Project 2: Vortex Transport

AE 623, Advanced Computational Fluid Dynamics, Winter 2017

Due: March 5, 11:55pm, electronically via Canvas

Introduction

In this project you will simulate the transport of an isentropic vortex in uniform flow, on a square domain with periodic boundaries. Accurate transport of vortices is important in the analysis of many aeronautical systems, including fixed-wing aircraft and rotorcraft. High-order resolution in space and time, on unstructured meshes, is vital in preserving vorticity and minimizing the introduction of spurious numerical errors that dissipate the vortex. Figure 1 shows the setup for the particular vortex problem you will be considering.

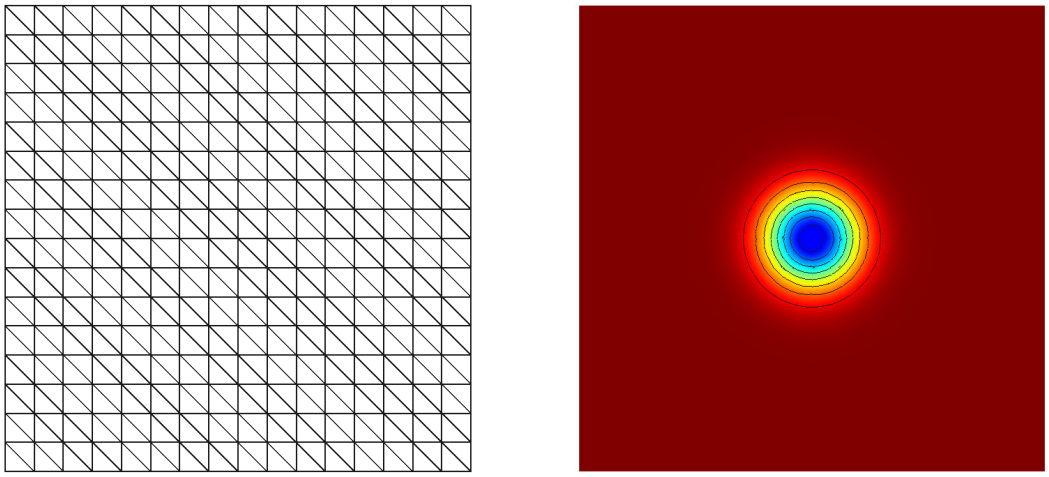


Figure 1: Unperturbed mesh ($N = 16$) and initial pressure field.

Problem Specification

Analytical Solution and Units

Use the following analytical vortex solution to the Euler equations: the state vector at point $\vec{x} = (x, y)$ and time t is

$$
\begin{aligned}
\mathbf{u} &= \left[\rho, \rho u, \rho v, \frac{p}{\gamma - 1} + \frac{1}{2}\rho(u^2 + v^2)\right] \\
\rho &= \rho_\infty f_1^{1/(\gamma-1)} \\
u &= U_\infty - f_2(y - y_0 - V_\infty t) \\
v &= V_\infty + f_2(x - x_0 - U_\infty t) \\
p &= p_\infty f_1^{\gamma/(\gamma-1)}
\end{aligned}
$$

where ρ is the density, (u, v) are velocity components and p is the pressure (not to be confused with approximation order), and

$$f_0 = 1 - \frac{|\vec{x} - \vec{x}_0 - \vec{V}_\infty t|^2}{r_c^2}, \quad f_1 = 1 - \epsilon^2(\gamma - 1)M_\infty^2 \frac{e^{f_0}}{8\pi^2}, \quad f_2 = \epsilon \frac{|\vec{V}_\infty|}{2\pi r_c} e^{f_0/2}.$$

The subscript ∞ denotes a background "free-stream" state without the vortex perturbation, and the free-stream Mach number is $M_\infty = |\vec{V}_\infty|/c_\infty$, $c_\infty = \sqrt{\gamma p_\infty/\rho_\infty}$. We use the following values, in convenient units: $\vec{x}_0 = (0, 0)$, $\vec{V}_\infty = (U_\infty, V_\infty) = (1, 1)/\sqrt{2}$, $\rho_\infty = 1$, $M_\infty = 0.5$, $\gamma = 1.4$, $\epsilon = 0.3$, $r_c = 1.0$. The time horizon of interest is up to $T = 10\sqrt{2}$, which is the time at which the vortex returns to its starting location.

Geometry, Initial, and Boundary Conditions

The geometry for this problem is a square, $(x, y) \in [-5, 5]^2$, with periodic boundaries. The initial condition is the analytical vortex solution centered in the middle of the domain. You should investigate two approaches for initializing the solution in the finite-dimensional approximation space:

1. Direct interpolation of the analytical solution to equally-spaced Lagrange nodes.
2. Least-squares projection of the analytical solution to the order p approximation space.

Note, in the second case, the coefficients U_j in the solution expansion $u_h(\vec{x}) = \sum_j U_j \phi_j(\vec{x})$ are determined by minimizing the square error,

$$E^2 = \int_\Omega [u_h - u_{\text{exact}}]^2 \, d\Omega, \tag{1}$$

where the integral is taken over one element Ω (all elements are treated independently). Differentiating with respect to U_j to determine the minimum of E^2 gives

$$\begin{aligned} \frac{\partial E^2}{\partial U_i} = 0 \quad \Rightarrow \quad \int_\Omega 2[u_h - u_{\text{exact}}]\phi_i \, d\Omega &= 0 \\ \int_\Omega \phi_i u_h \, d\Omega &= \int_\Omega \phi_i u_{\text{exact}} \, d\Omega \\ \mathbf{MU} &= \mathbf{b} \\ \mathbf{U} &= \mathbf{M}^{-1}\mathbf{b}, \end{aligned}$$

where $\mathbf{M}$ is the mass matrix and $b_i = \int_\Omega \phi_i u_{\text{exact}} \, d\Omega$. So in addition to the calculation of the mass matrix, least-squares projection needs only weighted integrals of the exact state. For systems of equations, this projection is done separately for each state component.

