ENAE685 Final Project

Solving the 2-D Navier-Stokes Equations

Matt Kennedy

May 20, 2019

ABSTRACT

In this project the Navier-Stokes equations are solved for compressible flow, including shear stresses and heat fluxes to the previously solved Euler Equations. The solutions are applied to a flat plate in a supersonic flow at varying incident angles, as well as Couette flow with varying pressure gradients. Solvers are developed in both MATLAB and CUDA C.

TOPIC OVERVIEW

While I initially set out to explore 2-D flow through a nozzle, I had trouble finding interesting ways to expand the analysis and instead turned my interest to solving the full Navier-Stokes equations. After some initial troubles getting the CUDA compiler to install on an old Linux machine, I did finally get a working version of the base case (flow over top of thin flat plate) working in CUDA. The CUDA version runs only moderately faster than the MATLAB version, however further optimizations can be done and I expect that the CUDA version will scale much better as the grid size is increased from 70x70. All plots and analysis in this report were generated with the MATLAB version, although the same results are expected with trivial changes to the CUDA version.

THEORY

For this project the 2-D Navier-Stokes are solved for compressible flow, integrating shear stresses and heat fluxes to the previously solved Euler Equations. The 2-D Navier-Stokes equations in their conservative form are:

Continuity:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\partial v) = 0$$

x-Momentum:

$$\frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial x}(\rho u^2 + p - \tau_{xx}) + \frac{\partial}{\partial y}(\rho uv - \tau_{xy}) = 0$$

y-Momentum:

$$\frac{\partial}{\partial t}(\rho v) + \frac{\partial}{\partial x}(\rho uv - \tau_{xy}) + \frac{\partial}{\partial y}(\rho v^2 + p - \tau_{yy}) = 0$$

Energy:

$$\frac{\partial}{\partial t}(E_t) + \frac{\partial}{\partial x}[(E_t + p)u + q_x - u\tau_{xx} - v\tau_{xy}] + \frac{\partial}{\partial y}[(E_t + p)v + q_y - u\tau_{yx} - v\tau_{yy}] = 0$$

Where E_t is the sum of kinetic and internal energy,

$$E_t = \rho e + \frac{1}{2}\rho(u^2 + v^2)$$

The shear and normal stresses, τ_{xx}, τ_{yy}, τ_{xy}:

$$\tau_{xy} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)$$

$$\tau_{xx} = \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + 2\mu\frac{\partial u}{\partial x}$$

$$\tau_{yy} = \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + 2\mu\frac{\partial v}{\partial y}$$

Where λ is the second viscosity coefficient estimated by Stokes to be:

$$\lambda = -\frac{2}{3}\mu$$

and, assuming a calorically perfect gas, Suutherland's law can be used to get the local μ:

$$\mu = \mu_0\left(\frac{T}{T_0}\right)^{3/2}\frac{T_0 + 110}{T + 110}$$

The heat flux terms q_x and q_y can be determined from Fourier's law of thermal conduction which relates the thermal conductivity, k, and the temperature gradient:

$$q = -k\nabla T = \left(-k\frac{\partial T}{\partial x}, -k\frac{\partial T}{\partial y}\right)$$

Thermal conductivity can be made a function of Temperature through assuming the Prandtl number is constant at ≈ 0.71:

$$k(T) = \frac{\mu(T)c_p}{\mathrm{Pr}}$$

There are now two flux vectors, F and G, corresponding to the fluxes in the x- and y-directions, in addition to the conserved variable vector Q:

$$F = \begin{bmatrix} \rho u \\ \rho u^2 + p - \tau_{xx} \\ \rho uv - \tau_{xy} \\ (E_t + p)u - u\tau_{xx} - v\tau_{xy} + q_x \end{bmatrix}$$

$$G = \begin{bmatrix} \rho v \\ \rho uv - \tau_{xy} \\ \rho v^2 + p - \tau_{yy} \\ (E_t + p)v - u\tau_{xy} - v\tau_{yy} + q_y \end{bmatrix}$$

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E_t \end{bmatrix}$$

For our cases, the gravity component of the Navier-Stokes has been ignored.

PROGRAM

The MATLAB program consists of a main file, *initialize.m*, ten functions which get called every iteration, and one function to calculate the analytical boundary layer thickness to assist in establishing an appropriate domain.

Chapter 10, Supersonic Flow over a Flat Plate: Numerical Solution by Solving the Complete Navier-Stokes Equations from Anderson was loosly followed.

MacCormack's method was used as a simple but fairly robust iteration scheme for 2-D flow. Uniform grid spacing was also selected for simplicity sake, and while this limits the application cases to flow over flat plates or walls (rather than say, an airfoil,) this is still sufficient for initially exploring the Navier-Stokes solutions.

With the plan of porting this code over to CUDA C/C++ from the start, functions within the main iteration loop were designed to operation on a single point at a time such that $(JMAX * KMAX)$ threads could be launched and operate on each point independently.

The boundary condition application was separated into different functions because for cases where the wall goes all the way to $j = 1$ or $j = JMAX$ on the left or right side, the inflow or outflow boundary needs to be evaluated prior to the wall values being updated with the inflow or outflow conditions. When all points are being evaluated simultaneously there is no way to enforce this order so it was split into two separate operations with the threads syncing in between.

Iteration Scheme:

MacCormack's scheme was implemented as follows:

$$\text{Flux\_Predictor}_{j,k} = \frac{\text{F}_{j+1,k} - \text{F}_{j,k}}{\Delta x} + \frac{\text{G}_{j,k+1} - \text{G}j,k}{\Delta y}$$
$$\text{Q\_Predictor}_{j,k} = \text{Q}_{j,k} - \Delta t(\text{Flux\_Predictor}_{j,k})$$

F_Predictor and G_Predictor are then calculated from the values of Q_Predictor.

$$\text{Flux}_{j,k} = \frac{\text{F\_Predictor}_{j,k} - \text{F\_Predictor}_{j-1,k}}{\Delta x} + \frac{\text{G\_Predictor}_{j,k} - \text{G\_Predictor}_{j,k-1}}{\Delta y}$$
$$\text{Q}_{j,k} = \frac{1}{2}\left[\text{Q}_{j,k} + \text{Q\_Predictor}_{j,k} - \Delta t(\text{Flux}_{j,k})\right]$$

Time Step:

Per Anderson, the time step size was defined as:

$$\overline{\Delta t} = \left[\frac{|u_{j,k}|}{\Delta x} + \frac{|v_{j,k}|}{\Delta y} + a_{i,j}\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}} + 2\overline{v}_{j,k}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)\right]^{-1}$$

where

$$\overline{v}_{j,k} = \max\left[\frac{\frac{4}{3}\mu_{j,k}(\gamma\mu_{j,k}/\mathrm{Pr}}{\rho_{j,k}}\right]$$
$$\Delta t = \min\left[K\overline{\Delta t}_{j,k}\right]$$

Boundary Conditions:

The boundary conditions are described for each specific case in the sections below.

Code Layout:

```
initialize.m
├─ BoundaryLayerThickness()
└─ Begin Iteration:
   ├─ calc_dt()
   ├─ calc_Q()
   ├─ calc_FG()
   │  ├─ heatFluxParameters()
   │  └─ shearParameters()
   ├─ MaccormackPredictorUniform()
   ├─ primativesFromQ()
   ├─ enforceBC_nonSurface()
   ├─ enforceBC_surface()
   ├─ calc_FG()
   │  ├─ heatFluxParameters()
   │  └─ shearParameters()
   ├─ MaccormackCorrectorUniform()
   ├─ primativesFromQ()
   ├─ enforceBC_nonSurface()
   └─ enforceBC_surface()
```

APPLICATION: Flow over a flat plate

Supersonic Flow Over Upper Surface, Zero Incident Angle

The simplest case of flow over a flat plate is only considering the flow over the upper surface.

The plate is considered to be infinitely thin, with a stagnation point at the leading edge point, (j,k) = (1,1). The left-hand side inflow boundary is fixed to freestream conditions with $u = u_\infty$ and $v = 0$. The top boundary is assumed to be sufficiently far away from the shockwave to avoid any interactions, and is also set to freestream conditions. The right outflow boundary is allowed to be free, and values are interpolated from the two points to it's interior.

Figure 1 below shows the basic system where a shockwave is expected to form from the leading edge of the plate when the incoming freestream flow is supersonic.

Figures 2 through 5 below depict the results, showing the pressure and temperature ratios along the plate itself, the pressure and temperature ratios of the entire calculated flow field, and a quiver plot depicting the u and v velocity vectors. All results are as expected, and we can now move on to examining additional systems with this code.

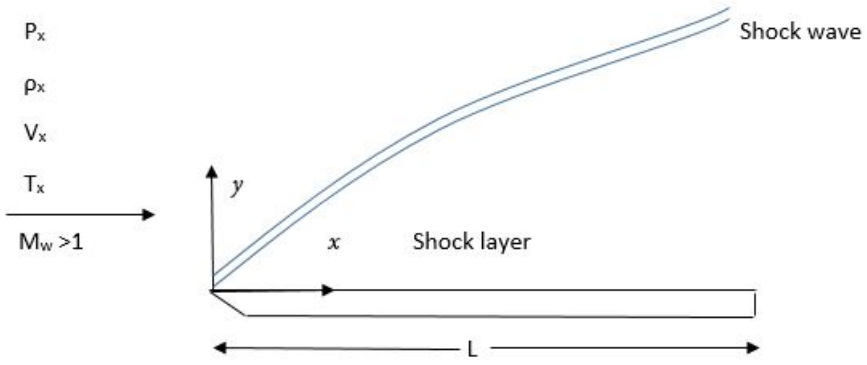

Figure 1: Supersonic flow over an infinitely thin flat plate, from Wikimedia Commons [2014].

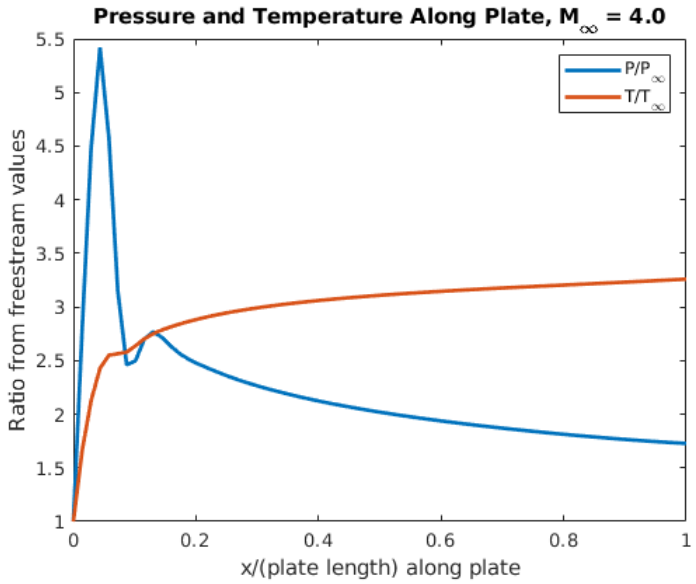

Figure 2: Pressure and Temperature ratio along the surface of the plate, where inflow is coming from the left and outflow to the right.

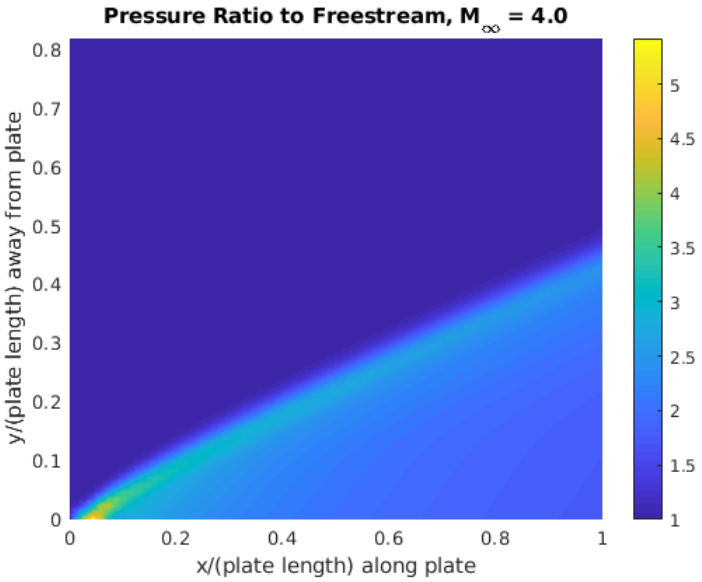

Figure 3: Contour of p/p_{atm} over the entire grid, with freestream flow moving from the left boundary to right and the flat plate fixed to the bottom boundary. The shockwave is visibe in lighter colors, compared to the dark blue being freestream conditions.

