Math 233 Homework 2

Majerle Reeves (mreeves3@ucmerced.edu)

October 5, 2020

1. Create a new C++ project.

I've uploaded the code on Github and attached the code in the zip folder!

2. First, we need to represent the velocity field as a Continuous Function of 2 variables. Create a new class CF2 which contain a public virtual operator(double x,double y). Why are we using virtual?

We use a virtual operator because we are overriding the operator.

3. In your main file create two children classes velocityX and velocityY of the class CF2,which will be used to represent the two components of the velocity field. For each class define their operator to be some analytic function of x and y (they will be updated later on when the velocity field will be specified)

I moved these to the CF2 file and out of the main file. These can be found in the code.

4. We will also need to compute first and second order derivatives of the solution. In the grid2D class add two functions which compute the forward and backward x-derivatives at node n of the function. Your functions should throw errors message if the function has the wrong size. Why are we using the pointer? What should you do at the boundary nodes?

We want to use & because we will be updating the value of function so it's better to pass a reference to function rather than a copy of the values. The values at the boundary depend on the boundary conditions of our problem. For this problem the derivative at the boundary is 0. And all of the values surronding the boundary of the problem are also zero. This problem is very forgiving.

5. Write the same functions in the y-direction.

In the code!

6. Write the functions for the second-order derivates in both directions.

In the code!

7. Create a class ENOAdvection which contain a grid2D, the solution at tn(stored as vector<double>), and two pointers to the velocity field components. Should these parameters be private or public?

These should be private. We don't want the velocity field, the grid, or the solution to change willy nilly. This means if we want to see what these variables are we need functions to access them.

8. Create all necessary functions (in the solver class) to set the solver parameters.

In the code! I needed to create some new functions in my Grid2D as well ass ENOAdvection.

9. In ENOAdvection, create a function oneStep(double t) which advances the solution from tn to tn+1=tn+ delta t.

This is where we actually implement ENO method! I kind of broke up this problem into multiple lines and saved new variables because I didn't want to confuse myself.

10. Implement a function that output the solution into a specified .vtk file.

This is how we plot some pretty pictures. For the videos I did not save the solution at every time step because this would have been a lot but I saved the videos at every 100 times steps.

11. What is the exact solution at the final time?

To solve this we need to convert the PDEs to a system of ODEs. We will get the following system of ODEs.

$$\begin{aligned}\frac{dx}{ds} &= -y \\ \frac{dy}{ds} &= x \\ \frac{dt}{ds} &= 1 \\ \frac{d\phi}{ds} &= 0\end{aligned}$$

When we integrate we get that t = s and that $\phi = \phi_0(x_0, y_0)$. This leaves us to solve the coupled differential equation.

$$\begin{pmatrix}\dot{x} \\ \dot{y}\end{pmatrix} = \begin{pmatrix}0 & -1 \\ 1 & 0\end{pmatrix}\begin{pmatrix}x \\ y\end{pmatrix}$$

We can solve this by finding the eigenvalues and the eigenvectors. $\lambda = \pm i$ with eigenvectors: $\begin{pmatrix}-1 \\ i\end{pmatrix}, \begin{pmatrix}1 \\ i\end{pmatrix}$. Using Euler's identity and some math we get that the solution is:

$$\begin{pmatrix}x \\ y\end{pmatrix} = c_1\begin{pmatrix}cos(t) \\ sin(t)\end{pmatrix} + c_2\begin{pmatrix}-sin(t) \\ cos(t)\end{pmatrix}$$

We can then use the initial conditions to get that $c_1 = x_0$ and $c_2 = y_0$. We finally get the solution:

$$\begin{pmatrix}x \\ y\end{pmatrix} = \begin{pmatrix}x_0 cos(t) - y_0 sin(t) \\ x_0 sin(t) - y_0 cos(t)\end{pmatrix}$$

We now need to solve for x_0 and y_0 Through algebra we can calculate that $y_0 = xsin(t) - ycos(t)$ and that $x_0 = xcos(t) - ysin(t)$. We can then plug this into the initial solution:

$$\phi(x,y,t) = \begin{cases} 1 & \sqrt{(xcos(t) + ysin(t) - 0.5)^2 + (xsin(t) - ycos(t))^2} - 0.2 <= 0 \\ 0 & \text{else} \end{cases}$$

And this is the solution as a function of t. You can see that at $t = 2\pi$ the solution will be the same as the initial condition. I didn't show all the work here because it is a lot of messy algebra.

12. Taking delta t= 0.25 delta x, compute the numerical solution for N= 64, 128, 256, 512. Plot the final solutions in paraview (hint: you might find the warpbyscalar filter pretty useful) and comment on the convergence of the method (extra credit: make a movie of the solution evolution).