Computational Meshes

Let N be the number of intervals in the x and y directions. Obtain an *unperturbed* mesh by dividing the domain into N^2 squares, and then splitting each square into two right triangles, for a total of $2N^2$ elements, as shown in Figure 1. Obtain a *perturbed* mesh by displacing

each interior node of the structured mesh according to $\delta x = 0.2h\sin((x+6)^{y+6})$, $\delta y = 0.2h\cos((y+6)^{x+6})$, where (x, y) are the original coordinates of the nodes in the unperturbed mesh, and $h = 10/N$ is the interval spacing.

Discretization

Use the discontinuous Galerkin [1] method with support for orders $p = 0, 1, 2, 3$. Implement the Roe flux for the Euler equations. Integrate in time using fourth-order Runge-Kutta with a stable time step. A compiled programming language is recommended for this project, to keep execution times reasonable.

Post-processing and Error Measure

Visualize high-order fields by subdividing each element into a large number of triangles and plotting a linear solution on each triangle. For the outputs, consider the following error measures (note, Ω is the entire domain):

L_2 state error:

$$\epsilon_{\mathbf{u}} = \left[\frac{1}{|\Omega|}\int_\Omega |\mathbf{u}(T) - \mathbf{u}(0)|^2 d\Omega\right]^{1/2}$$

Entropy error:

$$\begin{aligned} \epsilon_s &= \left[\frac{1}{|\Omega|}\int_\Omega s(T)^2 d\Omega\right]^{\frac{1}{2}}, \\ s(t) &= \ln\frac{p(t)}{p_\infty} - \gamma\ln\frac{\rho(t)}{\rho_\infty} \end{aligned}$$

Pressure functional error:

$$\begin{aligned} \epsilon_p &= J_p(T) - J_p^{\text{exact}}(0) \\ J_p(t) &= \frac{1}{|\Omega|}\int_\Omega w(\vec{x})p(\vec{x}, t)d\Omega \\ w(\vec{x}) &= r^4 e^{-2r^2} \\ r &= \sqrt{x^2 + y^2} \end{aligned}$$

Tasks

1. [**10%**] Write a code to generate unperturbed and perturbed triangular meshes of the domain for a given N.

[1] Professor Roe is currently developing a method designed as a competitor to DG methods. He believes that his Active Flux method (which he will talk about later in the semester) is more accurate on a given grid, but we presently have little idea about how the methods compare in cost. If this project goes well for you, you might consider joining up with Prof. Roe and his students to explore this. It probably involves going a bit beyond what is required for the project, because it is important when comparing two methods that they be equally well-coded. If you would be interested in participating, have a word with Prof Roe. You will probably get at least an acknowledgment in a published paper.

2. [**40%**] Write a DG solver for this problem, and describe all aspects of the implementation, including: the flux calculation, periodic boundary conditions, quadrature, basis functions, time step calculation, storage versus recomputation choices.
3. [**15%**] Run simulations up to $t = T = 10\sqrt{2}$, using $p = 2$ and $p = 3$, and $N = 8$ and $N = 16$ on the perturbed and unperturbed meshes (8 runs total). Make field plots of u and p, and comment on any differences.
4. [**15%**] Perform a convergence study for each error measure, at orders $p = 0, 1, 2, 3$, using $N = 8, 16, 32, 64$. Use interpolation for the state initialization, and adequate quadrature rules for the integrations. Comment on the effect of order on the errors and convergence rates.
5. [**10%**] Study the impact of interpolation versus least-squares projection of the initial condition on the different errors. Does the initialization affect the error magnitude? Does it affect the error convergence rate?

Deliverables

You should turn in **two files** electronically via Canvas:

1. A technical report as a `.pdf` file that describes your methods and results, and that addresses all of the above tasks. The report should be professional, complete, and concise. **10%** of your grade will be determined by the professionalism (neatness, labeling of axes, spelling, etc.) of your report.
2. Documented source files of all codes you wrote for this project, as one `.zip` archive.

This is an individual assignment. You can discuss the project at a high level with each other, but you must turn in your own work.

University of Michigan

Computational Fluid Dynamics II

Project II

Eric Parish

March 3, 2017

1 Overview

This document details the development of a 2D unstructured discontinuous Galerkin (DG) solver. A convenient feature of the developed solver is that it is implemented purely in Python. The solver makes use of modules included in the numpy package to efficiently compute the required residual terms that appear in the DG formulation.

