SDS 335 Final Report

Member names: Joe Garcia, Danny Nguyen, James Sullivan

Group TACC IDs: jag7548, dnguy3n, jsull3

Introduction:

In order to gain insight into a fluid dynamical system via simulation we are required foremost to use a numerical technique to solve the problem. While a numerical method does not provide a true solution we can still find an appropriate approximation. We plan to use a finite difference method in order to approximate the function. In using a finite difference method, we need to generate a grid, discretize, and then solve simultaneous algebraic equations. There are various types of grid generations we can use. However, we will focus on a simple quad in two dimensions. We do this in two dimensions for simplicity, though may extend it to three or use some shape other than a box in the future. Next, we will discretize using the Taylor series approximation. Depending on the order of the derivatives in our dynamic equation, we will need to solve up to a certain order in the Taylor series expansion. For instance, if a term in the equation required solving a second order derivative then we will need a second order Taylor expansion. We will for the most part ignore higher order terms, but still expect to observe reasonable results for a simulation based in Taylor approximations in this manner. Another important factor to consider is initial conditions. If we have non-linear equations governing the motion then any change in the starting parameters can lead to completely different behavior by the system.

Setup:

Among these initial conditions, we ended up choosing three that gave a range of different challenges for the code. We included options for a simple single flow source at the top left of the grid, random sparse diagonal elements, and an option with ten percent of the grid randomly filled with flow values. We originally had included an option that modeled a source of flow at the center of the grid, but ended up performing a trade of physical viability for code versatility. As a central source of flow is much more physically likely than say, a random distribution of flow values, we found that initial condition option to be too similar to that of the corner flow, and the random distribution gave an additional challenge for the code when values were not nice and predictable.

The other major set of options we have for setup are the grid options. We have a simple square setup option, and a simple rectangular setup option. We also originally included a triangular grid, but found that this was unlikely to give us any new insight into solving the problem, and was essentially just as good as using a square grid with the grid refined to twice the original resolution.

Other setup considerations included the size of the “world matrix”, which was n by n and where the initial conditions were declared, as well as the degree to which we wanted to sample the world matrix and use those monte-carlo samples to setup the grid matrix, which is encoded in the dimensions of the grid matrix. Finally, we also included a viscosity coefficient that was of an order that was convenient for calculation. It may be worth exploring in future work how different viscosity coefficients affect the results of the code for other partial differential equations, as ours only takes parabolic partial differential equations into account.

Methods & Algorithms.

Written in C, our project consists of a number of files that are specific to a method. For example, there are multiple sampling techniques and grid layouts that are each written as their own file. Our main program file is “problem.c” which wraps up the initial conditions, grid initialization, monte-carlo sampling, Navier-Stokes solving, each of which has its own function call within main() in problem.c. On a side note, the figures in the results section were generated using Python's matplotlib as a simple way to create visuals of our data.

To begin, the "world" matrix is initialized with values at different elements. Then either a simple or systematic sampling method was applied to the world matrix. Both are essentially the same method except systematic sampling samples at every k element from the total number of elements in the sample size (in our case it had a stride of 10). A smaller matrix, the grid, is then considered as what will be measured. The values sampled from the bigger matrix are then used to initialize the grid. Essentially, the grid was placed on top of the n by n matrix and each grid element contains a number of cells or points from the bigger matrix. The sampling occurs among these points and produces the value for the grid element. If a value was not found from the sample method then the average across all the points in the grid element is used to give a grid value. Areas of the grid in which the cells did not have many initial conditions resulted in little to no value for that grid element. Now, the grid has its own initial conditions based from sampling and is ready to run. The simulation was ran at different number of seconds in order to understand the difference time makes in the evolution of the system. Different initial conditions were also given in order to demonstrate the various behaviors that exist from different inputs.

We used the explicitly discretized Navier-Stokes equations for unsteady viscous flow evaluated at equidistantly spaced points in our grid. This is a parabolic partial differential equation that is second order in space and first order in time. We can use the forward and backwards difference methods discussed for computing derivatives in class to discretize the spatial derivatives of the Taylor approximations to the function of interest at the central point of the grid (e.g. a simplified equation of state or an averaged velocity of flow value.). Using these discretized derivatives, we can find the discretized second spatial derivatives for each coordinate direction, which will give us something like the discretized laplacian in two dimensions. We can prescribe the timestep to account for the change in time for the time derivative in the PDE, and the only remaining unknown is the Taylor approximation after it has been evolved one timestep, which is the quantity we desire.