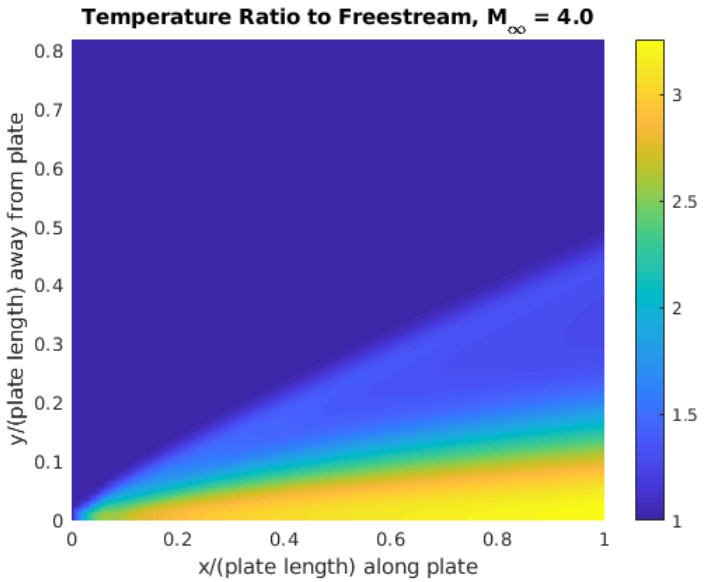

Figure 4: Contour of T/T_{atm} over the entire grid, with freestream flow moving from the left boundary to the right and the flat plate fixed to the bottom boundary. The shockwave is visible in lighter colors, compared to the dark blue being freestream conditions.

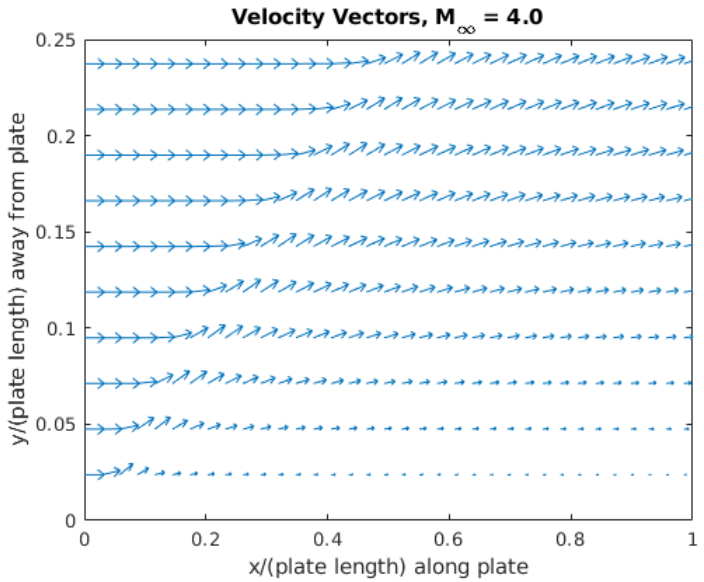

Figure 5: Quiver plot of the u and v velocities at each point, zoomed in around the bottom boundary to highlight the velocities through the shockwave.

Flow Over Upper and Lower Surfaces

Compared to the first case where only the upper surface was calculated, what if now both the top and bottom are considered simultaneously? This is a simple extension to the above code, with only small changes being made to the boundary conditions, and the grid being doubled in size in the y-direction.

Compared to before where the plate was at the lower boundary, it is now moved to the center of the grid in the y-direction. The bottom boundary condition now receives the same conditions as the inflow and top, in that v is enforced to be zero, and the other parameters are set to freestream conditions. At the "infinitely thin" plate, two cells in the vertical direction need to be allocated in order to track the pressure and temperature at the top and bottom surfaces independently. The u and v velocity are both set to zero on the plate as before to enforce the no-slip condition.

First lets examine flow around the plate at zero angle of attack, as seen below.

In Figure 6 the pressure and temperature profiles for the upper and lower portions of the plate; as the system is symmetrical, we expect the profiles to be identical which they appear to be.

In Figures 7 through 9 the flow is further visualized and the solution over the lower plate appears to be symmetrical with that over the upper plate.

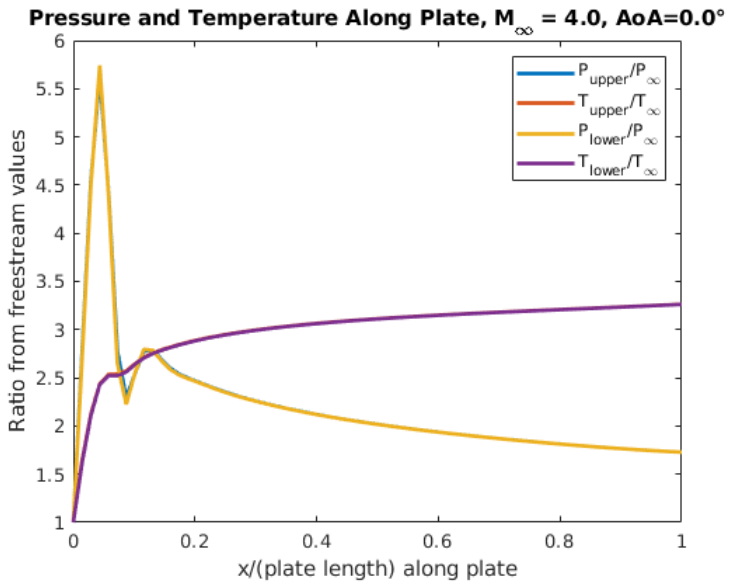

Figure 6: Pressure and Temperature ratio along the surface of the plate, where inflow is coming from the left and outflow to the right.

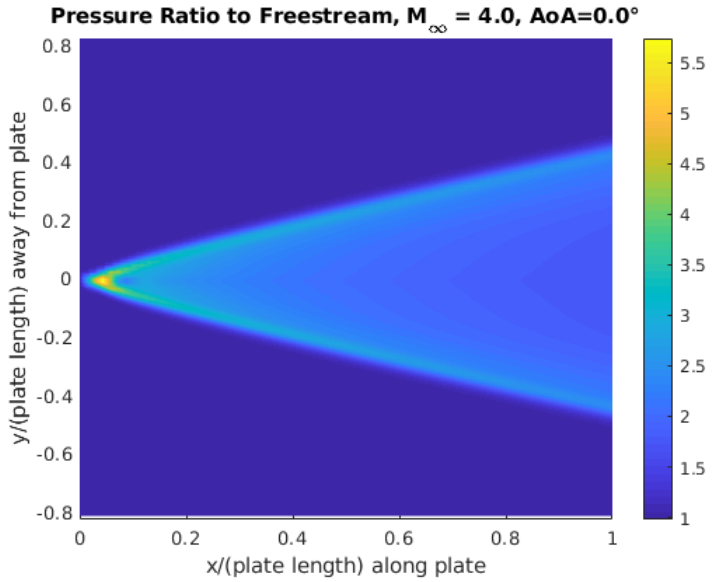

Figure 7: Contour of p/p_{atm} over the entire grid, with freestream flow moving from the left boundary to right and the flat plate fixed to the bottom boundary. The shockwave is visibe in lighter colors, compared to the dark blue being freestream conditions.

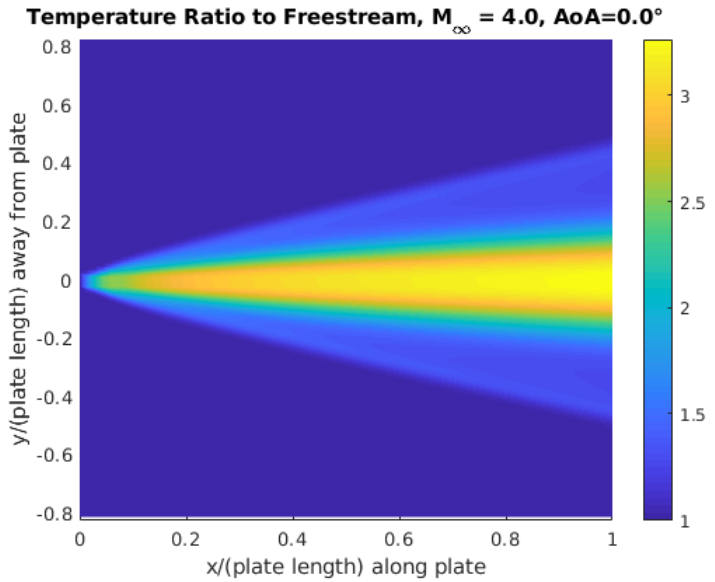

Figure 8: Contour of T/T_{atm} over the entire grid, with freestream flow moving from the left boundary to the right and the flat plate fixed to the bottom boundary. The shockwave is visible in lighter colors, compared to the dark blue being freestream conditions.

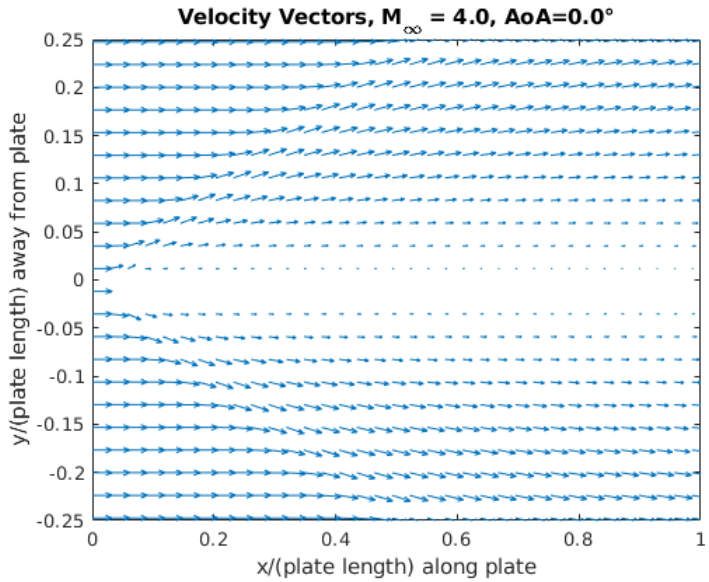

Figure 9: Quiver plot of the u and v velocities at each point, zoomed in around the bottom boundary to highlight the velocities through the shockwave.

But now how does the flow change when the plate is set at an angle of attack, rather than zero incident angle? This is applied by setting a non-zero value for the v velocity at the left and bottom (inflow) boundaries, where:

$$v = u \tan(\alpha)$$

for angle of attack α.

This flow profile is seen below in Figure 10, where as the angle of attack increases, the shockwave becomes less linear and the maximum pressure at the leading edge increases significantly.

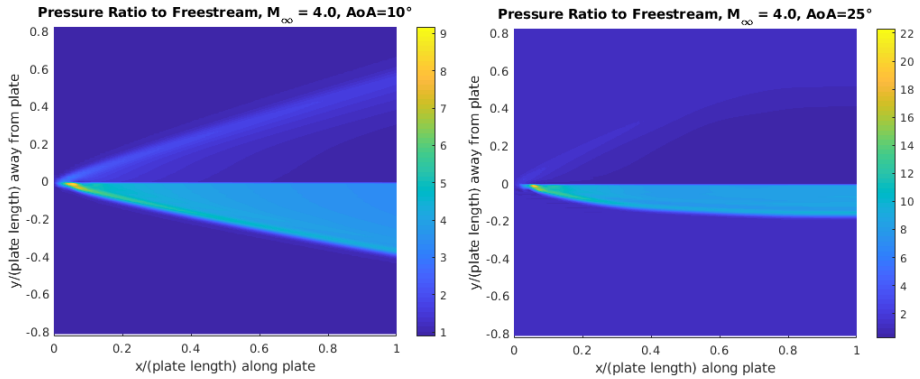

Figure 10: Contour of p/p_{atm} over the entire grid for angle of attack at 10 degrees on the left, and angle of attack of 25 degrees on the right. At higher angles of attack, the v velocity begins to deform the shockwave, and the maximum pressure becomes much higher at the leading edge.