2 The Discontinuous Galerkin Approach

The DG approach is a methodology for solving partial differential equations. Consider the non-linear conservation equation

$$\frac{\partial \mathbf{u}}{\partial t} = \nabla \cdot F(\mathbf{u}). \tag{2.1}$$

We proceed by casting Eq. 2.1 into the variational form. Define the test functions

$$\mathbf{v} \in \mathcal{V}.$$

Multiplying Eq. 2.1 with the test functions and integrating over the domain

$$\int \frac{\partial \mathbf{u}}{\partial t} v_j \Omega = \int \nabla \cdot F(\mathbf{u}) v_j \Omega. \tag{2.2}$$

The DG approach proceeds by integrating the right hand side of 2.2 by parts,

$$\int \frac{d\mathbf{u}}{dt} v_j d\Omega = -\int \mathbf{F} \cdot \nabla v_j d\Omega + \oint \mathbf{F} v_j \cdot d\mathbf{S}. \tag{2.3}$$

In the DG approach, the spatial domain Ω is sub-divided into N elements. On each element the solution is described by set of local basis functions.

2.1 Coordinate Transformations

Since each basis function is expressed in a local coordinate frame, it is desirable to evaluate the governing equations in a local coordinate frame. To do so consider the tensor mapping (which we will call the Jacobian)

$$J_j^i = \frac{\partial x^i}{\partial \zeta^j}.$$

On a standard element, the mapping becomes

$$\begin{Bmatrix} dx \\ dy \end{Bmatrix} = \begin{bmatrix} (x_1 - x_0) & (x_2 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) \end{bmatrix} \begin{Bmatrix} d\zeta \\ d\eta \end{Bmatrix}.$$

Similarly the inverse mapping can be obtained,

$$\overline{J}_j^i = \frac{\partial \zeta^i}{\partial x^j},$$

which can be evaluated to be,

$$\begin{Bmatrix} d\zeta \\ d\eta \end{Bmatrix} = \frac{1}{J} \begin{bmatrix} (y_2 - y_0) & (x_0 - x_2) \\ (y_0 - y_1) & (x_1 - x_0) \end{bmatrix} \begin{Bmatrix} dx \\ dy \end{Bmatrix}$$

where the J is the determinant of the forward mapping $\mathbf{J}$. The metrics of the mapping can be used to evaluate integrals and derivatives in a reference coordinate frame. To make use of the metrics, return to the variational form of the governing system,

$$\int \frac{d\mathbf{u}}{dt} v_j d\Omega = -\int \mathbf{F} \cdot \nabla v_j d\Omega + \oint \mathbf{F} v_j \cdot d\mathbf{S}. \tag{2.4}$$

Since the test functions v_j are generally defined in a local coordinate system, the chain rule needs to be used for differentiation. First, note that the determinant of the Jacobian represents the volume swept out in the local coordinate system to the volume swept out in the global coordinate system. As such, the volume and surface integrals can be computed by

$$\int \frac{d\mathbf{u}}{dt} v_j J d\omega = -\int \mathbf{F} \cdot \nabla v_j J d\omega + \oint J_{edge} v_j \mathbf{F} \cdot \mathbf{n} ds \tag{2.5}$$

where J_{edge} is the arc length ratio along the edge of the element between the global coordinate system and the local coordinate system. Since the gradients are still in a global frame, we also need to compute the contravariant derivatives. They can be computed via the chain rule,

$$\frac{\partial f}{\partial x^j} = \frac{\partial f}{\partial \zeta^i} \frac{\partial \zeta^i}{\partial x^j}.$$

3 Solver Details

This section will detail aspects of the developed solver.

3.1 Meshing

The developed solver is designed to operate on two-dimensional unstructured grids. The grids themselves are constructed using scipy's delauny triangulation routines. These routines take in a set of x and y coordinate points and return the delauny triangulation of the points. The package, by default, computes the vertices associated with each element as well as the neighboring elements. Several additional routines are needed to construct the required connectivities needed for a DG solver. Most notably, a routine that constructs the edge connectivities is required.

This calculation is performed within the "edgeHash" routine. The routine takes as input the triangle class produced by the Delauny triangulation and, for each edge, computes the associated two vertices, two elements, as well as the local edge number on each element. The routine first determines the interior edges, and then the boundaries edges. To construct a periodic mesh, the routine loops over the x boundary edges and pairs the edges if the two y-coordinates are the same. The same loop is done for the y boundary edges, where the pairing is then done by checking if the two x-coordinates are the same.

3.2 Basis Functions

The solver considers a triangular discretization of Ω. On each element, the local basis functions are taken to be a full-order Lagrange basis with degrees of freedom located at equally space intervals throughout the triangle. To develop this basis, first consider a regular element in reference space ω, where ω has coordinates $(\omega_1, \omega_2) = (\zeta, \eta) \in [0, 1]$. In this space a full-order polynomial can be written as

$$\phi_j(\zeta, \eta) = \sum_{s=0}^{p} \sum_{r=0}^{p-s} c_k \zeta^r \eta^s.$$

To provide clarity, we write the resulting polynomials for up to order $p = 3$,

$$p = 0 \rightarrow L(x, y) = c_0$$

$$p = 1 \rightarrow L(x, y) = c_0 + c_1\zeta + c_2\eta$$

$$p = 2 \rightarrow L(x, y) = c_0 + c_1\zeta + c_2\zeta^2 + c_3\eta + c_4\zeta\eta + c_5\eta^2$$

$$p = 3 \rightarrow L(x, y) = c_0 + c_1\zeta + c_2\zeta^2 + c_3\zeta^3 + c_4\eta + c_5\eta\zeta + c_6\eta\zeta^2 + c_7\eta^2 + c_8\zeta\eta^2 + c_9\eta^3.$$

