introduction to solving the boundary value problem. Doing so allowed us to see our first experience in scientific computing and have us work with manipulating arrays of 1 and 2 dimensions. This also allowed us to think a little harder about the way a matrix vector product works and manage how the entries of a matrix multiply with the entries of a vector. On top of the mathematics of this project, this also introduced to us the pain of debugging if dynamically allocated arrays were exceeded. Though I did not run into segmentation faults, I know there is a future where I will. With most of the modules provided in this first assignment we were also able to see the components of the makefile and how this is an efficient tool in compiling a whole bunch of files into a nice executable. Therefore in conclusion we were able to dissect and explore some of the debugging flags that gcc provides which provides the helpful messages. I am excited for the future assignments within this class as we begin to ramp up from here.