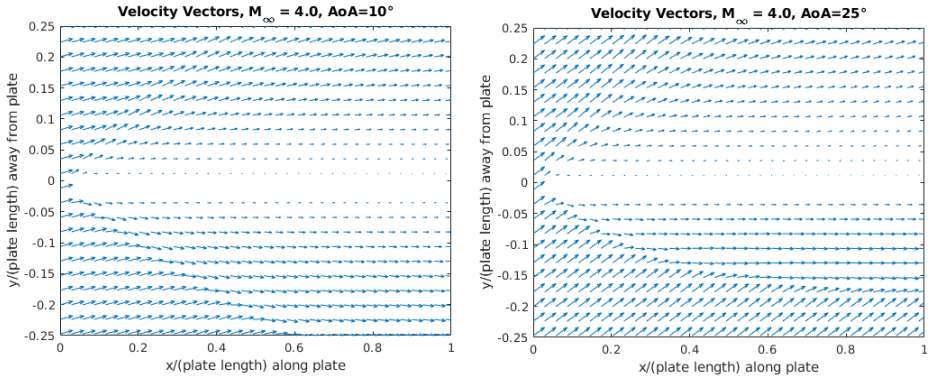

Figure 11: Quiver plot of the u and v velocities at each point, zoomed in around the plate boundaries to highlight the velocities through the shockwave.

As the angle of attack is increased the temperature profiles in Figure 12 remain similar, but the pressure values quickly diverge with the pressure under the plate

being greater than that on top of the plate. This is what creates lift!

Integrating the pressure along the top and bottom of the plate and taking the difference, a lift per unit length (into the page) can be calculated, as shown below in Figure 13.

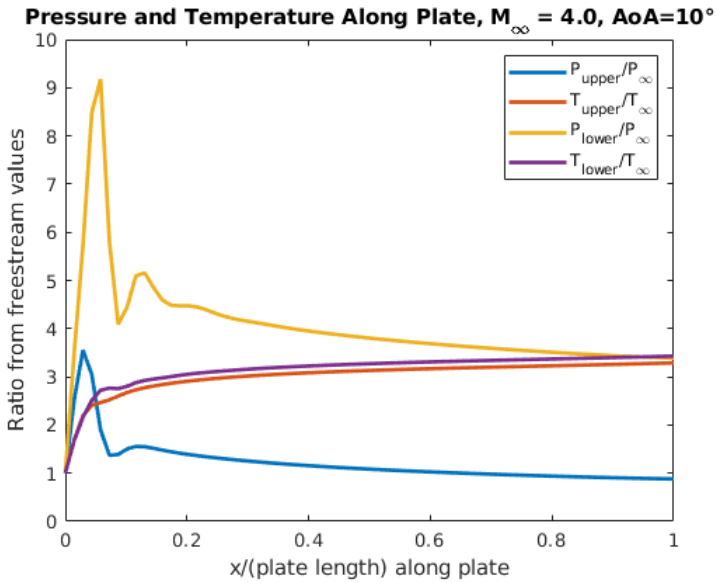

Figure 12: Pressure and temperature ratios along the upper and lower surface of the plate. As the angle of attack is increased, the pressure along the upper side decreases and the lower side increases, causing the creation of lift.

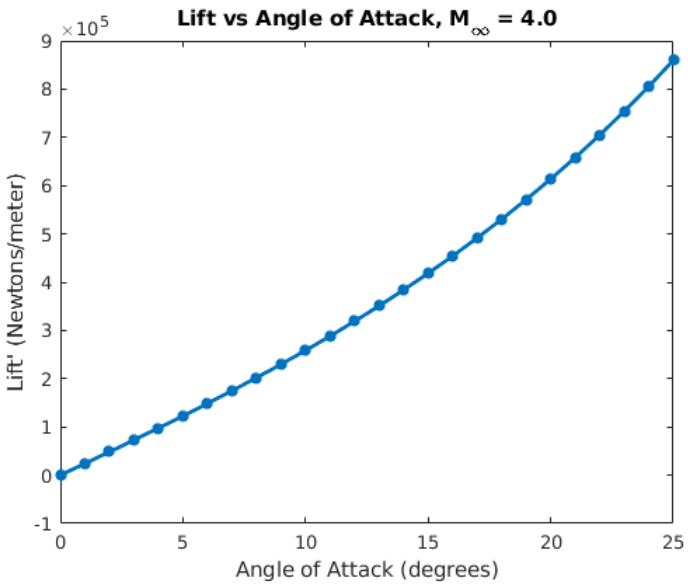

Figure 13: Calculation of lift per unit length (into the page) for a flat plate at 0 to 25 degrees angle of attack. Symmetric airfoils (or in this case infinitely thin flat plates,) do not create lift at zero angle of attack. Airfoil camber is introduced to change this.

APPLICATION: Couette Flow

Couette flow describes the flow between two flat plates, where one is stationary and the other is moving with some constant velocity. For initial conditions, it is assumed that the fluid within the channel starts at zero velocity in both x and y directions at time zero, and the upper wall is started instantaneously to it's nominal velocity. The inflow and outflow boundaries are both allowed to move freely and are interpolated from the flow interior. The no slip condition is enforced along both walls.

The velocity profile at the center (away from inflow and outflow) is shown below in Figure 14 through time, and depicts the flow going from zero velocity up to a linear decrease across the channel between the moving and stationary plates.

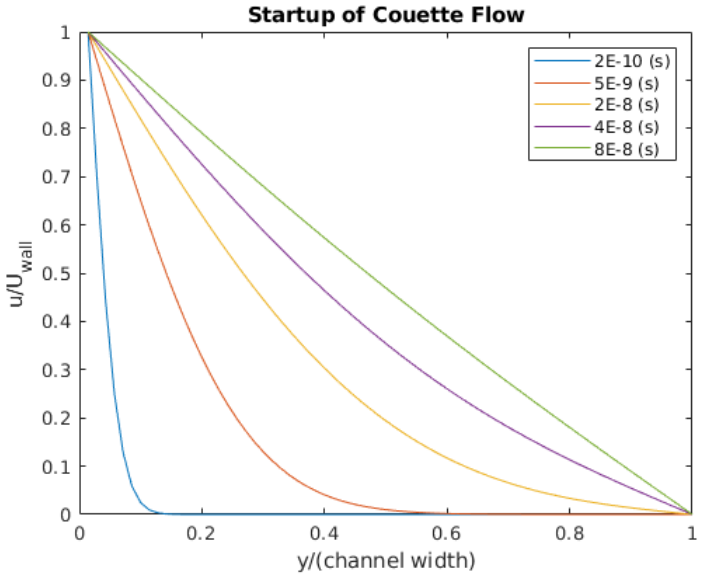

Figure 14: Velocity profile of channel flow from start to steady state. U_{wall} was set to 100 m/s and Channel Width 10^{-5}m to allow for faster convergence.

While the above case depicts how the flow develops under a zero pressure gradient, a pressure gradient is often present and can further increase or decrease the flow depending on if the pressure gradient is in the same direction as the moving plate or the opposite.

Figure 15 depicts the case of a negative pressure gradient, where the inflow boundary pressure was set to 1.25atm and the outflow 1.00atm; this pressure gives the flow an extra "push" down the pipe and we see the velocities increase above that of the wall speed.

For Figure 16 the inflow and outflow pressures were switched between 1.00atm and 1.25atm respectively. Now the pressure gradient is working against the moving wall and an overall decrease is seen, with flow closer to the stationary wall reversing direction and moving back towards the inflow boundary (which was left as free.)

Both of these results are as expected for idealized flow.

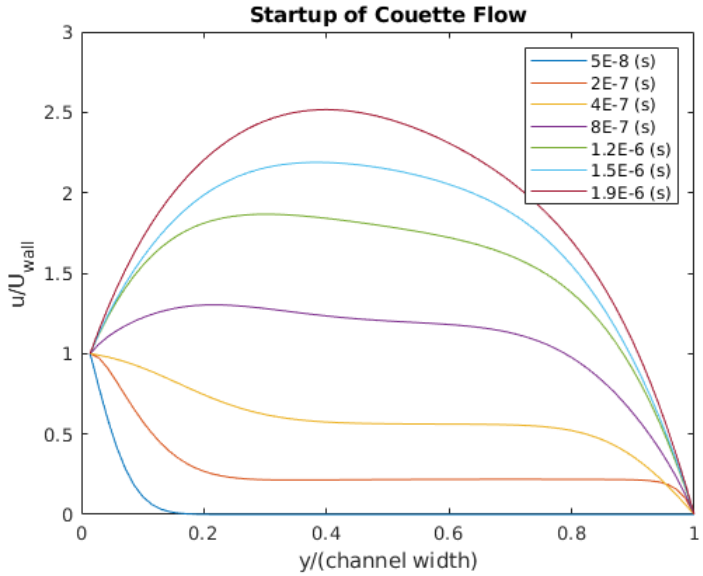

Figure 15: Velocity profile of channel flow from start to steady state, where inflow pressure was 1.25atm and outflow pressure 1.00atm. U_{wall} was set to 100 m/s and Channel Width 10^{-5}m to allow for faster convergence.

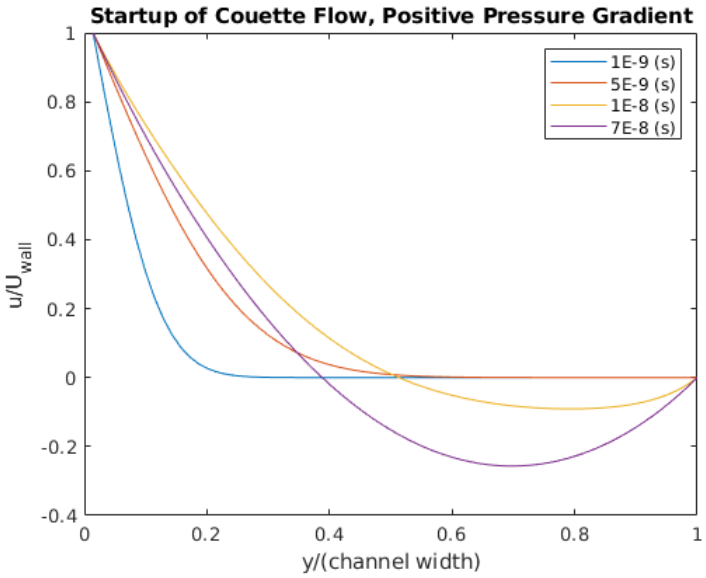

Figure 16: Velocity profile of channel flow from start to steady state, where inflow pressure was 1.00atm and outflow pressure 1.25atm. U_{wall} was set to 100 m/s and Channel Width 10^{-5}m to allow for faster convergence.

REFERENCES

[1] Anderson *Computational Fluid Dynamics: The Basics with Applications.* McGraw-Hill, 1995

[2] Baeder *MacCormack 2D Notes.*

[3] Silawat *Supersonic flow over a sharp leading edged flat plate at zero incidence* `https://commons.wikimedia.org/wiki/File:Supersonic_flow_over_a_sharp_leading_edged_flat_plate_at_zero_incidence.JPG`

CODE

Attached is the base code written in both CUDA C and then MATLAB. Small changes to plate location, initial flow conditions, and boundary conditions were made to the MATLAB version create each of the above cases.

