Boundary Valued Problem (BVP)

Michelle Pichardo Munoz
AM 129: Scientific Computing, HW 3
University of California, Santa Cruz
October 26, 2022

Contents

List of Figures

List of Tables

Git Commit Hash: b41bd44a5c39dddffa2c9c57e37bb6714a9d5baf

Introduction

Do we drive cars? Do we have heating issues in our homes? Do we occasionally wonder why a wave appears when runners run on a circular track? Some people do, and well, those are dynamic systems. Some of which we can solve, and in our case, we will examine no particular system that I'm familiar with, but it does have a solution. More importantly, we can use a computer program to solve it and obtain nearly the exact answer. We can get about 10^{-33} of closeness! The system in question is,

$$\begin{cases} \left(1 - \frac{d^2}{dx^2}\right) u = f \quad x \in (0, 2\pi) \\ u(0) = u(2\pi) \end{cases} \tag{1}$$

Where f is our forcing function,

$$f(x) = e^{sin(x)}(1 + sin(x) - cos^2(x)) + e^{cos(3x)}(9sin^2(3x) - 9cos(3x) - 1) \tag{2}$$

Pulling at straws, the belief is since f is composed of Fourier series related basis vectors i.e., $\{1, \cos(kn), sin(kn)\}$ for $k \in \mathbb{Z}$. We may assume u can be constructed as a series of projections of this basis with f, serving as the constants, multiplied by basis vectors. (This may be a stretch) The following are what we assume u and f could be represented by,

$$f(x) \stackrel{?}{=} a_0 + \sum_{k=0}^{\infty}(a_k \cos(kx) + b_k \sin(kx)) \tag{3}$$

$$u_k(x) \stackrel{?}{=} a_0 + \sum_{k=0}^{\infty}(\frac{a_k}{1+k^2} \cos(kx) + \frac{b_k}{1+k^2} \sin(kx)) \tag{4}$$

where our constants projection-integral representation are,

$$a_0 = \frac{1}{2\pi} \int_0^{2\pi} f(x) \cdot 1 \, dx \tag{5}$$

$$a_k = \frac{1}{\pi} \int_0^{2\pi} f(x) \cdot \cos(kx) \, dx \tag{6}$$

$$b_k = \frac{1}{\pi} \int_0^{2\pi} f(x) \cdot \sin(kx) \, dx \tag{7}$$

The assumptions turn out to be valid, and there exists a matrix representation of this system, which I assume is $[\mathbf{V} - \mathbf{T^{-1}fT}]\mathbf{u} = \mathbf{f}$ to some extent. Where $\mathbf{V}$ is a diagonal matrix serving as potential and $\mathbf{T}$ with its inverse represents the discrete Fourier Transform. Our program will discretize our domain x by setting the number of grid points N and, in turn, determine the length of our sum. In other words, truncate the infinite sum. Using the values of $\mathbf{x}$,$\mathbf{f}$,and $\mathbf{T}$ the program with obtain the constants a_0, a_k, and b_k. We aim to examine the accurateness of this result and the issues that come up from rounding error and truncation of the modes, i.e., values of N.

Methods

In this section, we mainly touch on prevalent issues and troubleshooting strategies.
When first examining the provided code and supplementing the necessary functions and subroutines, it became apparent troubleshooting within the modules would take a lot of work. In response, the file `array1.f90`served as a test area for the requested functions. File `array1.f90`, contained material which is mimicked to complete the matrix multiplication function. The code for T and $Tinv$ was also mimicked from the code to fill k, found in `dft.f90`.

Once the program was presumed complete and ready to run, another error arose. The data file was not generating, and `bvp.ex` was not fully executing. After investigating the file execution routines,

we found a write-to-file error. The program was dependent on a data directory existing locally. We resolved this by removing the snippet of code searching for the directory.

Two unresolved errors occurred, speculated to be user errors or something just deeply wrong with my Fortran install. First, the function `forcing` is created with integer values, as in, the format is not 1.0_fp. With integer values, the code runs and produces replicable results. However, when the floating point underscore is applied, the program breaks, providing strange sizing errors. This issue was unresolved, and we reverted the code to integers. The second issue was in the naming of directories. We changed the program's directory so it wouldn't have the same name as another in another branch. When the names are the same, the compiler will attempt to read modules of similar names but break when the subroutine doesn't exist. The resolution is to rename the code directory codehw3.

More detailed information on the files are found in Table 3 in the Appendix.

Questions

---

1. Why does `real(fp)` can be used throughout the files `bvp.f90` and `dft.f90`

The variable `real(fp)` is able to be called from each other module or program file so long as they import it with `use utility, only: fp`, which they do. It is worth noting, this particular snippet will not work if the module containing `utility` is not in the same directory as the other modules.

---

2 Comment on the separation of duties between `bvp.f90` and `dft.f90`.
Does the distinction make sense?

File `dft.f90` contains subroutines and functions specific to either computing a matrix-vector product or filling in necessary matrices for computation. The name "dft" escapes me. Whereas `bvp.f90` is our main program which calls on all other modules to import, execute, and produce information on the error and a data file containing our discrete domain x, approximated solution u, forcing function f and transformed forcing function ft.

