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 the code should be parallelized usingOpenMP.

1 Organizational bits

Please work in groups of 2 - 3 students. Showing that you actually collaborated, including using a shared git repo, is worth extra credit of up to 0.5 points. However, every team member should submit their own project in the end. You can collaborate on picking out methods, and figuring out what needs to be implemented, but then everyone should implement the main part of the code, ie., the calculation of the r.h.s., independently (talking to each other is okay, literally copy&pasting the code is not). On other parts of the code, for example, initialization, output, the timeloop, etc., I would like to see you collaborate, and, if possible, share common code.
Please develop your code in a git repository maintained on fishercat. You can just set up your own git repo ("git init", see previous classes) yourself. If you want to use git to collaborate (which would be a great), I'll be glad to set up a shared repo and get you started on how different people can work on the same code at the same time – together with telling you about the pitfalls of doing so. I will be glad to set up a trac page for your group's project if you would like to make use of the additional facilities, like the wiki, bug tracking, etc – just let me know.

2 Objectives

Creating a code that in fact solves the KdV equation is of course the main goal of the project, but a large 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: setting up git, and using 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: at a minimum, the project should have a Makefile, for full credit, use the autotools, and make sure you can create a "release" ("make distcheck") in the end.
- Documentation: When you are done with the project, please submit a final report. This should explain (briefly) what the project is about, 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, or as a trac wiki page.

- Parallelization: An additional goal is to use OpenMP 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 fishercat or trillian.

- 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, for example from the struct vector work is certainly an option.

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 $u_i := u(i\Delta x)$.
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

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

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 timestep 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.