Note that the above are not Lagrange basis. To determine the Lagrange basis functions, we set up systems of equations that enforce the i_{th} basis function to be zero at every interpolation point except i. For $p = 1$ we interpolate the following points,

$$p_1 = (0, 0), p_2 = (1, 0), p_3 = (0, 1)$$

which leads to

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{Bmatrix} c_0 \\ c_1 \\ c_2 \end{Bmatrix} = \delta_i.$$

This gives us the basis functions,

$$\ell_0 = 1 - \zeta - \eta, \ell_1 = \zeta, \ell_2 = \eta.$$

For $p = 2$ we have,

$$p_1 = (0, 0), p_2 = (1, 0), p_3 = (0, 1), p_4 = (0.5, 0), p_5 = (0, 0.5), p_6 = (0.5, 0.5)$$

which gives

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0.5 & 0.25 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0.5 & 0 & 0.25 \\ 1 & 0.5 & 0.25 & 0.5 & 0.25 & 0.25 \\ 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{Bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{Bmatrix} = \delta_i.$$

We get,

$$\ell_0 = 2\eta^2 + 4\eta\zeta - 3\eta + 2\zeta^2 - 3\zeta + 1$$

$$\ell_1 = 4\zeta - 4\eta\zeta - 4\zeta^2$$
$$\ell_2 = 2\zeta^2 - \zeta$$
$$\ell_3 = 4\eta - 4\eta\zeta - 4\eta^2$$
$$\ell_4 = 4\eta\zeta$$
$$\ell_5 = 2\eta^2 - \eta.$$

The same procedure can be followed to develop higher order basis functions.

3.3 Flux Functions

The DG approach requires the computation of a flux at the element edges. Here, the Roe flux is used. Compactly, the Roe flux is given by

$$\mathbf{F}^+ = \frac{1}{2}\Big(\mathbf{F}^+ + \mathbf{F}^-\Big) - \frac{1}{2}\big|\mathbf{A}(\mathbf{u}^*)\big|\Big(\mathbf{u}_R - \mathbf{u}_L\Big),$$

where $\mathbf{A} = \frac{\partial \mathbf{F}}{\partial \mathbf{u}}$. The Roe average state is given by

$$\rho^* = \sqrt{\rho_R \rho_L},$$
$$u^* = \frac{\sqrt{\rho_L}u_L + \sqrt{\rho_R}u_R}{\sqrt{\rho_R} + \sqrt{\rho_L}},$$
$$h^* = \frac{\sqrt{\rho_L}h_{0,L} + \sqrt{\rho_R}h_{0,R}}{\sqrt{\rho_R} + \sqrt{\rho_L}}.$$

The flux is computed by transforming the two-dimensional flux into a coordinate system aligned with the element interface. An entropy fix is additionally used.

3.4 Quadrature

The (2D) DG approach additionally requires the evaluation of surface and edge integrals. In practice, these integrals can not be performed analytically and, instead, numerical integration is used. In the presented solver, both surface and edge integrals are evaluated via Gaussian quadrature. The surface integration is performed by reconstructing the state variables at the Dunavant points, while the edge integrations are performed by reconstructing the state variables at the Gauss points on each edge. The solver takes the order of quadrature as input.

3.5 Visualization

The presented code uses vtk libraries to output the unstructured mesh as a .vtu file that can be visualized in Paraview. High order data is presented by creating a second triangulation of the Lagrange points.

3.6 Numerical Considerations for efficient computing in Python

The development of an efficient numerical solver in python requires special considerations. Python is a high level language, and control statements such as for loops are inefficient and not practical for large scale calculations. Considerable effort, however, has been put into the development of python routines that can be used to efficiently replace slow control statements. Further, python supports array slicing which in itself eliminates the need for control structures and allows for compact representation of code. To illustrate how these features are used to develop an efficient and concise DG solver, we consider various equivalent representations of three different critical routines in a DG solver.

3.6.1 Quadrature Reconstruction

First consider a routine that, given the basis coefficients a and basis functions v, evaluates the resulting polynomial at the x_j quadrature points. The basis coefficient is of dimension $n \times o \times k$, where n is the number of state variables (4 for Euler), o is the number of basis functions $(p+1)(p+2)/2$, and k is the total number of elements. The dimension of the v is $o \times i$, where again o is the number of basis functions and i is the number of quadrature points. The reconstruction of the state variable is given by

$$u_{nik} = v_{oi} a_{nok}, \tag{3.1}$$

where a summation over o is implied. The naive numerical implementation Eq. 3.1 is

Listing 1: Naive numerical implementation of Eq. 3.1

```
for n in range(0,nvars):
  for o in range(0,(p+1)*(p+2)/2)):
    for k in range(0,Nel)
      u[n,i,k] += v[o,i]*a[n,o,k]
```

The above algorithm, while effective in a low-level language, will be extremely slow in python. The above can be dramatically sped up by making use of python's slicing abilities,

Listing 2: Improved numerical implementation of Eq. 3.1

```
for n in range(0,nvars):
  for o in range(0,(p+1)*(p+2)/2)):
      u[n,i,:] += v[o,i]*a[n,o,:]
```

The improved algorithm presented in Listing 2 has eliminated the $\mathcal{O}(N)$ for loop and will be much improved. However, improvements in both conciseness and efficiency can still be made by utilizing numpy's einsum algorithm. Eq. 3.1 can be concisely represented as

Listing 3: Efficient numerical implementation of Eq. 3.1

```
u = np.einsum('oi,nok->nik',v,a)
```

The total time per function call is given in Table 3.6.1, where it i seen that Listing 3 is one thousand times faster than Listing 1.