NavierStokesSolver.cu

```
/*
 ============================================================================

 Name        : NavierStokesSolver.cu
 Author      : Matt Kennedy
 Version     : 1.0
 Description : Solve the Navier Stokes over a flat plate
 ============================================================================

 */

#include <iostream>
#include <numeric>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

using namespace std;



#define gpuErrchk(ans) { gpuAssert((ans), __FILE__,
    __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file,
     int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n",
          cudaGetErrorString(code), file, line);
      if (abort) exit(code);
```

```
27      }
28  }
29
30
31
32  //
        ============================================================================

33  //
        ============================================================================

34  // define any configuration parameters needed by both CPU
        and GPU
35  //
        ============================================================================

36  //
        ============================================================================

37
38  // note that we'll probably get segfaults if the cpu and
        gpu variables are set to different values!
39  // (they didn't seem to be happy trying to set one from
        the other, so need to manually change both for now)
40  // (and host code isn't happy about reading from a device
         global variable)
41  int jmax = 70;
42  int kmax = 70;
43  __constant__ int jmax_d = 70;
44  __constant__ int kmax_d = 70;
45
46  double plateLength = 0.00001;
47  __constant__ double plateLength_d = 0.00001;
48
49  double CFL = 0.2; // can fudge this to help stability
50  __constant__ double CFL_d = 0.2; // can fudge this to
        help stability
51
52  double M0 = 4.0;
53
54  double dx = plateLength / (jmax   1); // uniform for now
55  //double dy = plateLength / (kmax   1);
56  double dy = 1.1869 * pow(10.0, 7); // calculated from
        boundary layer
57
58
59  __constant__ double u0_d = 1361.12;
60
61
62
```

```
63 //
       ================================================================

64 //
       ================================================================

65 // define any global variables needed by the GPU
66 //
       ================================================================

67 //
       ================================================================


68
69 __constant__ double gam_d = 1.4; // seems "gamma" is a
       protected name from the numeric library
70 __constant__ double Pr_d = 0.71; // Prandtl number
71 __constant__ double R_d = 287.0; // specific gas constant
72 __constant__ double Cv_d = 0.7171*1000; // specific heat
       capacity of air
73 __constant__ double Cp_d = 1.006 * 1000; // specific heat
        capacity of air
74 __constant__ double mu0_d = 1.7894E 5; // dynamic
       viscosity of air
75 __constant__ double T0_d = 288.16;
76 __constant__ double p0_d = 101325.0;
77
78
79
80 //
       ================================================================

81 //
       ================================================================

82 // define any global variables needed by the CPU
83 //
       ================================================================

84 //
       ================================================================


85
86
87 double a0 = 340.28;
88 double u0 = M0*a0;
89
90 double p0 = 101325.0;
91 double T0 = 288.16;
92
```

```
93
94  double v0 = 0;
95
96  double gam = 1.4; // seems "gamma" is a protected name
        from the numeric library
97  double Pr = 0.71; // Prandtl number
98  double R = 287.0; // specific gas constant
99  double Cv = 0.7171*1000; // specific heat capacity of air
100 double Cp = 1.006 * 1000; // specific heat capacity of
        air
101
102 double rho0 = p0 / (R * T0);
103 double e0 = T0 * Cv;
104
105 double mu0 = 1.7894E 5; // dynamic viscosity of air
106 double Re = rho0 * u0 * plateLength / mu0;
107
108
109
110
111
112
113
114 __device__ void calc_Q(double* Q_d, double* u_d, double*
        v_d, double* p_d, double* T_d, int j, int k) {
115
116         int ind2 = j*kmax_d + k; // flattened index for
                our 2d arrays
117         int ind3_0 = (j + 0*jmax_d)*kmax_d + k; //
                flattened index for the first dim of our 3d
                arrays
118         int ind3_1 = (j + 1*jmax_d)*kmax_d + k; // stack
                them like extra rows
119         int ind3_2 = (j + 2*jmax_d)*kmax_d + k;
120         int ind3_3 = (j + 3*jmax_d)*kmax_d + k;
121
122         double rho_val = p_d[ind2] / (R_d * T_d[ind2]);
123         double e_val = Cv_d * T_d[ind2]; // energy of air
                 based on temp
124         double Et_val = rho_val * (e_val + 0.5*(u_d[ind2
                ]*u_d[ind2] + v_d[ind2]*v_d[ind2]));
125
126         Q_d[ind3_0] = rho_val;
127         Q_d[ind3_1] = rho_val * u_d[ind2];
128         Q_d[ind3_2] = rho_val * v_d[ind2];
129         Q_d[ind3_3] = Et_val;
130
131 }
132
133
```

```
134 __device__ void heatFluxParameters(double* T_d, double
    mu_val, bool isPredictor, int j, int k, double dx,
    double dy, double* q) {
135
136         double dTdx;
137         double dTdy;
138
139         if (isPredictor) { // scheme is forward, make
                this backward
140
141                 if (j > 0)
142                         dTdx = (T_d[j*kmax_d + k]    T_d[(
                            j 1)*kmax_d + k])/dx;
143                 else if (j == 0)
144                         dTdx = (T_d[(j+1)*kmax_d + k]
                            T_d[j*kmax_d + k])/dx;
145
146                 if (k > 0)
147                         dTdy = (T_d[j*kmax_d + k]    T_d[j
                            *kmax_d + (k 1)])/dy;
148                 else if (k == 0)
149                         dTdy = (T_d[j*kmax_d + k + 1]
                            T_d[j*kmax_d + k])/dy;
150
151         }
152         else { // scheme is backward, make this forward
153
154                 if (j < jmax_d 1)
155                         dTdx = (T_d[(j+1)*kmax_d + k]
                            T_d[j*kmax_d + k])/dx;
156                 else if (j == jmax_d    1)
157                         dTdx = (T_d[j*kmax_d + k]    T_d[(
                            j 1)*kmax_d + k]) / dx;
158
159                 if (k < kmax_d 1)
160                         dTdy = (T_d[j*kmax_d+k+1]    T_d[j
                            *kmax_d + k]) / dy;
161                 else if (k == kmax_d    1)
162                         dTdy = (T_d[j*kmax_d + k]    T_d[j
                            *kmax_d + k 1]) / dy;
163
164         }
165
166         double k_cond = mu_val * Cp_d / Pr_d;
167
168         q[0] =  k_cond * dTdx;
169         q[1] =  k_cond * dTdy;
170 }
171
```

```
172 __device__ void shearParameters(double* u_d, double* v_d,
        double mu, bool isPredictor, int j, int k, double dx,
        double dy, double* shears) {
173
174         // calculate shear for a single location (j,k)
175         // inputs are assumed to be entire matrices
176
177         double dvdx_FB;
178         double dudx_FB;
179         double dvdy_FB;
180         double dudy_FB;
181         double dvdx_C;
182         double dudx_C;
183         double dvdy_C;
184         double dudy_C;
185
186         // calculate the forward or backward differenced
                versions
187         if (isPredictor) {
188             // want opposite direction from scheme step
                    differencing
189             // scheme is forward, make this backward
190
191             if (j > 0) {
192                 dvdx_FB = (v_d[j*kmax_d + k]    v_d[(j 1)*
                        kmax_d + k])/dx;
193                 dudx_FB = (u_d[j*kmax_d + k]    u_d[(j 1)*
                        kmax_d + k])/dx;
194             }
195             else {
196                 dvdx_FB = (v_d[(j+1)*kmax_d + k]    v_d[j*
                        kmax_d + k])/dx; // except first point
                         forward
197                 dudx_FB = (u_d[(j+1)*kmax_d + k]    u_d[j*
                        kmax_d + k])/dx; // except first point
                         forward
198             }
199
200
201
202             if (k > 0) {
203                 dudy_FB = (u_d[j*kmax_d+k]    u_d[j*kmax_d
                        +k 1])/dy;
204                 dvdy_FB = (v_d[j*kmax_d+k]    v_d[j*kmax_d
                        +k 1])/dy;
205             }
206             else {
207                 dudy_FB = (u_d[j*kmax_d+k+1]    u_d[j*
                        kmax_d+k])/dy; // except first point
                        forward
```

```
208                 dvdy_FB = (v_d[j*kmax_d+k+1]    v_d[j*
                        kmax_d+k])/dy; // except first point
                        forward
209             }
210
211         }
212         else {
213
214             // scheme is backward, make this forward
215
216             if (j < jmax_d    1) {
217                 dvdx_FB = (v_d[(j+1)*kmax_d + k]    v_d[j*
                        kmax_d + k])/dx;
218                 dudx_FB = (u_d[(j+1)*kmax_d + k]    u_d[j*
                        kmax_d + k])/dx;
219             }
220             else {
221                 dvdx_FB = (v_d[j*kmax_d+k]    v_d[(j 1)*
                        kmax_d + k])/dx; // except jmax
                        backward
222                 dudx_FB = (u_d[j*kmax_d+k]    u_d[(j 1)*
                        kmax_d + k])/dx; // except jmax
                        backward
223             }
224
225             if (k < kmax_d 1) {
226                 dudy_FB = (u_d[j*kmax_d + k+1]    u_d[j*
                        kmax_d + k])/dy;
227                 dvdy_FB = (v_d[j*kmax_d + k+1]    v_d[j*
                        kmax_d + k])/dy;
228             }
229             else {
230                 dudy_FB = (u_d[j*kmax_d + k]    u_d[j*
                        kmax_d + k 1])/dy; // except kmax
                        backward
231                       dvdy_FB = (v_d[j*kmax_d + k]
                              v_d[j*kmax_d + k 1])/dy; //
                              except kmax backward
232             }
233
234
235         }
236
237         // and then we want centeral differenced versions
238
239         if (j == 0) {
240             dvdx_C = (v_d[(j+1)*kmax_d + k]    v_d[j*
                    kmax_d + k])/dx;
241             dudx_C = (u_d[(j+1)*kmax_d + k]    u_d[j*
                    kmax_d + k])/dx;
```

```
242             }
243             else if (j == jmax_d    1)
244             {
245                 dvdx_C = (v_d[j*kmax_d + k]    v_d[(j 1)*
                        kmax_d + k])/dx;
246                 dudx_C = (u_d[j*kmax_d + k]    u_d[(j 1)*
                        kmax_d + k])/dx;
247             }
248             else {
249                 dvdx_C = (v_d[(j+1)*kmax_d + k]    v_d[(j 1)*
                        kmax_d + k])/(2*dx);
250                 dudx_C = (u_d[(j+1)*kmax_d + k]    u_d[(j 1)*
                        kmax_d + k])/(2*dx);
251             }
252
253
254
255             if (k == 0) {
256                 dudy_C = (u_d[j*kmax_d + k+1]    u_d[j*kmax_d
                        + k])/dy;
257                 dvdy_C = (v_d[j*kmax_d + k+1]    v_d[j*kmax_d
                        + k])/dy;
258             }
259             else if (k == kmax_d 1) {
260                 dudy_C = (u_d[j*kmax_d + k]    u_d[j*kmax_d +
                        k 1])/dy;
261                 dvdy_C = (v_d[j*kmax_d + k]    v_d[j*kmax_d +
                        k 1])/dy;
262             }
263             else {
264                 dudy_C = (u_d[j*kmax_d + k+1]    u_d[j*kmax_d
                        + k 1])/(2*dy);
265                 dvdy_C = (v_d[j*kmax_d + k+1]    v_d[j*kmax_d
                        + k 1])/(2*dy);
266             }
267
268
269             // these come from page 65 and 66 in Anderson
270
271             double lambda =  (2.0/3.0) * mu; // second
                    viscosity coefficient estimated by Stokes
272
273             // use the forward/backward du/dx and central dv/
                    dy for both F and G
274             double txx = lambda * ( dudx_FB + dvdy_C ) + 2 *
                    mu * dudx_FB;
275
276             // use the forward/backward dv/dy and central du/
                    dx for both F and G
```

```
277         double tyy = lambda * ( dudx_C + dvdy_FB ) + 2 *
                mu * dvdy_FB;
278
279         double txy_F = mu * ( dvdx_FB + dudy_C );
280         double txy_G = mu * ( dvdx_C + dudy_FB );
281
282         shears[0] = txx;
283         shears[1] = tyy;
284         shears[2] = txy_F;
285         shears[3] = txy_G;
286
287 }
288
289
290 __device__ void calc_FG(double* F_d, double* G_d, double*
        u_d, double* v_d, double* p_d, double* T_d, bool
    isPredictor, int j, int k, double dx, double dy) {
291
292         int ind2 = j*kmax_d + k; // flattened index for
                our 2d arrays
293         int ind3_0 = (j + 0*jmax_d)*kmax_d + k; //
                flattened index for the first dim of our 3d
                arrays
294         int ind3_1 = (j + 1*jmax_d)*kmax_d + k; // stack
                them like extra rows
295         int ind3_2 = (j + 2*jmax_d)*kmax_d + k;
296         int ind3_3 = (j + 3*jmax_d)*kmax_d + k;
297
298         double rho_val = p_d[ind2] / (R_d * T_d[ind2]);
299         double e_val = Cv_d * T_d[ind2]; // energy of air
                based on temp
300         double Et_val = rho_val * (e_val + 0.5*(u_d[ind2
                ]*u_d[ind2] + v_d[ind2]*v_d[ind2]));
301
302         double mu_val = mu0_d * pow(T_d[ind2] / T0_d,
                1.5) * (T0_d + 110)/(T_d[ind2] + 110); //
                sutherlands law
303
304         double q[2];
305         double shears[4];
306
307         heatFluxParameters(T_d, mu_val, isPredictor, j, k
                , dx, dy, q);
308         shearParameters(u_d, v_d, mu_val, isPredictor, j,
                k, dx, dy, shears);
309
310         // and unpack these for easier use
311         double qx = q[0];
312         double qy = q[1];
313         double txx = shears[0];
```

```
314         double tyy = shears[1];
315         double txy_F = shears[2];
316         double txy_G = shears[3];
317 
318 
319         F_d[ind3_0] = rho_val * u_d[ind2];
320         F_d[ind3_1] = rho_val * pow(u_d[ind2],2) + p_d[
                ind2]   txx;
321         F_d[ind3_2] = rho_val * u_d[ind2]*v_d[ind2]
                txy_F;
322         F_d[ind3_3] = (Et_val + p_d[ind2]) * u_d[ind2]
                u_d[ind2] * txx   v_d[ind2] * txy_F + qx;
323 
324         G_d[ind3_0] = rho_val * v_d[ind2];
325         G_d[ind3_1] = rho_val * u_d[ind2] * v_d[ind2]
                txy_G;
326         G_d[ind3_2] = rho_val * pow(v_d[ind2],2) + p_d[
                ind2]   tyy;
327         G_d[ind3_3] = (Et_val + p_d[ind2]) * v_d[ind2]
                u_d[ind2] * txy_G   v_d[ind2] * tyy + qy;
328 
329 }
330 
331 
332 __device__ void MacCormackPredictorUniform(double*
        Q_pred_d, double* Q_d, double* F_d, double* G_d,
        double dt, int j, int k, double dx, double dy) {
333 
334 // DO MACCORMACKS FOR INTERIOR POINTS ONLY
335         if (j == 0 || k == 0 || j == jmax_d 1 || k ==
                kmax_d 1)
336                 return;
337 
338         // have each thread calculate all 4 dimensions at
                 a single loc
339         double flux;
340         for (int dim=0; dim<4; dim++) {
341 
342                 int ind_this = (j + dim*jmax_d)*kmax_d +
                        k;
343                 int ind_nextJ = (j+1 + dim*jmax_d)*kmax_d
                         + k;
344                 int ind_nextK = (j + dim*jmax_d)*kmax_d +
                         k+1;
345 
346                 flux = (F_d[ind_nextJ]   F_d[ind_this])/
                        dx + (G_d[ind_nextK]   G_d[ind_this])/
                        dy;
347                 Q_pred_d[ind_this] = Q_d[ind_this]   dt *
                         flux;
```

```
348         }
349 }
350
351
352 __device__ void MacCormackCorrectorUniform(double*
    Q_pred_d, double* Q_d, double* F_d, double* G_d,
    double dt, int j, int k, double dx, double dy) {
353
354         // DO MACCORMACKS FOR INTERIOR POINTS ONLY
355         if (j == 0 || k == 0 || j == jmax_d 1 || k ==
            kmax_d 1)
356                 return;
357
358         // have each thread calculate all 4 dimensions at
                a single (j,k) location
359         double flux;
360         for (int dim=0; dim<4; dim++) {
361
362                 int ind_this = (j + dim*jmax_d)*kmax_d +
                    k;
363                 int ind_prevJ = (j 1 + dim*jmax_d)*kmax_d
                     + k;
364                 int ind_prevK = (j + dim*jmax_d)*kmax_d +
                     k 1;
365
366                 flux = (F_d[ind_this]   F_d[ind_prevJ])/
                    dx + (G_d[ind_this]   G_d[ind_prevK])/
                    dy;
367                 Q_d[ind_this] = 0.5*( Q_d[ind_this] +
                    Q_pred_d[ind_this]   dt*flux );
368         }
369 }
370
371
372 __device__ void primativesFromQ(double* Q_d, double*
    rho_d, double* u_d, double* v_d, double* p_d, double*
    T_d, double* e_d, int j, int k) {
373
374         int ind2 = j*kmax_d + k; // flattened index for
            our 2d arrays
375         int ind3_0 = (j + 0*jmax_d)*kmax_d + k; //
            flattened index for the first dim of our 3d
            arrays
376         int ind3_1 = (j + 1*jmax_d)*kmax_d + k; // stack
            them like extra rows
377         int ind3_2 = (j + 2*jmax_d)*kmax_d + k;
378         int ind3_3 = (j + 3*jmax_d)*kmax_d + k;
379
380         rho_d[ind2] = Q_d[ind3_0];
381         u_d[ind2] = Q_d[ind3_1] / Q_d[ind3_0];
```

```
382         v_d[ind2] = Q_d[ind3_2] / Q_d[ind3_0];
383         e_d[ind2] = Q_d[ind3_3] / Q_d[ind3_0]   0.5*( pow
                (u_d[ind2], 2) + pow(v_d[ind2], 2) );
384
385         T_d[ind2] = e_d[ind2] / Cv_d;
386         p_d[ind2] = Q_d[ind3_0] * R_d * T_d[ind2];
387
388 }
389
390
391 __device__ void enforceBC_nonSurface(double* u_d, double*
        v_d, double* p_d, double* T_d, int j, int k) {
392
393         // need to first establish all the boundary
                conditions at the non surface
394         // values, and then go back and do the surface
                boundary conditions
395
396         // this is really only needed if the surface goes
                 all the way to the outflow
397         // so that the last surface point can be
                interpolated with updated values
398
399         int ind = j*kmax_d + k;
400
401         if ( j == 0 && k == 0) {  // leading edge
402
403                 u_d[ind] = 0;
404                 v_d[ind] = 0;
405                 p_d[ind] = p0_d;
406                 T_d[ind] = T0_d;
407         }
408         else if (j == 0 || k == kmax_d 1) { // inflow
                from upstream OR upper boundary
409
410                 u_d[ind] = u0_d;
411                 v_d[ind] = 0;
412                 p_d[ind] = p0_d;
413                 T_d[ind] = T0_d;
414         }
415         else if (j == jmax_d 1) { // outflow
                extrapolate from interior values
416                 int ind1 = (j 1)*kmax_d + k;
417                 int ind2 = (j 2)*kmax_d + k;
418
419                 u_d[ind] = 2*u_d[ind1]   u_d[ind2];
420                 v_d[ind] = 2*v_d[ind1]   v_d[ind2];
421                 p_d[ind] = 2*p_d[ind1]   p_d[ind2];
422                 T_d[ind] = 2*T_d[ind1]   T_d[ind2];
423
```

```
424         }
425 }
426
427
428 __device__ void enforceBC_surface(double* u_d, double*
      v_d, double* p_d, double* T_d, int j, int k) {
429
430         // need to first establish all the boundary
              conditions at the non surface
431         // values, and then go back and do the surface
              boundary conditions
432
433         // this is really only needed if the surface goes
               all the way to the outflow
434         // so that the last surface point can be
              interpolated with updated values
435
436         int ind = j*kmax_d + k;
437
438         if (k == 0 && j > 0){
439                 u_d[ind] = 0;
440                 v_d[ind] = 0;
441                 p_d[ind] = 2*p_d[j*kmax_d + 1]    p_d[j*
                      kmax_d + 2];
442                 T_d[ind] = T_d[j*kmax_d + 1];
443         }
444
445 }
446
447
448
449
450 __global__ void iterateScheme_part1(double* x_d, double*
      y_d, double* u_d, double* v_d, double* p_d, double*
      T_d, double* rho_d, double* e_d, double* Q_d, double*
      Q_pred_d, double* F_d, double* G_d, double dx, double
      dy, double dt) {
451
452         int j = blockIdx.x * blockDim.x + threadIdx.x;
453         int k = blockIdx.y * blockDim.y + threadIdx.y;
454
455         if (j < jmax_d && k < kmax_d)
456         {
457
458                 calc_Q(Q_d, u_d, v_d, p_d, T_d, j, k);
459
460                 bool isPredictor = true;
