Project 1

Overview

The goal of this project is to write a simulation code that will numerically integrate the 1-d Korteweg-de-Vries (KdV) equation. The time integration should be performed using an explicit method (e.g., Runge-Kutta), and as a final step, the code should be parallelized usingOpenMP.

1 Organizational bits

Ideally, please work in groups of 2 - 3 students. Showing a history of at least some collaboration is regarded as a positive. It is up to you whether you want to create a shared submission (project report + code in git repo), or if you want to submit individual reports and possibly variations of the code (you could share, for example, initialization and output / visualization, but keep the main integration separate). Your final submission should include a description of the contributions of each team member.
Please develop your code in a git repository on github. You are welcome to use the same teams you have worked in recently, or form new teams (if you need help finding team members, let me know.) I will provide an assignment you can sign up for, just like the regular homework (but it'll be an empty repository to start.)
It is important to note that this project extends beyond just the coding part. The final project report is an important part of your submission, and it should include an introduction to the problem, details your numerical approach, a description of the code you wrote, how it is being used, and some sample results. For the parallelization part, you should examine parallel scalability and discuss the results.

2 Objectives

Creating a code that in fact solves the KdV equation is of course an important part of the project, but a significant part of my evaluation will be not just on the "it works in the end", but on following practices that we learned in class during development. In particular:

- Version management: set up git (I provided a github classroom assignment repo), and use it to keep a history of development. Versions being checked into git don't need to be perfect, in fact going small steps at a time is a plus, and bugs are normal, no need to get things working perfectly before checking them in.
- Build: Use a modern, portable build system (recommendation: cmake).
- Testing: You don't have to fully unit test everything, but it'd be good to have at least some automated tests for non-trivial parts, for example the part that implements the actual r.h.s. of the KdV equation. Using googletest would be a plus.

- It is up to you to choose language / libraries. It is certainly possible to do this project using Fortran and its built-in arrays if that is your preference. If you go the C/C++ route, this project can be done using just plain C arrays, but I'd recommend using C++ and xtensor's arrays.

- Documentation: When you are done with the project, please submit a final report. This should explain (briefly) what the project is about (including a bit on the math background), how to build and run the code, what its output is, how to make plots. It should include some test cases that you ran, including the results (plots). A movie would be a great. You can just write the report in LaTeX or Word, as a README.md file or use a github wiki.

- Parallelization: An additional goal is to use OpenMP (which we are about to learn about) to parallelize the code. However, this should also be done near the end of the project when everything already works. I'd like to see some timing and scaling analysis of the code on a machine that has at least a few cores.

- Your code should be well organized and structured. This means, in particular, not everything in one file and one big function, but split up into logical, simple pieces. Some comments will be helpful (but you don't have to go overboard). I can be picky on code aesthetics, the most important thing being consistency. Use capitalization consistently, properly indent the code, use filenames that make sense, etc. Reusing some existing code from previous classes is certainly an option where it makes sense.

3 KdV

The KdV equation is a partial differential equation, describing the time evolution of a spatially 1-d field u:

$$\partial_t u + \partial_{xxx} u + 6u\partial_x u = 0.$$

This equation describes, amongst others, shallow water waves, where u is the height of the water on top of the seafloor. The second term is "dispersive", ie., waves with different wavelengths move at different speeds. The third term is a nonlinear advection term, which leads to, e.g., the steepening of waves as they approach the beach (but the equation can't model actual wave breaking).
The equation is interesting mathematically as it has "soliton" solutions, long time stable nonlinear solutions that propagate and don't interact with each other. There are fancy methods of solving the equation analytically (inverse scattering transform), but our goal here is to solve it numerically for given initial conditions. Part of your task is to find suitable initial conditions for testing, which will require some literature research, or at least googling.

4 Discretization in space

In order to make this problem tractable on a computer, we need to discretize it, that is sample the value of the field u at a finite number of spatial locations. We will just use a uniform grid, such that $x_i = i\Delta x$, where where i is an integer index.
So our discretized field is $u_i \equiv u(i\Delta x)$.
Spatial derivatives are approximated using finite differences. For example, you should be able to easily convince yourself that

$$(\partial_x u)_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$

using Taylor expansion $u_{i+1} = u(i\Delta x + \Delta_x) \approx u(i\Delta x) + \Delta x\, \partial_x u|_{x=i\Delta x}$ and similar for u_{i-1}.
An approximation for the second derivative can be found as follows:

$$(\partial_{xx} u)_i = (\partial_x \partial_x u)_i \approx \frac{(\partial_x u)_{i+1/2} - (\partial_x u)_{i-1/2}}{\Delta x}$$

After substituting $(\partial_x u)_{i+1/2} \approx \frac{u_{i+1} - u_i}{\Delta x}$, one finds

$$(\partial_{xx} u)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

You need a third derivative,

$$(\partial_{xxx} u)_i \approx \frac{(\partial_{xx} u)_{i+1} - (\partial_{xx} u)_{i-1}}{2\Delta x}$$

where you still have to substitute in the expression for the 2nd derivative.
After plugging in the finite-difference approximations for the spatial derivatives into the KdV equation, you end up with only the time derivative left, ie., a system of ODEs in the u_i.

4.1 Periodic boundary conditions

We will use periodic boundary conditions in this project, that is we assume the fields satisfy

$$u(x + Ln, t) = u(x, t)$$

for all integers n. You can hence choose the domain to be just $[0 : L]$ and hence $\Delta x = L/mx$, where mx is the number of grid points.
The discretized equation can be expressed in general form as

$$\partial_t u_i = rhs_i(u_{i-2}, u_{i-1}, u_i, u_{i+1}, u_{i+2})$$

where the rhs_i is what's needed for the time integrator (see later). For example, calculating rhs_5 would require u_3, u_4, u_5, u_6, u_7 as input. However, near the boundary, e.g., for rhs_0, you need $u_{-2}, u_{-1}, u_0, u_1, u_2$, which is a bit of a problem since u_{-2}, u_{-1} are not inside the domain, so you don't have them. However, because of periodicity, these values are equal to values within the domain that you do have, and so you have to use these – I'll leave it for your group to figure out the details. It's worthwhile thinking about how these special cases near the boundaries can be implemented, and I encourage you to discuss your ideas with me when it comes to implementing this part.

5 Timestepping

I'm basically leaving it up to you to select an ODE integrator – as mentioned before, the spatial discretization basically reduced the PDEs to a system of ODEs. Your life will be easier with a standard prescribed timestepping scheme like RK4 (4th order Runge-Kutta).
For the more adventurous, an adaptive timestep explicit method would be something to look into – just be aware that you will have to invest slightly more effort when parallelizing the code.