Table 3.1: Summary of computational times for Listings 1,2, and 3.

	Listing 1	Listing 2	Listing 3
CPU Time	3.68 s	5.5 ms	3.89 ms
Speed Up	1	0.64×10^3	0.97×10^3

3.6.2 Numerical surface integration

In a similar fashion, consider the evaluation of the numerical surface integral. Evaluation of a surface integral requires three arrays: the quadrature weights, w, the function to be integrated

at the quadrature points, U, and the determinant of the Jacobian, J. The dimensions of these arrays are $w = i \times 1$, $U = n \times i \times k$, and $J = k \times 1$ where i is the number of quadrature points, n is the number of state variables, and k is the number of elements. The naive integration is performed by

Listing 4: Naive numerical integration

```
for n in range(0,nvars):
  for k in range(0,Nel):
    for i in range(0,quadpoints):
      integ[n,k] += w[i]*U[n,i,k]*J[i]
```

Like previously, this integration will be extremely slow. Directly writing the computation using python's slicing techniques is challenging due to the fact that the arrays are different dimensions. It can be efficiently written, however, using numpy's broadcasting functionality and numpy's sum function,

Listing 5: Improved numerical integration

```
integ = np.sum(w[None,:,None]*U*J[None,None,:],axis=1).
```

The broadcasting functionality adds dummy indices to the weighting vector and Jacobian vector and allows for efficient computation. However, the above can be even further sped up by again using the einsum function,

Listing 6: Efficient numerical integration

```
integ = np.einsum('i,nik->nk',w,U)*J[None].
```

The performance of each listing is given in Table 3.6.2. It is again seen that making of the einsum functionality provides over a 2000× speed up over the naive implementation, and another $4x$ speedup over what is already an efficient implementation in Listing 5.

Table 3.2: Summary of computational times for Listings 4,5, and 6.

	Listing 4	Listing 5	Listing 6
CPU Time	436 ms	781 μs	200 μs
Speed Up	1	560	2180

3.6.3 Flux Evaluation

Lastly, we comment on the flux evaluation routine. Unlike a structured mesh, unstructured meshes have no natural ordering that can be used to make use of slicing. For example, a central difference can not be computed by

Listing 7: Central difference computation for a 1D structured mesh.

```
F = 0.5/dx*(u[2::] - u[0:-2])
```

For unstructured meshes, it appears at first glance that one will have to loop over the edges, look up the edge connectivities, and then compute the flux for each edge. This loop will be of $\mathcal{O}(N)$ and will bottleneck the performance of the code. However, python's slicing has a very nice feature in that the an array can be used as the arguments for the slicing. For example,

Listing 8: Non-uniform numpy slicing.

```
a = np.array([5,1,2])
b = np.array([2,1,0])
=================
a[b] = array([2,1,5])
```

We can efficiently make use of this functionality for unstructured meshes by passing the connectivity information as arguments to the arrays. In this manner the flux computation can be done without a single for loop.

Listing 9: Routine to evaluate the edge fluxes.

```
##Flux computation routine
#inputs
#normals (triangle normals stored in a 2,3,Nel sized array)
# u_edges (value of u at the edges, stored in a
# nvars,quadpoints,3,Nel sized array)
def computeFlux(normals,IE,u_edges):
    u1 = u_edges[:,:,IE[:,4],IE[:,2]]
    u2 = u_edges[:,::-1,IE[:,5],IE[:,3]]
    normals = normals[:,IE[:,4],IE[:,2]]
    flux = roeflux(u1,u2,normals) #evaluate flux for entire mesh
    return flux
```

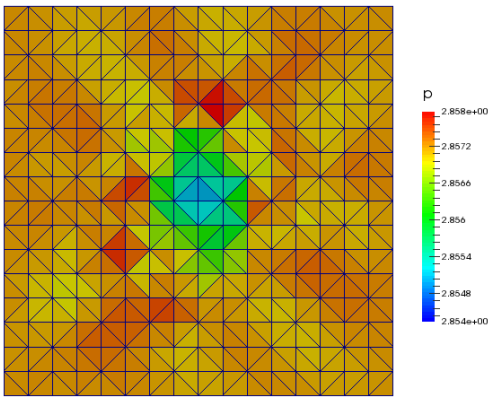
In the routine we first get the value of the state variable u at the left edge. The local edge number is stored in IE$[:, 4]$ (where the first axis in IE is sliced over the edges) and the element number is stored in IE$[:, 2]$. Next we get the value of the u at the right edge using a similar methodology. Finally we look up the normals of the edges and evaluate the flux in one global evaluation.

3.6.4 Summary of Numerical Considerations

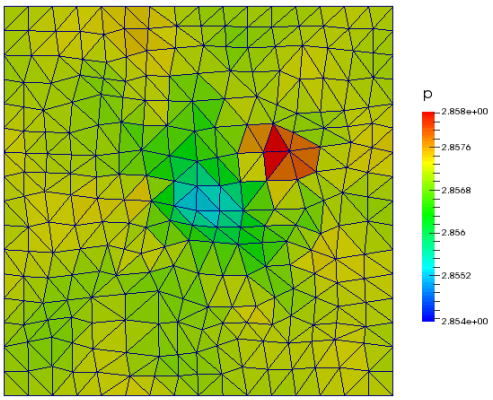
By making use of Python's slicing capabilities and built in functions, efficient concise high level code can be developed. In the presented code, there are no $\mathcal{O}(N)$ for loops that occur at the python level in the right hand side evaluation. This allows for python code to be competitive with codes developed in a low language. While it is more than likely that a code written well in a low-level language will always be faster, I believe that a well done python code can be competitive. As a sidenote, I spent some extra time on this section because I am a believer that the capabilities of python for high performance scientific computing are often underestimated.