461                 calc_FG(F_d, G_d, u_d, v_d, p_d, T_d,
                      isPredictor, j, k, dx, dy);
462
```

```
463
464                 // think we need to actually do different
                        kernel launches here ...
465                 // seems to be no easy way to sync all
                        blocks, and inherently not all blocks
                        may be executed at once if the grid
                        gets too large
466
467 //              MacCormackPredictorUniform(Q_pred_d, Q_d,
        F_d, G_d, dt, j, k, dx, dy);
468
469         }
470 }
471
472
473 __global__ void iterateScheme_part2(double* x_d, double*
        y_d, double* u_d, double* v_d, double* p_d, double*
        T_d, double* rho_d, double* e_d, double* Q_d, double*
        Q_pred_d, double* F_d, double* G_d, double dx, double
        dy, double dt) {
474
475         int j = blockIdx.x * blockDim.x + threadIdx.x;
476         int k = blockIdx.y * blockDim.y + threadIdx.y;
477
478         if (j < jmax_d && k < kmax_d)
479         {
480
481                 // think we need to actually do different
                        kernel launches here ...
482                 // seems to be no easy way to sync all
                        blocks, and inherently not all blocks
                        may be executed at once if the grid
                        gets too large
483
484                 MacCormackPredictorUniform(Q_pred_d, Q_d,
                        F_d, G_d, dt, j, k, dx, dy);
485
486                 primativesFromQ(Q_pred_d, rho_d, u_d, v_d
                        , p_d, T_d, e_d, j, k);
487
488         }
489 }
490
491
492 __global__ void iterateScheme_part3(double* x_d, double*
        y_d, double* u_d, double* v_d, double* p_d, double*
        T_d, double* rho_d, double* e_d, double* Q_d, double*
        Q_pred_d, double* F_d, double* G_d, double dx, double
        dy, double dt) {
493
```

```
494         int j = blockIdx.x * blockDim.x + threadIdx.x;
495         int k = blockIdx.y * blockDim.y + threadIdx.y;
496 
497         if (j < jmax_d && k < kmax_d)
498         {
499 
500                 enforceBC_nonSurface(u_d, v_d, p_d, T_d,
                        j, k);
501 
502         }
503 }
504 
505 
506 __global__ void iterateScheme_part4(double* x_d, double*
        y_d, double* u_d, double* v_d, double* p_d, double*
        T_d, double* rho_d, double* e_d, double* Q_d, double*
        Q_pred_d, double* F_d, double* G_d, double dx, double
        dy, double dt) {
507 
508         int j = blockIdx.x * blockDim.x + threadIdx.x;
509         int k = blockIdx.y * blockDim.y + threadIdx.y;
510 
511         if (j < jmax_d && k < kmax_d)
512         {
513 
514                 enforceBC_surface(u_d, v_d, p_d, T_d, j,
                        k);
515 
516         }
517 }
518 
519 
520 __global__ void iterateScheme_part5(double* x_d, double*
        y_d, double* u_d, double* v_d, double* p_d, double*
        T_d, double* rho_d, double* e_d, double* Q_d, double*
        Q_pred_d, double* F_d, double* G_d, double dx, double
        dy, double dt) {
521 
522         int j = blockIdx.x * blockDim.x + threadIdx.x;
523         int k = blockIdx.y * blockDim.y + threadIdx.y;
524 
525         if (j < jmax_d && k < kmax_d)
526         {
527 
528                 bool isPredictor = false;
529                 calc_FG(F_d, G_d, u_d, v_d, p_d, T_d,
                        isPredictor, j, k, dx, dy);
530 
531         }
532 }
```

```
533
534
535 __global__ void iterateScheme_part6(double* x_d, double*
    y_d, double* u_d, double* v_d, double* p_d, double*
    T_d, double* rho_d, double* e_d, double* Q_d, double*
    Q_pred_d, double* F_d, double* G_d, double dx, double
    dy, double dt) {
536
537         int j = blockIdx.x * blockDim.x + threadIdx.x;
538         int k = blockIdx.y * blockDim.y + threadIdx.y;
539
540         if (j < jmax_d && k < kmax_d)
541         {
542
543                 MacCormackCorrectorUniform(Q_pred_d, Q_d,
                        F_d, G_d, dt, j, k, dx, dy);
544
545                 primativesFromQ(Q_d, rho_d, u_d, v_d, p_d
                        , T_d, e_d, j, k);
546
547         }
548 }
549
550
551 __global__ void iterateScheme_part7(double* x_d, double*
    y_d, double* u_d, double* v_d, double* p_d, double*
    T_d, double* rho_d, double* e_d, double* Q_d, double*
    Q_pred_d, double* F_d, double* G_d, double dx, double
    dy, double dt) {
552
553         int j = blockIdx.x * blockDim.x + threadIdx.x;
554         int k = blockIdx.y * blockDim.y + threadIdx.y;
555
556         if (j < jmax_d && k < kmax_d)
557         {
558
559                 enforceBC_nonSurface(u_d, v_d, p_d, T_d,
                        j, k);
560
561         }
562 }
563
564
565 __global__ void iterateScheme_part8(double* x_d, double*
    y_d, double* u_d, double* v_d, double* p_d, double*
    T_d, double* rho_d, double* e_d, double* Q_d, double*
    Q_pred_d, double* F_d, double* G_d, double dx, double
    dy, double dt) {
566
567         int j = blockIdx.x * blockDim.x + threadIdx.x;
```

```
568         int k = blockIdx.y * blockDim.y + threadIdx.y;
569 
570         if (j < jmax_d && k < kmax_d)
571         {
572 
573                 enforceBC_surface(u_d, v_d, p_d, T_d, j,
                        k);
574 
575         }
576 }
577 
578 
579 
580 
581 
582 
583 double BoundaryLayerThickness() {
584         return 5 * plateLength / sqrt(Re);
585 }
586 
587 void setupGrid(double* x, double* y) {
588         // just do a uniform grid for now
589 
590         for (int j=0; j<jmax; j++)
591                 x[j] = j*dx;
592         for (int k=0; k<kmax; k++)
593                 y[k] = k*dy;
594 }
595 
596 void initializePrimatives(double* u, double* v, double* p
        , double* T, double* rho, double* e) {
597 
598         for (int j=0; j<jmax; j++) {
599                 for (int k=0; k<kmax; k++) {
600                         u[j*kmax+k] = u0;
601                         v[j*kmax+k] = v0;
602                         p[j*kmax+k] = p0;
603                         T[j*kmax+k] = T0;
604                         rho[j*kmax+k] = rho0;
605                         e[j*kmax+k] = e0;
606                 }
607         }
608 }
609 
610 
611 void applyBC_UpperPlate(double* u, double* v, double* p,
        double* T) {
612 
613 
614         // leading edge
```

```
615             u[0*kmax+0] = 0;
616             v[0*kmax+0] = 0;
617             p[0*kmax+0] = p0;
618             T[0*kmax+0] = T0;
619
620             // inflow (j=0, k=all)
621             for (int k=0; k<kmax; k++) {
622                     u[0*kmax+k] = u0;
623                     v[0*kmax+k] = 0;
624                     p[0*kmax+k] = p0;
625                     T[0*kmax+k] = T0;
626             }
627
628             // upper boundary (j=all, k=kmax-1)
629             for (int j=0; j<jmax; j++) {
630                     u[j*kmax+kmax-1] = u0;
631                     v[j*kmax+kmax-1] = 0;
632                     p[j*kmax+kmax-1] = p0;
633                     T[j*kmax+kmax-1] = T0;
634             }
635
636             // outflow (j=jmax-1, k=all)
637             // extrapolate from interior values
638             for (int k=0; k<kmax; k++) {
639                     u[(jmax-1)*kmax + k] = 2*u[(jmax-2)*kmax
                            + k] - u[(jmax-3)*kmax + k];
640                     v[(jmax-1)*kmax + k] = 2*v[(jmax-2)*kmax
                            + k] - v[(jmax-3)*kmax + k];
641                     p[(jmax-1)*kmax + k] = 2*p[(jmax-2)*kmax
                            + k] - p[(jmax-3)*kmax + k];
642                     T[(jmax-1)*kmax + k] = 2*T[(jmax-2)*kmax
                            + k] - T[(jmax-3)*kmax + k];
643             }
644
645             // and plate surface (j=all, k=0)
646             for (int j=0; j<jmax; j++) {
647                     u[j*kmax + 0] = 0;
648                     v[j*kmax + 0] = 0;
649                     p[j*kmax + 0] = 2*p[j*kmax + 1] - p[j*
                            kmax + 2];
650                     T[j*kmax + 0] = T[j*kmax + 1];
651             }
652
653
654 }
655
656
657 double calc_dt(double* u, double* v, double* p, double* T
        , double dx, double dy) {
658
```

```
659         // not sure the best way to do this on the GPU
660         // seems to be some parallel reduce functions
                which can be called as their own kernels
661         // which would at least prevent us from having to
                copy back the primative variables to the host
                every iteration
662         // but lets not worry about that for now
663
664         double rho_val;
665         double mu_val;
666         double temp_val;
667
668         double vprime =  INFINITY;
669         for (int j=0; j<jmax; j++) {
670                 for (int k=0; k<kmax; k++) {
671
672                         int ind = j*kmax + k;
673
674                         rho_val = p[ind] / (R * T[ind]);
675                         mu_val = mu0 * pow(T[ind]/T0,
                                1.5) * (T0 + 110)/(T[ind] +
                                110);
676
677                         temp_val = (4/3) * mu_val * (gam
                                * mu_val / Pr) / rho_val; //
                                find the max of this
678                         if (temp_val > vprime)
679                                 vprime = temp_val;
680                 }
681         }
682
683         double spaceUnit = pow( 1/(dx*dx) + 1/(dy*dy),
                0.5 );
684         double term1;
685         double term2;
686         double term3;
687         double term4;
688         double dt_cfl;
689
690         double dt = INFINITY;
691         for (int j=0; j<jmax; j++) {
692                 for (int k=0; k<kmax; k++) {
693
694                         int ind = j*kmax + k;
695
696                         rho_val = p[ind] / (R * T[ind]);
697
698                         term1 = abs( u[ind] ) / dx;
699                         term2 = abs( v[ind] ) / dy;
```

```
700                             term3 = pow( gam*p[ind]/rho_val,
                                    0.5 ) * spaceUnit;
701                             term4 = 2 * vprime * pow(
                                    spaceUnit, 2);
702
703                             dt_cfl = 1/(term1 + term2 + term3
                                     + term4);
704
705                             if (CFL*dt_cfl < dt)
706                                     dt = CFL*dt_cfl;
707                     }
708             }
709
710             return dt;
711     }
712
713
714
715     void arrayToCSV(double* values, char* filename, int
            numDims) {
716      FILE *fp;
717      fp = fopen(filename, "w+");
718
719      for (int dim=0; dim<numDims; dim++) {
720              for (int j=0; j<jmax; j++) {
721                      for (int k=0; k<kmax; k++) {
722
723                              fprintf(fp, ", %f", values[(j +
                                    dim*jmax)*kmax + k]);
724                      }
725                      fprintf(fp, "\n");
726              }
727
728              if (dim < numDims 1)
729                      fprintf(fp, "Dimension Starting: %i\n",
                            dim+1);
730      }
731
732
733     }
734
735
736     int main(void)
737     {
738
739             double* x = (double*)malloc(jmax*sizeof(double));
740             double* y = (double*)malloc(kmax*sizeof(double));
741
742             double* u = (double*)malloc( jmax*kmax*sizeof(
                    double) );
```

```
743         double* v = (double*)malloc( jmax*kmax*sizeof(
                double) );
744         double* p = (double*)malloc( jmax*kmax*sizeof(
                double) );
745         double* T = (double*)malloc( jmax*kmax*sizeof(
                double) );
746         double* rho = (double*)malloc( jmax*kmax*sizeof(
                double) );
747         double* e = (double*)malloc( jmax*kmax*sizeof(
                double) );
748
749
750
751         initializePrimatives(u, v, p, T, rho, e);
752         applyBC_UpperPlate(u, v, p, T);
753
754
755
756
757         // technically only needed in GPU memory, but I
                assume we may want to copy back intermediate
                results for debugging
758         // calculating these will be a main component of
                what's being done in parallel, so don't need
                to initialize anything
759         double* Q = (double*)malloc( 4*jmax*kmax*sizeof(
                double));
760         double* Q_pred = (double*)malloc( 4*jmax*kmax*
                sizeof(double));
761         double* F = (double*)malloc( 4*jmax*kmax*sizeof(
                double));
762         double* G = (double*)malloc( 4*jmax*kmax*sizeof(
                double));
763
764
765         // iniitialize and allocate device variables
766         double* x_d;
767         double* y_d;
768         double* u_d;
769         double* v_d;
770         double* p_d;
771         double* T_d;
772         double* rho_d;
773         double* e_d;
774         double* Q_d;
775         double* Q_pred_d;
776         double* F_d;
777         double* G_d;
778
779         cudaError_t err;
```

```
780
781         err = cudaMalloc((void**)&x_d, jmax*kmax*sizeof(
                double));
782         err = cudaMalloc((void**)&y_d, jmax*kmax*sizeof(
                double));
783         err = cudaMalloc((void**)&u_d, jmax*kmax*sizeof(
                double));
784         err = cudaMalloc((void**)&v_d, jmax*kmax*sizeof(
                double));
785         err = cudaMalloc((void**)&p_d, jmax*kmax*sizeof(
                double));
786         err = cudaMalloc((void**)&T_d, jmax*kmax*sizeof(
                double));
787         err = cudaMalloc((void**)&rho_d, jmax*kmax*sizeof
                (double));
788         err = cudaMalloc((void**)&e_d, jmax*kmax*sizeof(
                double));
789
790         // these are all 3d arrays
791         err = cudaMalloc((void**)&Q_d, 4*jmax*kmax*sizeof
                (double));
792         err = cudaMalloc((void**)&Q_pred_d, 4*jmax*kmax*
                sizeof(double));
793         err = cudaMalloc((void**)&F_d, 4*jmax*kmax*sizeof
                (double));
794         err = cudaMalloc((void**)&G_d, 4*jmax*kmax*sizeof
                (double));
795
796
797         err = cudaMemcpy(x_d, x, jmax*kmax*sizeof(double)
                , cudaMemcpyHostToDevice);
798         err = cudaMemcpy(y_d, y, jmax*kmax*sizeof(double)
                , cudaMemcpyHostToDevice);
799         err = cudaMemcpy(u_d, u, jmax*kmax*sizeof(double)
                , cudaMemcpyHostToDevice);
800         err = cudaMemcpy(v_d, v, jmax*kmax*sizeof(double)
                , cudaMemcpyHostToDevice);
801         err = cudaMemcpy(p_d, p, jmax*kmax*sizeof(double)
                , cudaMemcpyHostToDevice);
802         err = cudaMemcpy(T_d, T, jmax*kmax*sizeof(double)
                , cudaMemcpyHostToDevice);
803         err = cudaMemcpy(rho_d, rho, jmax*kmax*sizeof(
                double), cudaMemcpyHostToDevice);
804         gpuErrchk( cudaMemcpy(e_d, e, jmax*kmax*sizeof(
                double), cudaMemcpyHostToDevice) );
805
806
807
808
809
```

```
810
811         dim3 threadsPerBlock(16,16);
812         dim3 numBlocks(jmax/threadsPerBlock.x + 1, kmax/
                threadsPerBlock.y + 1);
813
814         // so to force the threads to sync I think it's
                safer to just do the different stages in
                different kernel calls, at least initially
815         int maxIter = 1000;
816         for (int iter=0; iter<maxIter; iter++) {
817
818             printf("Calculating iteration %i / %i\n",
                    iter+1, maxIter);
819
820             double dt = calc_dt(u, v, p, T, dx, dy);
821
822             // calculate F, G, and Q
823             iterateScheme_part1<<<numBlocks,
                    threadsPerBlock>>>(x_d, y_d, u_d, v_d
                    , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                     F_d, G_d, dx, dy, dt);
824             gpuErrchk( cudaPeekAtLastError() );
825             gpuErrchk( cudaDeviceSynchronize() );
826
827
828             // calculate MacCormack's Predictor and
                    get back primatives out of Q
829             iterateScheme_part2<<<numBlocks,
                    threadsPerBlock>>>(x_d, y_d, u_d, v_d
                    , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                     F_d, G_d, dx, dy, dt);
830             gpuErrchk( cudaPeekAtLastError() );
831             gpuErrchk( cudaDeviceSynchronize() );
832
833             // enforce boundary conditions at non
                    surface points
834             iterateScheme_part3<<<numBlocks,
                    threadsPerBlock>>>(x_d, y_d, u_d, v_d
                    , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                     F_d, G_d, dx, dy, dt);
835             gpuErrchk( cudaPeekAtLastError() );
836             gpuErrchk( cudaDeviceSynchronize() );
837
838             // enforce boundary conditions at surface
                     points
839             iterateScheme_part4<<<numBlocks,
                    threadsPerBlock>>>(x_d, y_d, u_d, v_d
                    , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                     F_d, G_d, dx, dy, dt);
840             gpuErrchk( cudaPeekAtLastError() );
```

```
841                 gpuErrchk( cudaDeviceSynchronize() );
842
843                 // update F and G for corrected
                        primatives
844                 iterateScheme_part5<<<numBlocks,
                        threadsPerBlock>>>(x_d, y_d, u_d, v_d
                        , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                         F_d, G_d, dx, dy, dt);
845                 gpuErrchk( cudaPeekAtLastError() );
846                 gpuErrchk( cudaDeviceSynchronize() );
847
848                 // calculate MacCormack's Corrector and
                        get back primatives out of Q
849                 iterateScheme_part6<<<numBlocks,
                        threadsPerBlock>>>(x_d, y_d, u_d, v_d
                        , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                         F_d, G_d, dx, dy, dt);
850                 gpuErrchk( cudaPeekAtLastError() );
851                 gpuErrchk( cudaDeviceSynchronize() );
852
853                 // enforce boundary conditions at non
                        surface points
854                 iterateScheme_part7<<<numBlocks,
                        threadsPerBlock>>>(x_d, y_d, u_d, v_d
                        , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                         F_d, G_d, dx, dy, dt);
855                 gpuErrchk( cudaPeekAtLastError() );
856                 gpuErrchk( cudaDeviceSynchronize() );
857
858                 // enforce boundary conditions at surface
                         points
859                 iterateScheme_part8<<<numBlocks,
                        threadsPerBlock>>>(x_d, y_d, u_d, v_d
                        , p_d, T_d, rho_d, e_d, Q_d, Q_pred_d,
                         F_d, G_d, dx, dy, dt);
860                 gpuErrchk( cudaPeekAtLastError() );
861                 gpuErrchk( cudaDeviceSynchronize() );
862
863                 cudaMemcpy(u, u_d, jmax*kmax*sizeof(
                        double), cudaMemcpyDeviceToHost);
864                 cudaMemcpy(v, v_d, jmax*kmax*sizeof(
                        double), cudaMemcpyDeviceToHost);
865                 cudaMemcpy(p, p_d, jmax*kmax*sizeof(
                        double), cudaMemcpyDeviceToHost);
866                 cudaMemcpy(T, T_d, jmax*kmax*sizeof(
                        double), cudaMemcpyDeviceToHost);
867
868         }
869
870
```

```
871         cudaMemcpy(F, F_d, 4*jmax*kmax*sizeof(double),
                cudaMemcpyDeviceToHost);
872         cudaMemcpy(G, G_d, 4*jmax*kmax*sizeof(double),
                cudaMemcpyDeviceToHost);
873         cudaMemcpy(Q, Q_d, 4*jmax*kmax*sizeof(double),
                cudaMemcpyDeviceToHost);
874
875         arrayToCSV(u, "u.csv", 1);
876         arrayToCSV(v, "v.csv", 1);
877         arrayToCSV(p, "p.csv", 1);
878         arrayToCSV(T, "T.csv", 1);
879
880         arrayToCSV(F, "F.csv", 4);
881         arrayToCSV(G, "G.csv", 4);
882         arrayToCSV(Q, "Q.csv", 4);
883
884
885
886
887
888
889         free(x);
890         free(y);
891         free(u);
892         free(v);
893         free(p);
894         free(T);
895         free(rho);
896         free(e);
897         free(Q);
898         free(Q_pred);
899         free(F);
900         free(G);
901
902         cudaFree(x_d);
903         cudaFree(y_d);
904         cudaFree(u_d);
905         cudaFree(v_d);
906         cudaFree(p_d);
907         cudaFree(T_d);
908         cudaFree(rho_d);
909         cudaFree(e_d);
910
911         cudaFree(Q_d);
912         cudaFree(Q_pred_d);
913         cudaFree(F_d);
914         cudaFree(G_d);
915
916         printf("Finishing!\n");
917
```

```
        return 0;
}
```