---

3. Why are many of the variables at the top of `bvp.f90` marked `allocatable`?
Where they allocated/deallocated?

Our program needs to be capable of changing N regularly to examine error fluctuations with increasing and decreasing N. Since N is used to determine the size of every vector or matrix, we cannot tell the compiler what amount of memory to store initially. To account for this, we can define our variables as allocatable. As soon as we use the function `allocate` on the required variable, the compiler determines the amount of memory needed and provides it to the variable assigned. Once the program is complete, as a good safety measure, the `deallocate` function returns the memory used to the computer.

---

4. The functions `exact_sol` and `forcing` in `problemsetup.f90` are marked as `elemental`.
What does this mean? Why are they marked this way?

It is assumed elemental functions versus normal functions differ in processing input parameters. When troubleshooting, the normal Fortran function would not accept an array as an input. However, the elemental function allowed an array input and produced an array output. It is deduced that elemental functions retain the shape of their inputs.

Results

Set of tables with varying N, number of grid points, and varying floating point precision, `fp`.

Table 1: Precision: fp = `selected_real_kind(15)`, i.e. double precision of at least 15 decimal places.(8-Byte IEEE Float)

Number of Points (N)	Error	Observation (if applicable)
21	9.82E-03	Largest error correlates to lowest N
41	5.58E-07	
61	1.09E-11	
81	5.995E-15	
101	5.77E-15	We see ΔError is decreasing
211	2.66E-15	This is the last odd value of points before increasing again
213	5.55E-15	
215	3.99E-15	From here onward the error fluctuates

Table 1 reveals a steady decrease of error by a factor of 1k each time we increase by 20 points. However, at $N = 101$, we see this tread stop, and at $N = 211$, the error is at its lowest before fluctuating with larger N values.

Changed `fp = selected_real_kind(15)` to `fp = selected_real_kind(6)` in `utility.f90`.

Table 2: Precision: fp = `selected_real_kind(6)`, i.e. Single precision of at least 6 decimal places. (4-Byte IEEE Float)

Number of Points (N)	Error	Observation (if applicable)
21	9.82E-03	Largest error correlates to lowest N
41	8.11E-06	The rate of Δ Error decreasing has leveled out quicker than in Table 1
61	2.38E-06	
71	1.43E-06	This is the lowest error before increasing again
73	2.15E-06	
81	4.53E-06	
101	3.81E-06	

Our error is more substantial with fewer modes, but the plateau occurs sooner. The dominating error contribution is the lack of modes to flesh out our series solution. However, some rounding errors will still happen, causing a plateau.

Conclusion

Since N represents the number of modes, or terms, in our series approximation, we expect a large N to give less error. However, the computer has finite precision, resulting in rounding errors per mode used. Therefore, as we increase N, we ultimately reach a plateau and then fluctuation due to accumulated rounding errors. Several strange program-related issues occurred, as outlined in Methods, which would be a key point to revisit and revise. Moving forward, we aim to flesh out the theory and produce plots to compare the exact solution against our series representation $u(x)$.

Appendix

Materials

Table 3: Software

Name	Description
Visual Studio Code	Code editor
Fortran	Main programming language used (compiler based)
Linux	open-source Unix-like operating system
Python	high-level programming language - to display data produced by Fortran
Poetry	Dependency/Environment management tool
Git	open source software for distributed version control - to track code changes
Given Files	`makefile`, `bvp.f90`, `problemsetup.f90`, `utility.f90`, `dft.f90`, `bvp.f90`, `bvp.init`
Processed Files	`problemsetup.f90`, `dft.f90`, `bvp.f90` - supplemented with required functions/subroutines
`problemsetup.f90`	- Contains subroutines and functions specific to checking our solution $u(x)$ against the exact solution and outputting a `.dat` file. - Along with a subroutine to extract N from our `bvp.init` file.
`dft.f90`	Contains functions and subroutines that preform matrix-vector operations and fill the matrix T, Discrete Fourier Transformation Matrix
`bvp.f90`	Main program file which preforms all needed steps to solve the BVP.
`utility.f90`	Module: Contains declarations available to all other program/module files
`bvp.init`	Holds the number of points or modes to use - having the init file allowed for easy editing and data processing
`makefile`	Commands: `make, make data, make move, make clean` - executes/compiles the program, moves data files into a dir, moved obj files to a dir, removes unwanted files respectively

Bonus

Changed `fp` to have 30 decimal places of precision.

Table 4: Precision: fp = `selected_real_kind(30)`, i.e. precision of at least 30 decimal places.(16-Byte IEEE Float) Most Resolved Grid.

Number of Points (N)	Error	Observation (if applicable)
21	9.82E-03	Largest error correlates to lowest N
41	5.58E-07	
61	1.09E-11	
81	2.56E-16	
101	2.17E-21	
201	2.70E-33	This is not the smallest value possible it is the smallest in this set.
301	5.59E-33	

With a higher value of `fp` we see the leveling out of our error requires a substantial amount of points when compared to Table 1 and Table 2. Our error is $10E18$ time smaller than any other error seen i.e our solution is very nearly the exact solution. Or at least $10E18$ times closer! Pretty sick if you ask me.