4 Numerical Results

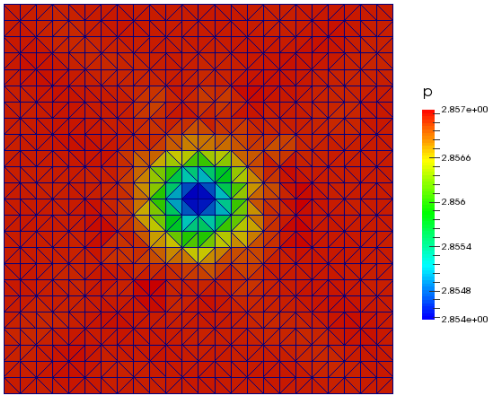
Figure 4.1 through 4.4 show plots for the numerical solution for pressure and velocity at $t = 10\sqrt{(2)}$. Figures 4.1 and 4.2 show the results for p_2 and p_3 using $N = 8$, while Figures 4.3



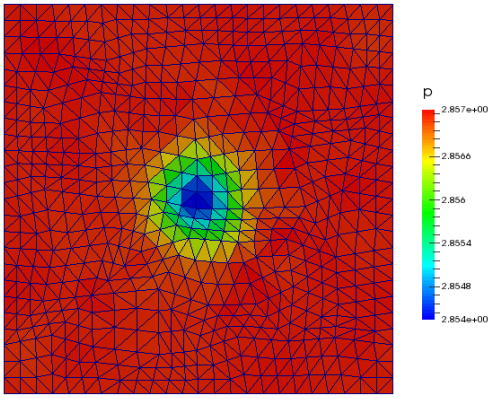
(a) $p = 2$, $N = 8$.



(b) $p = 2$, $N = 8$ on deformed mesh.



(c) $p = 3$, $N = 8$.



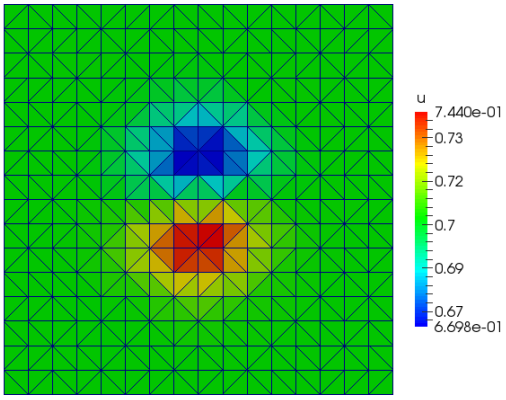
(d) $p = 3$, $N = 8$ on deformed mesh.

Figure 4.1: Pressure field at $t = 10\sqrt{2}$ with $N = 8$.

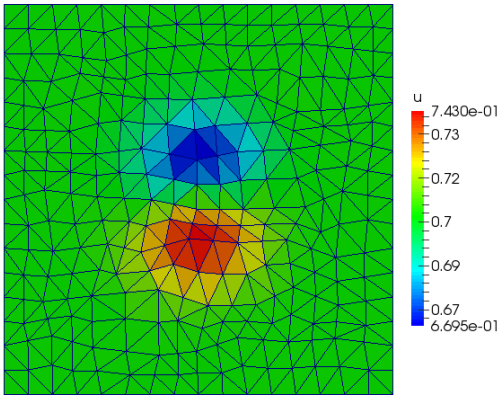
and 4.4 show the results for $N = 16$ elements. For the case where $N = 8$, it is seen that the $p = 2$ solution is unable to track the vortex across the domain. The $p = 3$ solution, however, tracks the solution reasonably and is comparable to the $p = 2, N = 16$ solutions. The deformed mesh is seen to slightly distort the vortex. In Figures 4.3 and 4.4, it is seen that the vortex is well tracked in both cases. The $p = 3$ solution does a better job.
Figure 4.5 shows the convergence of the developed solver for $p = 0, 1, 2, 3$ on meshes of $N = 8, 16, 32, 64$. All error norms are computed by taking the exact solution to be the initial solution (i.e. the initial solution after interpolation/projection). It is seen that the $p = 0$ simulations do not reach the asymptotic range of convergence for the considered meshes. All other orders of p converge with at roughly $p + 1$ for all measures of error. Computed convergence rates for the entropy error are given in Table 4.1.

4.1 Effect of the initial conditions

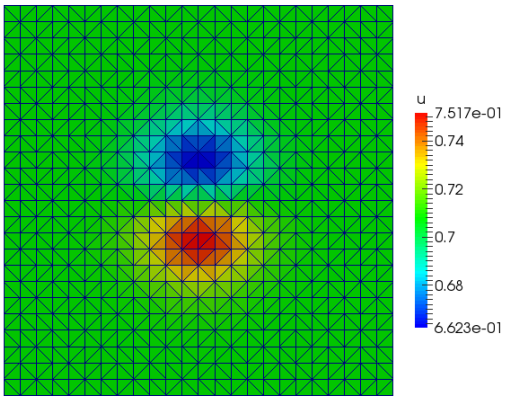
Figure 4.6 shows the convergence plots for the least-squares projection set of initial conditions. The convergence rates for the entropy error for the two different initial conditions are given in Table 4.1. The convergence rates for the projected ICs are a small bit better, but it is not as significant as what I would expect from the class discussion. I am thinking that maybe this is because I am computing errors where I am taking the exact solution to be the initial solution