initialize.m

```

clear

global gamma Pr mu0 R Cv Cp K LHORI Twall T0 M0 u0 p0 x y Re dx dy jmax kmax

LHORI = 0.00001; % plate length
Twall = 288.16; % wall temperature
adiabaticWall = true; % if false, enforce Twall temp at surface




gamma = 1.4;
Pr = 0.71; % Prandtl number
mu0 = 1.7894E-5; % dynamic viscosity of air
R = 287; % specific gas constant
Cv = 0.7171 * 1000; % specific heat capacity of air   I assume we want these in J not kJ
Cp = 1.006 * 1000; % specifc heat capacity of air   I assume we want these in J not kJ

K = 0.2; % Courant number    acts as a fudge factor(?) in equation 10.16


%% define initial values
a0 = 340.28;
p0 = 101325.0;
T0 = 288.16;
M0 = 4.0;
u0 = M0*a0;
v0 = 0;
rho0 = p0 / (R * T0);
e0 = T0 * Cv;

% [rho0, a0, mu0, Et0] = derivedParameters(u0, v0, p0, T0); % this works for single values or matrices

% and get the reynolds number now that we've defined u
Re = rho0 * u0 * LHORI / mu0;


```

```
40 %% set up grid
41
42 xmax = LHORI;
43 ymax = 5 * BoundaryLayerThickness(LHORI);
44
45 jmax = 70;
46 kmax = 70;
47 % dx = (jmax+1)/LHORI;
48 % dy = dx; % just make it a square grid?
49 %
50 % xVals = 0 : dx : (jmax 1)*dx;
51 % yVals = 0: dy : (kmax 1)*dy;
52
53 xVals = linspace(0, xmax, jmax);
54 yVals = linspace(0, ymax, kmax);
55 dx = xVals(2)    xVals(1);
56 dy = yVals(2)    yVals(1);
57
58
59 x = zeros(jmax,kmax);
60 y = zeros(jmax,kmax);
61
62 for k = 1:kmax
63     x(:,k) = xVals;
64 end
65 for j = 1:jmax
66     y(j,:) = yVals;
67 end
68
69 %% set up matrices with initial conditions
70 u = u0 * ones(size(x));
71 v = v0 * ones(size(x));
72 p = p0 * ones(size(x));
73 T = T0 * ones(size(x));
74 rho = rho0 * ones(size(x));
75 e = e0 * ones(size(x));
76 % and I think we just need to set the surface BC with u
      =0?
77 for j=1:jmax
78     u(j,1) = 0;
79 end
80
81
82
83 Q = zeros(jmax, kmax, 4);
84 F = zeros(jmax, kmax, 4);
85 G = zeros(jmax, kmax, 4);
86
87 Q_pred = zeros(jmax, kmax, 4); % need to keep both Q and
      Q_pred, but F and G can be overwritten between stages
```

```
88
89
90  %% calculate our initial dt
91  % lets initially plan to just do this on CPU and return
        back the primatives
92  % every time step
93  for it = 1:10000
94
95      it
96
97      dt = calc_dt(u, v, p, T);
98
99      %
            ==========================================================

100     % LAUNCH KERNEL HERE
101     %
            ==========================================================

102
103     %% update Q, F, and G
104     % updating F and G is definitely our most expensive
            totaly independent
105     % operation, so we can at least do that on the GPU
106     % but if we're careful with syncing our threads I
            think we can get everyone to stop and wait at the
            right times
107
108     for j = 1:jmax
109         for k = 1:kmax
110             Q = calc_Q(Q, u, v, p, T, j, k);
111         end
112     end
113
114     % for debugging:
115     Q1 = Q(:,:,1);
116     Q2 = Q(:,:,2);
117     Q3 = Q(:,:,3);
118     Q4 = Q(:,:,4);
119
120
121     isPredictor = true;
122     for j = 1:jmax
123         for k = 1:kmax
124             [F, G] = calc_FG(F, G, u, v, p, T,
                    isPredictor, j, k);
125         end
126     end
127
128     % for debugging:
```

```
129     F1 = F(:,:,1);
130     F2 = F(:,:,2);
131     F3 = F(:,:,3);
132     F4 = F(:,:,4);
133     G1 = G(:,:,1);
134     G2 = G(:,:,2);
135     G3 = G(:,:,3);
136     G4 = G(:,:,4);
137
138     %
            ================================================================

139     % MUST SYNC THREADS HERE (but I think we can continue
            within same kernel)
140     %
            ================================================================


141
142     %% run Maccormack's predictor (for interior points
            only)
143     % this is for a uniform mesh but we have that other
            non uniform mesh Maccormack's
144     for j = 1:jmax
145         for k = 1:kmax
146
147             Q_pred = MaccormackPredictorUniform(Q_pred, Q
                    , F, G, dt, j, k);
148
149             % and we can just update the primatives here
                    without waiting for a
150             % sync right?
151             [rho, u, v, e, p, T] = primativesFromQ(rho, u
                    , v, e, p, T, Q_pred, j, k);
152
153         end
154     end
155
156
157     %
            ================================================================

158     % MUST SYNC THREADS HERE (but I think we can continue
            within same kernel)
159     %
            ================================================================


160
161     % actually with the extrapolation we need all the
            primatives to finish
162     % calculating before applying BC
```

```
163
164     % note that the vast majority of threads will be
            sitting idle here, but
165     % can't think of any better options, and it's a super
            quick operation
166     for j = 1:jmax
167         for k = 1:kmax
168             [u, v, p, T] = enforceBC_nonSurface(u, v, p,
                    T, j, k);
169         end
170     end
171
172     %
            ================================================================

173     % MUST SYNC THREADS HERE (but I think we can continue
            within same kernel)
174     %
            ================================================================

175
176     % and if the outflow extrapolation overlaps with the
            surface
177     % extrapolation, need to do them in separate stages
            so they don't step
178     % on each others toes (or really just occur in a
            random order, which we
179     % don't want)
180
181     % note that the vast majority of threads will be
            sitting idle here, but
182     % can't think of any better options, and it's a super
            quick operation
183     for j = 1:jmax
184         for k = 1:kmax
185             [u, v, p, T] = enforceBC_surface(u, v, p, T,
                    j, k, adiabaticWall);
186         end
187     end
188
189
190     %
            ================================================================

191     % MUST SYNC THREADS HERE (but I think we can continue
            within same kernel)
192     %
            ================================================================

193
```

```
194     %% update F and G for Q_pred
195     isPredictor = false;
196     for j = 1:jmax
197         for k = 1:kmax
198             [F, G] = calc_FG(F, G, u, v, p, T,
                    isPredictor, j, k);
199         end
200     end
201
202     % for debugging:
203     F1 = F(:,:,1);
204     F2 = F(:,:,2);
205     F3 = F(:,:,3);
206     F4 = F(:,:,4);
207     G1 = G(:,:,1);
208     G2 = G(:,:,2);
209     G3 = G(:,:,3);
210     G4 = G(:,:,4);
211
212
213     %
            ================================================================

214     % MUST SYNC THREADS HERE (but I think we can continue
            within same kernel)
215     %
            ================================================================

216
217     %% and then run Maccormack's corrector (for interior
            points only)
218
219     for j = 1:jmax
220         for k = 1:kmax
221             Q = MaccormackCorrectorUniform(Q, Q_pred, F,
                    G, dt, j, k);
222
223             % and we can just update the primatives here
                    without waiting for a
224             % sync right?
225             [rho, u, v, e, p, T] = primativesFromQ(rho, u
                    , v, e, p, T, Q, j, k);
226
227         end
228     end
229
230     %
            ================================================================

231     % MUST SYNC THREADS HERE (but I think we can continue
```

```
        within same kernel)
232     %
        ================================================================

233
234     % actually with the extrapolation we need all the
            primatives to finish
235     % calculating before applying BC
236
237     % note that the vast majority of threads will be
            sitting idle here, but
238     % can't think of any better options, and it's a super
            quick operation
239     for j = 1:jmax
240         for k = 1:kmax
241             [u, v, p, T] = enforceBC_nonSurface(u, v, p,
                    T, j, k);
242         end
243     end
244
245     %
        ================================================================

246     % MUST SYNC THREADS HERE (but I think we can continue
            within same kernel)
247     %
        ================================================================

248
249     % and if the outflow extrapolation overlaps with the
            surface
250     % extrapolation, need to do them in separate stages
            so they don't step
251     % on each others toes (or really just occur in a
            random order, which we
252     % don't want)
253
254     % note that the vast majority of threads will be
            sitting idle here, but
255     % can't think of any better options, and it's a super
            quick operation
256     for j = 1:jmax
257         for k = 1:kmax
258             [u, v, p, T] = enforceBC_surface(u, v, p, T,
                    j, k, adiabaticWall);
259         end
260     end
261
262
263     % and that's it right?
```

```
264     % copy back u, v, p, T
265
266 %     figure(1)
267 %     plot(p(:,1) / p0)
268 %
269 %     figure(2)
270 %     plot(T(:,1) / T0)
271
272
273 end
```

