PX390 2021-22, Assignment 3

1 Intro

The purpose of this exercise is to further expand your knowledge of C and your ability to understand and test numerical code. You are given a code which should solve two coupled partial differential equations. The code is, however, broken and needs to be fixed. Your task is to read the specification, understand the program, and 'debug' it. At the end of this task the program should do what these specifications say it is meant to do. Some of these problems with the code are basic C programming errors, other 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 (this description will not be marked but helps both of us keep track of what you have changed). 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 -o assign3 assign3.c -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. Some errors may be repeated in multiple statements, or may effect multiple lines of code. Note that **the function reading in the input parameters** (its declaration and the definition) **should not be changed**. The use of the function (a call in main()) may or may not be correct.

1.1 How is it marked

Marks are given for the absence of each bug (in an otherwise correctly working code) and 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.

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 strange 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 given to you 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.

8. You can check the behaviour of this code in simple cases (where the splitting is not needed for stability).

9. It is expected that you will use malloc to allocated arrays on the heap and that the allocated memory is released (freed) after use.

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

$$\frac{\partial u}{\partial t} + D\frac{\partial^2 u}{\partial x^2} - v = 0 \tag{1}$$

$$\frac{\partial v}{\partial t} + D\frac{\partial^2 v}{\partial x^2} + u = 0 \tag{2}$$

Note that this is equivalent to the equation for complex-valued $z = u + iv$

$$\frac{\partial z}{\partial t} + D\frac{\partial^2 z}{\partial x^2} + iz = 0 \tag{3}$$

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 must not be changed, but the way it is called may not be right.

These equation are to be solved using Strang time step splitting technique, so that a separate solutions are found for the diffusive (spatial second-derivative) and the decay terms of these equations (third terms on the LHS of these equations). Let A_1 represent equations (1) and (2) with only the diffusion terms and

A_2 the same eqations with only the decay/growth term. A complete solution can then be obtained in three steps: (i) evolving subsystem A_1 with time step $\tau = 0.5dt$ then (ii) solving the subsystem A_2 with the time step $\tau = dt$ and, finally, (iii) solving subsystem A_1 again for time length $\tau = 0.5dt$. Recall that a solution produced at a given step is treated as a current profile for the next step.

The initial condition is $u(x, 0) = 1.0 + \sin(2\pi x/L)$, $v(x, 0) = 0.0$. There are nx grid points, with the first grid point at $x = 0$, and the final point at $x = L = (nx - 1) \times \delta x$. Periodic boundary conditions should be enforced, that is, $u(x = 0) = u(x = L)$ and $v(x = 0) = v(x = L)$ (the derivatives are also periodic).

2.1 Input

A file 'input.txt' is used for input, and there is an example file on the Moodle page. The file contains the parameter D on the first line, then the domain length L, then the number of grid points nx and the final simulation time t_F. 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 equal to the chosen time step dt: it should output the initial values (at t=0), and at $dt, 2dt, 3dt \ldots$ (but not necessarily the final value).

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