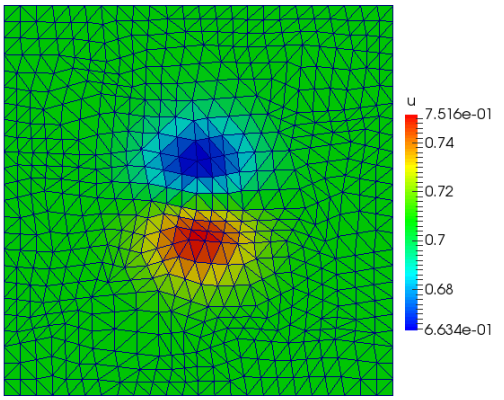
(a) $t = 10\sqrt{2}$ with $p = 2$, $N = 8$.



(b) $t = 10\sqrt{2}$ with $p = 2$, $N = 8$ on deformed mesh.

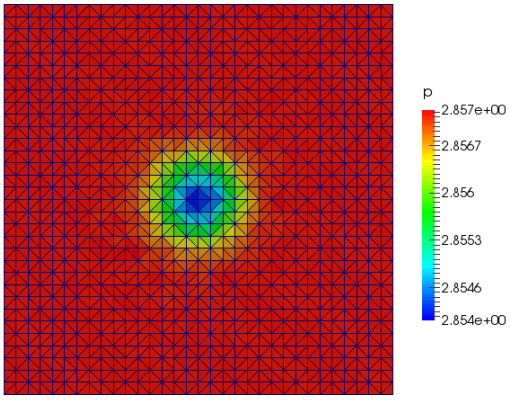


(c) $t = 10\sqrt{2}$ with $p = 3$, $N = 8$.

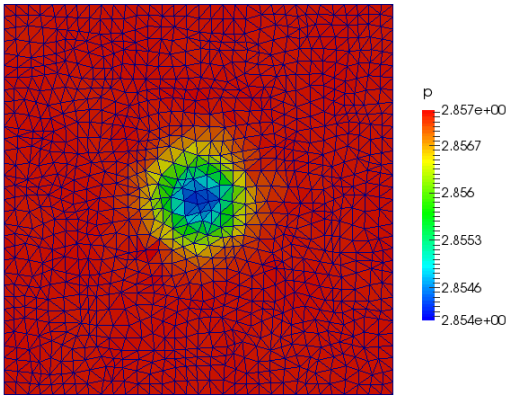


(d) $p = 3$, $N = 8$ on deformed mesh.

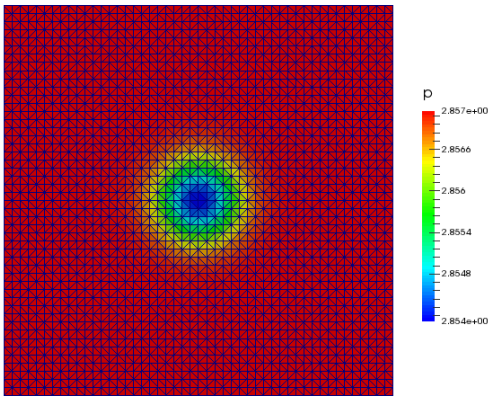
Figure 4.2: Velocity field at $t = 10\sqrt{2}$ with $N = 8$.



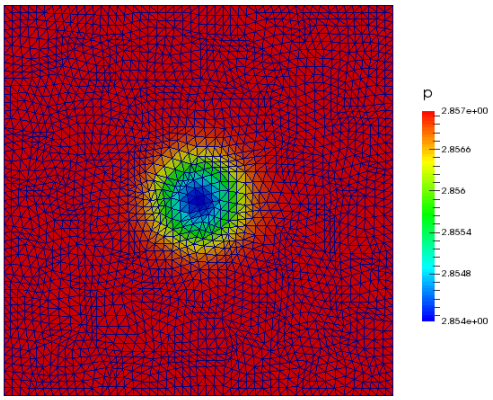
(a) $p = 2$, $N = 16$.



(b) $p = 2$, $N = 16$ on deformed mesh.

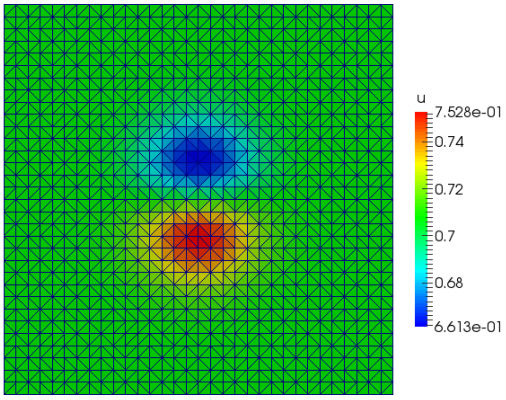


(c) $p = 3$, $N = 16$.

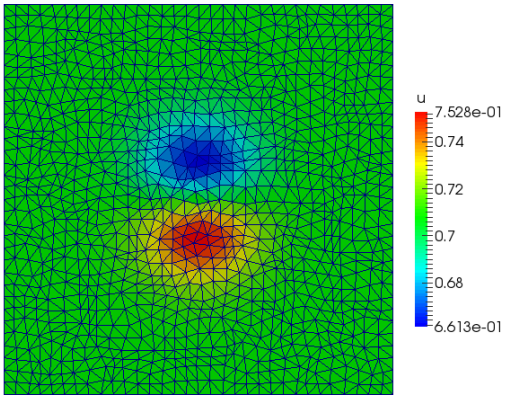


(d) $p = 3$, $N = 16$ on deformed mesh.

