Projection Operator on Time Dependent Navier-Stokes Equations

I. Chang, J. Lee, R. Johnson, A. Wong

December 17, 2011

1 Abstract

We solve the time dependent Navier Stokes equations $\frac{\partial u_d}{\partial t} + \sum_{d=0}^{D-1} \frac{\partial u_d u'_d}{\partial x'_d} = -\frac{\partial p}{\partial x_d}$ for a periodic velocity field using the projection operator as described in Colella's paper. We implement the Runge Kutta method of the fourth order to solve the ordinary differential equation and discretization schemes that we discuss below. To test our code, we ran it on a jet in a doubly periodic boundary for the Euler conditions, the same as the the example in Bell, Colella, and Glaz. We planned to use this data as a control on the data we received and to test the accuracy of our code.

2 Mathematical Framework

Applying a projection operator on a vector field $\vec{V}$ allows us to interpret the Navier Stokes equations as an evolution equation for velocity within the space of divergence-free vector fields. We test the projection method with the following divergence-free velocity field.

$$u = tanh(y - 0.25)/\rho) \text{ for } y \leq 0.5$$
$$u = tanh(0.75 - y)/\rho) \text{ for } y \geq 0.5$$
$$v = \delta sin(2\pi x)$$

3 Discretization Methods

3.1 Projection Operator

The projection operator involved two sub-operations: divergence and gradient. The divergence is defined on a scalar single-dimensional MDArray based on the input FieldData (which is two-dimensional). The value of each element is dependent on the surrounding element, and both dimensions are taken into account when figuring out the scalar value

of a single point. The gradient is stored in a two-dimensional DeltaVelocity object which depends on the values of the scalar [ghosted] MDArray representing the divergence (after being solved through the Poisson Solver). The x-axis values of the gradient is obtained by consulting the two neighboring cells (on the left or right) of the cell in question. Similarly, the y-axis value is obtained by consulting the cells above and below the cell in question. Finally, to calculate the projected value, each dimension of the gradient is subtracted from the respective initial velocity field.

$$\mathcal{P} = \mathcal{I} - grad(\Delta)^{-1} div$$

3.1.1 Divergence Operator

$$div^h(u^h)_i = \frac{1}{2h}[(u_0)_{i+e^0} - (u_0)_{i-e^0} + (u_1)_{i+e^1} - (u_1)_{i-e^1}]$$

3.1.2 Laplacian Operator

$$\Delta^h(\phi^h)_i = \frac{1}{4h^2}(-4\phi_i + \phi_{i+e^0} + \phi_{i-e^0} + \phi_{i+e^1} + \phi_{i-e^1})$$

3.1.3 Gradient Operator

$$grad^h(\phi^h)_i = \frac{1}{2h} < \phi_{i+e^0} - \phi_{i-e^0}, \phi_{i+e^1} - \phi_{i-e^1} >$$

3.2 Advection Operator

The advection operator is discretized in the following way.

$$\frac{\partial}{\partial x_d}(u_d\vec{u}) \approx \frac{(u_d)_{i+\frac{1}{2}e^d}(\vec{u})_{i+\frac{1}{2}e^d} - (u_d)_{i-\frac{1}{2}e^d}(\vec{u})_{i-\frac{1}{2}e^d}}{h}$$
$$\vec{u}_{i+\frac{1}{2}e^d} = \frac{1}{2}(\vec{u}_i + \vec{u}_{i+e^d})$$

3.3 Runge-Kutta Method

The Runge-Kutta method is an iterative method to approximate the solutions of ordinary differential equations. Given a ordinary differential equation, we can solve for it in the following way.

$$y' = f(t, y)$$
$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), t_{n+1} = t_n + h$$
$$k_1 = hf(t_n, y_n)$$
$$k_2 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$$

$$hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2)$$
$$k_4 = hf(t_n + h, y_n + k_3)$$

3.4 Poisson Solver

The Poisson Solver takes in the input data in MDArray and applies the Forward Fast Fourier Transform to the data. It then divides the coefficients resulted from the transformation by the symbol operator. Also, it set the coefficient to zero where the low corner of the grid was defined. The solver then takes new coefficients and applies the Inverse Faster Fourier Transform and normalize them by squared of the mesh spacing.

$$(\Delta^h f)_j = \frac{1}{h^2}(-4f_j + f_{j+(1,0)} + f_{j+(-1,0)} + f_{j+(0,1)} + f_{j+(0,-1)})$$

3.5 Compute Euler RHS

Computing the Euler equation requires two major operators: Advection and Projection. It first increments the velocity field data and applies the Advection Operator to the incremented data. The result then gets multiplied by the input (Δt) before adding to the input field data. The projection operator was applied to the new field data to compute the difference between the projected velocity data and input data.