BoundaryLayerThickness.m

```
1  function blt = BoundaryLayerThickness(LHORI)
2  % Blasius calculation
3
4  global Re
5
6  blt = 5*LHORI / sqrt(Re);
7
8  end
```

calc_dt.m

```
1  function dt = calc_dt(u, v, p, T)
2
3  % for finding the max/min I'm not sure the best way to do
       this on the GPU
4  % lets just plan on doing it on the CPU initially
5
6  global gamma Pr dx dy K jmax kmax R mu0 T0
7
8  %dv = max(max( (4/3).*mu.*(gamma*mu/Pr)./rho ));
9
10 dv =  inf;
11 for j = 1:jmax
12     for k = 1:kmax
13
14         rho_val = p(j,k) / (R * T(j,k));
15         mu_val = mu0 * (T(j,k) / T0)^1.5 * (T0 + 110)./(T
               (j,k) + 110);
16
17         val = (4/3) * mu_val * (gamma*mu_val/Pr) /
               rho_val; % find the max of this
18         if val > dv
19             dv = val;
20         end
21     end
22 end
23
```

```
24
25
26
27 spaceUnit = sqrt(1/dx^2 + 1/dy^2);
28
29 % dt = min(min( K * dt_cfl ));
30 dt = inf;
31 for j = 1:jmax
32     for k = 1:kmax
33
34         rho_val = p(j,k) / (R * T(j,k));
35
36         term1 = abs(u(j,k)) / dx;
37         term2 = abs(v(j,k)) / dy;
38         term3 = sqrt(gamma * p(j,k) / rho_val) *
               spaceUnit;
39         term4 = 2 * dv * spaceUnit*spaceUnit;
40
41         dt_cfl = 1./(term1 + term2 + term3 + term4);
42
43         % multiply by Courant number as a "fudge factor"
               and take the min
44         if K*dt_cfl < dt
45             dt = K*dt_cfl;
46         end
47
48     end
49 end
50
51
52
53 end
```