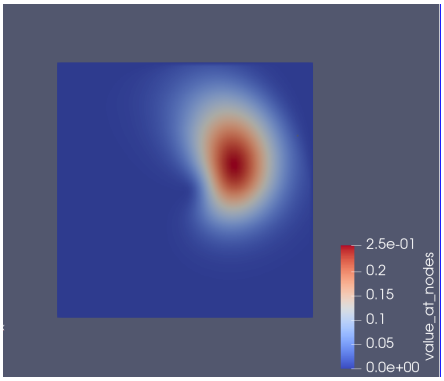

Figure 1: Grid Spacing of N = 64

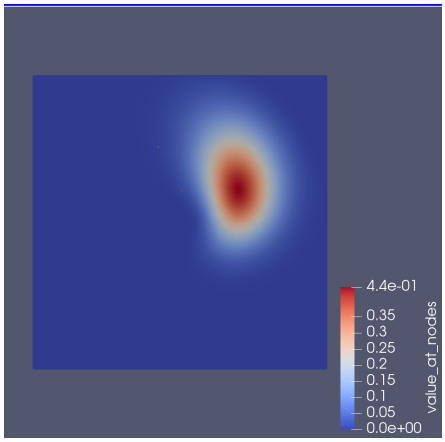

Figure 2: Grid Spacing of N = 128

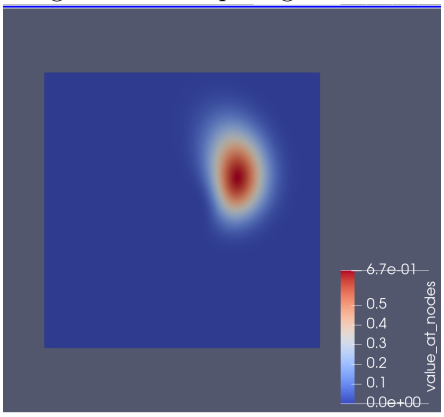

Figure 3: Grid Spacing of N = 256

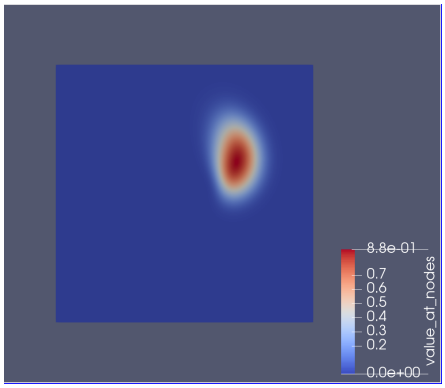

Figure 4: Grid Spacing of N = 512

We can see that as N increases we have less diffusion of the solution and more of the height is maintained. We can also see that it appears that the centroid of the solution goes a little bit past where the analytic solution should be. I think this is because we need to use a smaller delta t. Here are some plots where $\delta t = \delta x^2/$. I did not include the case where N=512 because that simulation is to computationally expensive.

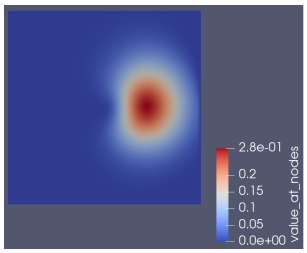

Figure 5: N = 64

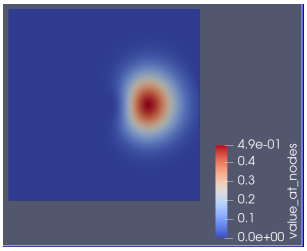

Figure 6: N = 128

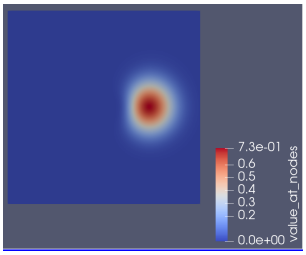

Figure 7: N = 256

The solution does look much better when we take a smaller time step. We see that now are solution is in the proper place and we also see less diffusion of the solution. Now lets talk about convergence. The first thing that I did was calculate the error. I used the 2 norm of the analytic solution minus the computed solution to calculate the error and then normalized this error by dividing by N^2. The following errors in the table are computed using $\delta t = \delta x^2$. I calculated the order using this equation:

$$\text{order} = \frac{ln[E(2h)] - ln[E(h)]}{ln([h]}$$

Grid	Error	Order
64	0.00220974	
128	0.000953314	1.21
256	0.000392398	1.28

Table 1: Spatial Order.

We expect the obtain first order in time and second order in space. However, when calculating the order in space we see that we are not quite able to achieve second order. We also were not able to see that first order convergence in time. We do however see marked improvement with using a smaller δt. Does this have anything to do with the CFL condition? Since we have a velocity field that varies with time is it possible that for every very small delta t we may not always be meeting the CFL condition for every point of the grid?

13. Write a function use the standard central differences formula for the second order derivatives (instead of ENO). Simulate the same problem, with the exact same parameters using this new method. Compare the results with the one obtained for the ENO scheme.

I honestly get really similar results for when I use central second order derivatives. I'm not sure why this is. I have included a video of the solution using central differences.

14. By now you should be confident that your code is working. Use openMP to parallelize it. Check that your still getting the same results. Do you see any improvement in the performances ?

The parallelization gives me the same results and really speeds up the run time. For N = 512 and delta t = 0.25 delta x this sped up my code from approx 8 minutes to 2 minutes. However, with a very small delta t (delta x squared) and a very fine grid it still doesn't speed it up enough to run quickly. We can only parallelize the grid steps and not the time steps because we don't want to have a data race.

15. Consider now for the initial contour, the reinitialized level set function representing the initial circular interface . Consider the same velocity field. Simulate the evolution of the interface, for N= 256 and tf= 2. Plot the interface at different timestep (hint: use the contour filter in paraview). Comment on both the interface position and the level set function. What are we missing to correctly simulate the evolution of ?

I have included the video of the interface. The interface is at the center of the plot. It looks like the interface travels in the same counter clockwise pattern as our previous simulation. The interface does not approach the edges of the plot.