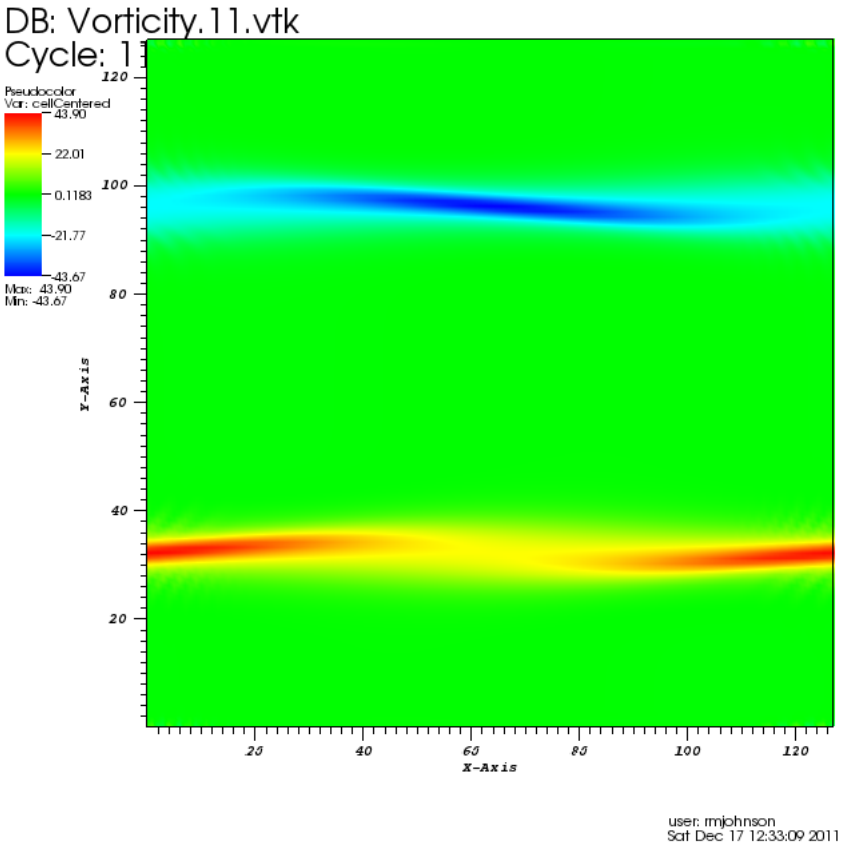
4 Progress

Projection is tested through ProjectionTest which uses non-divergent velocities. This meant our test should have produced a constant value of 0 for all points in the grid, which it did. Admittedly, this is not a thorough test, but it does confirm that Projection behaves as it should in this subset of operations. The advection operator is tested through AdvectionTest, which applies the operator to our initial data set. We used the plot on visit to verify that our operator is functional.

Currently, our driver (EulerTest) ends up with an error after the first step. We have tracked the problem down to PoissonSolvers h value. h represents the mesh spacing (1/N) and is used in modifying the Fourier coefficients and to normalize the real portion of the solution after inverting the Fourier coefficients. We have found that by manually modifying the value of h, we were able to achieve longer simulation. This only occurred for values smaller than the norm (1/N). We noticed this from a curious difference between the behavior of our OSX versions and Linux versions of the test. The value of h was defiend as: $h = \frac{1}{m_N}$, which OSX knew to upgrade to floating point arithmetic, but Linux left as integer arithmetic assigned to a floating point variable. this meant the Linux machines

were effectively working with h = 0 and the simulation ran for a longer period of time while the OSX machines failed after the first time step.

When running with smaller values of h (on the order of 1e-4), the simulation produces seemingly incorrect results. The streams seem to twist a bit, but quickly taper off and disappear. This behavior is demonstrated below, at the 11th time step in our computation.



5 Computational Experience

While doing this project, we learned how to work with a group of people on a modularized piece of software. We were able to break up pieces of this project and split them up basically based on the different operators mentioned above. Using svn allowed us to maintain currency, in most cases with one another while not working each other directly. We also learned about debugging one another's code. We found that when working on any piece of code that is dependent on others, you will find bugs in places that you havent touched and have to solve them yourself.

Further, we learned a lot about debugging more generally. It is favorable to keep as

much possible modularized so that we could test methods on their own. This allowed us to verify that some parts of our software were running correctly, allowing us to rule them out when searching for bugs in the fully assembled project. We also gained experience in debugging using gdb, which was essential to discovering the majority of our errors. This was especially effective for seg-faults, which are difficult to find without a back trace. The more difficult errors we had were those that allowed our tests to run, but returned unexpected data. These bugs do not announce their location, but instead force you to walk through code, frequently checking visit to see what changes have occurred.

The data structures that we used for this project were FieldData and DeltaVelocity. These are multidimensional MDArrays that store both X and Y components of velocity for a group of particles. Within these structures we defined methods such as fillGhosts, and increment that are used to operate on the structures within other methods. This was helpful because we only had to write the code for any of these operations once, and while debugging it limited the number of locations we could have errors. Understanding the data structures and the methods they implemented was something that our group struggled with while creating our code. We frequently went around methods like tupleIndex in MDArray, which allows you to iterate over the values in an MDArray much easier and which is less prone to bugs.