calc_Q.m

```
1  function Q = calc_Q(Q, u, v, p, T, j, k)
2
3  % kernel function
4
5  global R Cv
6
7  rho_val = p(j,k) ./ (R * T(j,k));
8  e_val = Cv * T(j,k); % energy of air based on temp
9  Et = rho_val .* (e_val + (u(j,k)^2 + v(j,k)^2)/2); %
       total energy
10
11 Q(j,k,1) = rho_val;
12 Q(j,k,2) = rho_val * u(j,k);
13 Q(j,k,3) = rho_val * v(j,k);
14 Q(j,k,4) = Et;
```

```

end
```

calc_FG.m

```
function [F, G] = calc_FG(F, G, u, v, p, T, isPredictor, j, k)

% kernel function

% lets also track mu from this function since I think the only other place
% that needs it is the time step update


% update F and G for location (j,k)
% inputs are assumed to be entire matrices

global R Cv T0 mu0

rho_val = p(j,k) / (R * T(j,k));
e_val = Cv * T(j,k); % energy of air based on temp
Et_val = rho_val * (e_val + (u(j,k)^2 + v(j,k)^2)/2); % total energy

% Sutherlands Law:
mu_val = mu0*(T(j,k) / T0).^1.5 * (T0 + 110)./(T(j,k) + 110);

[qx, qy] = heatFluxParameters(T, mu_val, isPredictor, j, k);
[txx, tyy, txy_F, txy_G] = shearParameters(u, v, mu_val, isPredictor, j, k);

F(j,k,1) = rho_val * u(j,k);
F(j,k,2) = rho_val * u(j,k)^2 + p(j,k)   txx;
F(j,k,3) = rho_val * u(j,k) * v(j,k)   txy_F;
F(j,k,4) = (Et_val + p(j,k)) * u(j,k)   u(j,k) * txx   v(j,k) * txy_F + qx;

G(j,k,1) = rho_val * v(j,k);
G(j,k,2) = rho_val * u(j,k) * v(j,k)   txy_G;
G(j,k,3) = rho_val * v(j,k)^2 + p(j,k)   tyy;
G(j,k,4) = (Et_val + p(j,k)) * v(j,k)   u(j,k) * txy_G   v(j,k) * tyy + qy;


% for debugging:
F1 = F(:,:,1);
```

```
F2 = F(:,:,2);
F3 = F(:,:,3);
F4 = F(:,:,4);
G1 = G(:,:,1);
G2 = G(:,:,2);
G3 = G(:,:,3);
G4 = G(:,:,4);


end
```

heatFluxParameters.m

```
function [qx, qy] = heatFluxParameters(T, mu_val,
    isPredictor, j, k)

% calculate the heat flux parameters for location (j,k)
% inputs T and mu are entire matrices

global Cp Pr dx dy jmax kmax

k_cond = mu_val * Cp / Pr; % conductivity parameter using
     mu from Sutherlands Law


if isPredictor % scheme is forward, make this backward

    if j > 1
        dTdx = (T(j,k)    T(j-1,k))/dx;
    else
        dTdx = (T(2,k)    T(1,k))/dx;
    end

    if k > 1
        dTdy = (T(j,k)    T(j,k-1))/dy;
    else
        dTdy = (T(j,2)    T(j,1))/dy;
    end

else

    if j < jmax
        dTdx = (T(j+1,k)    T(j,k))/dx;
    else
        dTdx = (T(jmax,k)    T(jmax-1,k))/dx;
    end

    if k < kmax
        dTdy = (T(j,k+1)    T(j,k))/dy;
    else
```

```
36          dTdy = (T(j,kmax)    T(j,kmax-1))/dy;
37      end
38
39  end
40
41  qx =  k_cond * dTdx;
42  qy =  k_cond * dTdy;
43
44
45
46  end
```

shearParameters.m

```
1   function [txx, tyy, txy_F, txy_G] = shearParameters(u, v,
        mu, isPredictor, j, k)
2
3   % calculate shear for a single location (j,k)
4   % inputs are assumed to be entire matrices
5
6   global dx dy jmax kmax
7
8
9   %% calculate the forward or backward differenced versions
10  if isPredictor
11      % want opposite direction from scheme step
            differencing
12      % scheme is forward, make this backward
13
14      if j > 1
15          dvdx_FB = (v(j,k)    v(j-1,k))/dx;
16          dudx_FB = (u(j,k)    u(j-1,k))/dx;
17      else
18          dvdx_FB = (v(2,k)    v(1,k))/dx; % except first
                point forward
19          dudx_FB = (u(2,k)    u(1,k))/dx; % except first
                point forward
20      end
21
22      if k > 1
23          dudy_FB = (u(j,k)    u(j,k-1))/dy;
24          dvdy_FB = (v(j,k)    v(j,k-1))/dy;
25      else
26          dudy_FB = (u(j,2)    u(j,1))/dy; % except first
                point forward
27          dvdy_FB = (v(j,2)    v(j,1))/dy; % except first
                point forward
28      end
29
30  else
```

```
31
32      % scheme is backward, make this forward
33
34      if j < jmax
35          dvdx_FB = (v(j+1,k)    v(j,k))/dx;
36          dudx_FB = (u(j+1,k)    u(j,k))/dx;
37      else
38          dvdx_FB = (v(j,k)    v(j-1,k))/dx; % except jmax
                backward
39          dudx_FB = (u(j,k)    u(j-1,k))/dx; % except jmax
                backward
40      end
41
42      if k < kmax
43          dudy_FB = (u(j,k+1)    u(j,k))/dy;
44          dvdy_FB = (v(j,k+1)    v(j,k))/dy;
45      else
46          dudy_FB = (u(j,kmax)    u(j,kmax-1))/dy; % except
                kmax backward
47          dvdy_FB = (v(j,kmax)    v(j,kmax-1))/dy; % except
                kmax backward
48      end
49
50  end
51
52  %% and then we want centeral differenced versions
53
54  if j == 1
55      dvdx_C = (v(2,k)    v(1,k))/dx;
56      dudx_C = (u(2,k)    u(1,k))/dx;
57  elseif j == jmax
58      dvdx_C = (v(jmax,k)    v(jmax-1,k))/dx;
59      dudx_C = (u(jmax,k)    u(jmax-1,k))/dx;
60  else
61      dvdx_C = (v(j+1,k)    v(j-1,k))/(2*dx);
62      dudx_C = (u(j+1,k)    u(j-1,k))/(2*dx);
63  end
64
65
66  if k == 1
67      dudy_C = (u(j,2)    u(j,1))/dy;
68      dvdy_C = (v(j,2)    v(j,1))/dy;
69  elseif k == kmax
70      dudy_C = (u(j,kmax)    u(j,kmax-1))/dy;
71      dvdy_C = (v(j,kmax)    v(j,kmax-1))/dy;
72  else
73      dudy_C = (u(j,k+1)    u(j,k-1))/(2*dy);
74      dvdy_C = (v(j,k+1)    v(j,k-1))/(2*dy);
75  end
76
```

```
77
78 % these come from page 65 and 66 in Anderson
79
80 lambda = (2/3) * mu; % second viscosity coefficient
       estimated by Stokes
81
82 % use the forward/backward du/dx and central dv/dy for
       both F and G
83 txx = lambda * ( dudx_FB + dvdy_C ) + 2 * mu * dudx_FB;
84
85 % use the forward/backward dv/dy and central du/dx for
       both F and G
86 tyy = lambda * ( dudx_C + dvdy_FB ) + 2 * mu * dvdy_FB;
87
88 txy_F = mu * ( dvdx_FB + dudy_C );
89 txy_G = mu * ( dvdx_C + dudy_FB );
90
91
92
93 end
```

MaccormackPredictorUniform.m

```
1  function Q_pred = MaccormackPredictorUniform(Q_pred, Q, F
       , G, dt, j, k)
2
3  % kernel function
4
5  global dx dy jmax kmax
6
7  % DO MACCORMACKS FOR FOR INTERIOR POINTS ONLY
8  % if an edge point, just do nothing
9  if j == 1 || k == 1 || j == jmax || k == kmax
10     return;
11 end
12
13
14 % have each thread calculate all 4 dimensions at a single
       loc
15 for dim = 1:4
16
17     Q_pred(j,k,dim) = Q(j,k,dim)     dt * ( (F(j+1,k,dim)
           F(j,k,dim))/dx + (G(j,k+1,dim)     G(j,k,dim))/dy )
          ;
18
19 end
20
21
22 end
```

primativesFromQ.m

```
function [rho, u, v, e, p, T] = primativesFromQ(rho, u, v, e, p, T, Q, j, k)

global Cv R

rho_val = Q(j,k,1); % just to reduce calls to variables in global memory

rho(j,k) = rho_val;
u(j,k) = Q(j,k,2) / rho_val;
v(j,k) = Q(j,k,3) / rho_val;
e(j,k) = Q(j,k,4) / rho_val   0.5*( u(j,k)^2 + v(j,k)^2 );


% actually want to update p and T here too
T(j,k) = e(j,k) / Cv;
p(j,k) = rho_val * R * T(j,k);


end
```

enforceBC_nonSurface.m

```


function [u, v, p, T] = enforceBC_nonSurface(u, v, p, T, j, k)

% need to first establish all the boundary conditions at the non surface
% values, and then go back and do the surface boundary conditions

% this is really only needed if the surface goes all the way to the outflow
% so that the last surface point can be interpolated with updated values

global p0 T0 u0 kmax jmax


if j == 1 && k == 1  % leading edge
    u(j,k) = 0;
    v(j,k) = 0;
    p(j,k) = p0;
    T(j,k) = T0;

```

```
20 elseif j == 1 || k == kmax  % inflow from upstream OR
       upper boundary
21     u(j,k) = u0;
22     v(j,k) = 0;
23     p(j,k) = p0;
24     T(j,k) = T0;
25
26 elseif j == jmax % outflow
27     % extrapolate from interior values
28 %     u(j,k) = 2*u(j,k-1)   u(j,k-2);
29 %     v(j,k) = 2*v(j,k-1)   v(j,k-2);
30 %     p(j,k) = 2*p(j,k-1)   p(j,k-2);
31 %     T(j,k) = 2*p(j,k-1)   p(j,k-2);
32     u(j,k) = 2*u(j-1,k)   u(j-2,k);
33     v(j,k) = 2*v(j-1,k)   v(j-2,k);
34     p(j,k) = 2*p(j-1,k)   p(j-2,k);
35     T(j,k) = 2*T(j-1,k)   T(j-2,k);
36 end
37
38
39 end
```

enforceBC_surface.m

```
1
2 function [u, v, p, T] = enforceBC_surface(u, v, p, T, j,
      k, adiabaticWall)
3
4 % need to first establish all the boundary conditions at
      the non surface
5 % values, and then go back and do the surface boundary
      conditions
6
7 % this is really only needed if the surface goes all the
      way to the outflow
8 % so that the last surface point can be interpolated with
       updated values
9
10 global Twall
11
12 if k == 1 && j > 0
13     u(j,k) = 0;
14     v(j,k) = 0;
15     p(j,k) = 2*p(j,k+1)   p(j,k+2);
16
17     if adiabaticWall
18         T(j,k) = T(j,k+1);
19     else
20         T(j,k) = Twall;
21     end
```

```
22 end
```

MaccormackCorrectorUniform.m

```
1  function Q = MaccormackCorrectorUniform(Q, Q_pred, F_pred
       , G_pred, dt, j, k)
2
3  % kernel function
4
5  global dx dy jmax kmax
6
7  % DO MACCORMACKS FOR FOR INTERIOR POINTS ONLY
8  % if an edge point, just do nothing
9  if j == 1 || k == 1 || j == jmax || k == kmax
10     return;
11 end
12
13
14 % have each thread calculate all 4 dimensions at a single
       loc
15 for dim = 1:4
16
17     flux = (F_pred(j,k,dim)    F_pred(j 1,k,dim))/dx + (
           G_pred(j,k,dim)    G_pred(j,k 1,dim))/dy;
18     Q(j,k,dim) = 0.5*( Q(j,k,dim) + Q_pred(j,k,dim)    dt
           * flux);
19
20 end
21
22
23 end
```