While problem.c was the main script which initialized and produced results, it was the nv_solver.c file that did most of the computations. Once the grid was initialized it was put into the solver in order to approximate the next grid element. The setup of the solver was one time loop and inside that time loop was a double loop that accessed each index in the grid. So, at time t+0 every index was updated then after going through all indices the time was updated and the indices were recalculated to get the next approximation. Calculating index i,j was done somewhat similar to the traditional Euler method where the approximation is the current value plus some other value. This “other” value is composed of either the forward, backward, or central difference method depending on where i,j is located in the grid. If i,j is in the first column then only a forward difference method can be used whereas the last column corresponds to using a backwards difference method. Because we were working in two dimensions, the difference methods were considered in terms of moving forward/backward in x or y. Indices not on an edge utilized the central difference method. In nv_solver.c there are multiple conditional statements that perform the difference calculations depending on which indice the loop is currently at. The x and y values are calculated then the average of those two values are taken in order to generate a new point for the grid index.

Results and Discussion:

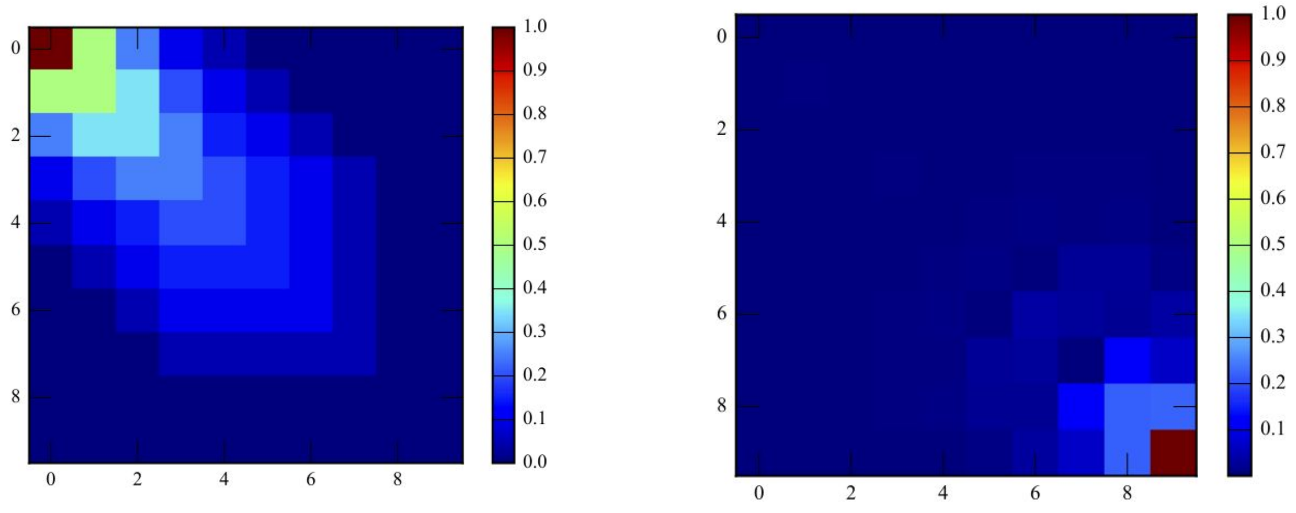

Figure 1: The matrix representations of the post-solver output for the top-left corner point concentrated matrix

