PX390, Autumn 2019, Assignment 3

November 4, 2019

1 Intro

The purpose of this exercise is to further expand your knowledge of C and your ability to understand and test numerical code.

After half of the employees in your company have been forcibly deported, it is up to you to clean up the mess! One of them has written a code that is horribly broken and needs to be fixed. Your task is to read the specification, and the program, and to 'debug' the program: make it do what it is meant to do. Some of these problems with the code are basic c programming errors, and some are more subtle mismatches between the required and actual code behaviour.

You should submit a single C source file with a list of corrected bugs, at the top of the c source file (in comments), where I have added a placeholder. You may submit your code via the link in the assignment section of the moodle page.

The code must compile and generate no warnings when compiled with

```
gcc -Wall -Werror -std=c99 -lm
```

There are a finite number of bugs, but I'm not going to tell you how many: make sure that your code actually works by testing it rather than just looking for obvious errors. Note that the part that reads in the input parameters (last function in the file) is fine, so don't change this.

1.1 How is it marked

I'm giving marks for the absence of each bug (in an otherwise correctly working code). I'm also giving marks for the code compiling correctly. You may lose marks in more than one category for sufficiently severe problems. A correctly working code will contain a stable and consistent finite difference method for solving the equations below (I'm not looking for anything fancy).

1.2 Tips

1. The compiler is your friend. Start with the first warning/error messages and work through them until there are none left.

2. Some bugs are hard to catch just by inspecting the code. Adding printf statements is usually the easiest way to check that the code is doing what you think it is: check the maths at the first timestep, for example.

3. You can often catch bugs in numerical code by looking at the output graphically: does it do something wierd at the boundaries? A variety of tools (Matlab/matplotlib/Origin) exist which can take numerical output and plot it for you. This is one of the things you should learn while doing this assignment as it will be essential later on.

4. Learn how to use debuggers (gdb). Memory checking tools like valgrind can help catch issues to do with reading/writing into an incorrect memory location.

5. The boundary/initial conditions are a bit complicated: if you want to test that your code is correctly solving the equation, you can temporarily choose simpler ones with closed form analytic solutions.

6. Bugs include both incorrect lines of code as well as missing functionality. There are comments which are misleading: I'm not treating these as bugs, but you might like to fix them as you go.

7. The code as written attempts to use a 'time-splitting' method to resolve the coupling between U and V. This is actually a good idea, even if the implementation is wrong.

2 Specification

The code must use a simple, convergent, finite difference scheme to solve the coupled differential equation for real-valued functions U and V with

$$\frac{\partial U}{\partial t} + C\frac{\partial U}{\partial x} - V = 0 \tag{1}$$

$$\frac{\partial V}{\partial t} + C\frac{\partial V}{\partial x} + U = 0 \tag{2}$$

Note that this is equivalent to the equation for complex-valued $Z = U + iV$

$$\frac{\partial Z}{\partial t} + C\frac{\partial Z}{\partial x} + iZ = 0 \tag{3}$$

The equation is to be solved on a 1D x domain with $x \in [0, L]$, as an initial value problem. The domain length, grid size, length of time over which to solve and the coefficients are read in from a file: the function that reads this data (and the function prototype) is the only part of the code that is bug free (i.e. don't change this bit), but the way it is called may not be right.

The initial condition is $U(x, 0) = 1.0$, $V(x, 0) = 0.0$. There are 'nx' grid points, with the first grid point at $x = 0$, and the final point at $x = L = (\mathrm{nx}-1) \times \delta x$.

Boundary conditions at $x = 0$ (the left-hand-side of the domain) are $U = V = 0$.

2.1 Input

A file 'input.txt' is used for input: note that there is an example on the moodle page. The file contains the parameter C on the first line, then the domain length L, then the number of grid points nx, the simulation time t_F, then the output timestep t_o on the last line. You may assume all these inputs are positive (and the integers are not huge).

2.2 Output

The code outputs simulation data at a fixed interval in time, the output timestep (t_o): it should output the initial values (at t=0), and at $t_0, 2t_o, 3t_o$ etc. (but not necessarily the final value). The simulation timestep and output timestep need not be equal.

The time t, x coordinate, and U and V are written in that order in a single output line for each gridpoint. Please don't add extra comments/blank lines to the output, which should contain only numbers! Remember that I'll need to read the numbers, so if it is output in an unreadable format then you'll lose marks.