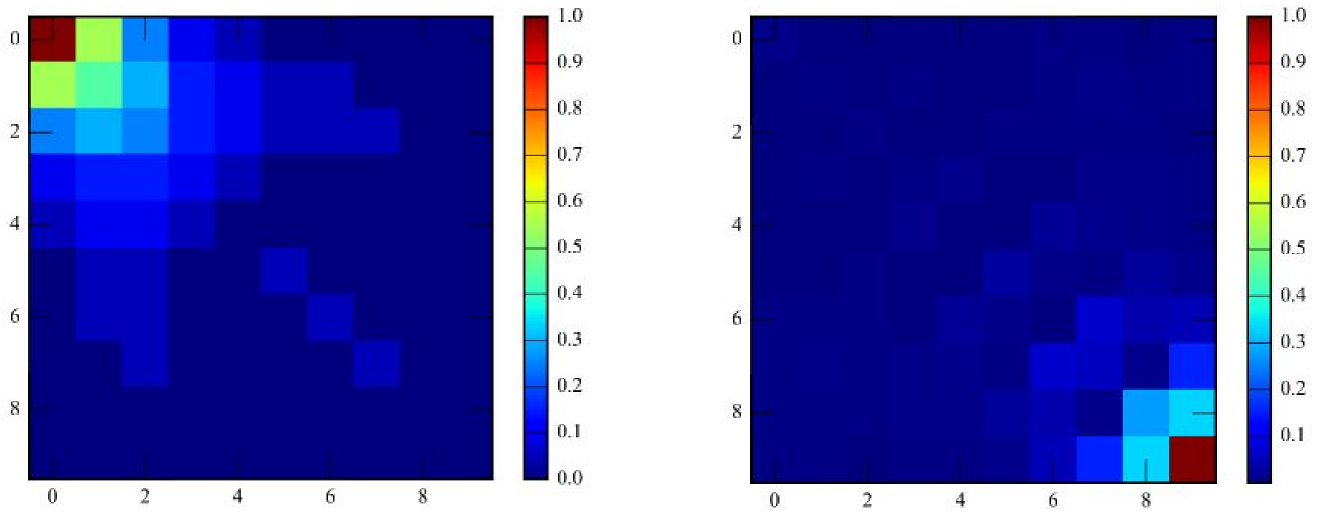

Figure 2: The matrix representations of the post-solver output for the sparse diagonal randomly valued matrix

Taking a look at a subset of our results, you can see in our figures the output of the program after the Navier-Stokes solver has acted on the matrix. We have chosen to display the simple case of the quad grid setup with a relatively low resolution of 1000 x 1000 for the world matrix, and a 10 x 10 grid matrix with each element containing a 100 x 100 chunk that gets sampled. In both Figure 1 and Figure 2, the leftmost image is a visualization of the output after the Navier-Stokes solver function has taken one time-step iteration, and the rightmost image is a visualization of the output after the solver has taken fifty time-step iterations. Taking many more timesteps results in non-significant departures from these values, and eventually overflow errors, so we figured fifty was a good stopping point.

We plotted the matrix visualizations in python using matplotlib after reading in the data that our main program outputs to the “output.dat” file. The colorbars express a linear relative weight between the maximal value of 1 (red) and the minimal value of 0 (dark blue), and represent the velocity values of flow in our square grid. The x and y axes are the grid coordinates for the monte-carlo sampled chunks.

Notably, our results actually match pretty well with what we’d expect just from physical intuition of these processes, even though we are far from experts in parabolic unsteady viscous flows. Beginning with the more easily approachable top-left corner matrix, Figure 1 illustrates the progression from a point concentration to a more diffuse one even after one timestep. It’s apparent from the figure that the concentration of flow in the upper quadrant of the grid spreads outwards as the point flow spreads toward the rest of the box, and appears to approach a state where the flow has moved to the opposite corner of the box.

In the case of Figure 2, the situation is a bit less transparent, but we would expect a diagonally distributed flow to move faster toward the bottom-right corner of the box due to the additional diagonal flow. We can see that the near diagonal configuration in the leftmost part of

Figure 2 exhibits behaviour similar to that of the Figure 1 configuration. We say “near diagonal” specifically, because one timestep has already occurred, and middle values of the matrix have already changed up to three times. Interestingly, there appears to be a sort of wave pattern in the Figure 2 final matrix visualization not present in the corresponding Figure 1 matrix visualization. The Navier-Stokes solver doesn’t explicitly account for reflection off of boundaries, but it is possible that the finite difference scheme and boundary conditions are enough to exhibit this behavior for the faster flow scenario of the diagonal initial condition.

While our original plan was to model diverse physical systems, like stars and plasma, we have still produced a useful tool which may be used to at least intuitively and quantitatively understand viscosity and simplified Navier-Stokes equations. Future work could include writing physical system scenarios, or including solvers for different families of partial differential equations, or even simply adding back in terms of the full Navier-Stokes equations. On the programming side, the code could be better optimized via parallelism, and convenient features to add would be restart capability and a better graphical output, ideally a dynamic one.