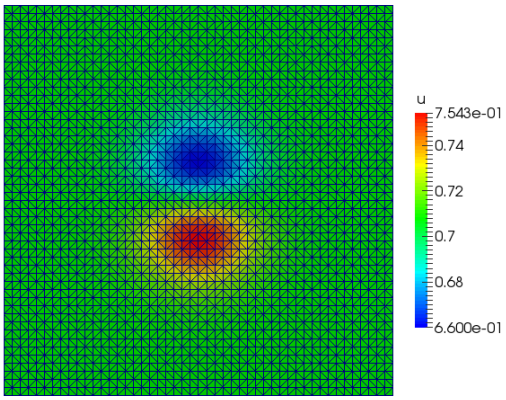
Figure 4.3: Pressure field at $t = 10\sqrt{2}$ with $N = 16$.



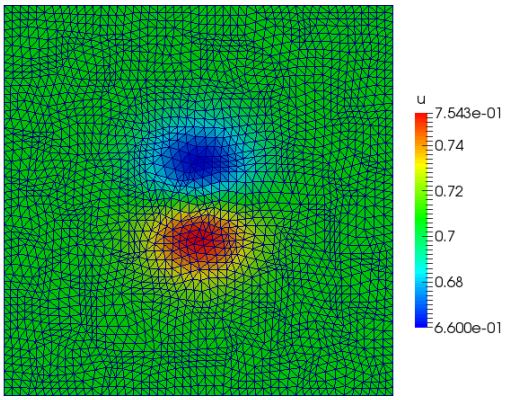
(a) $p = 2$, $N = 16$.



(b) $p = 2$, $N = 16$ on deformed mesh.

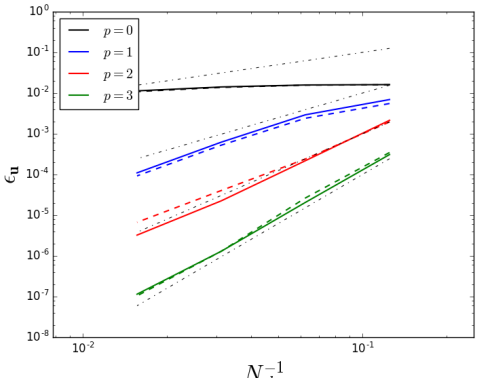


(c) $p = 3$, $N = 16$.

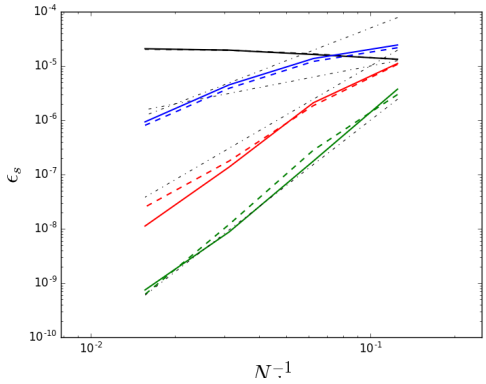


(d) $p = 3$, $N = 16$ on deformed mesh.

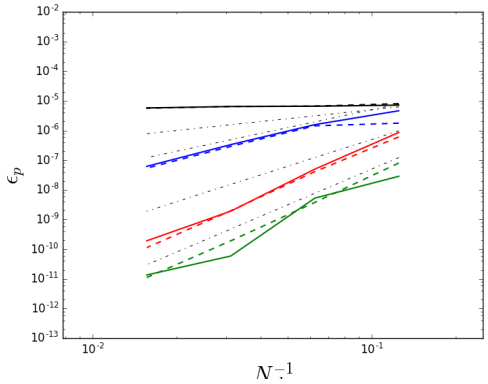
Figure 4.4: Velocity field at $t = 10\sqrt{2}$ with $N = 16$.



(a) Convergence in state variable error.

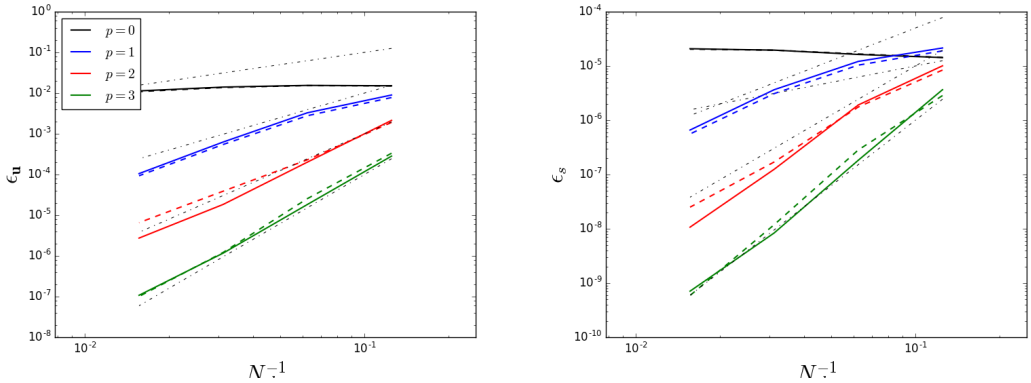


(b) Convergence in entropy error.



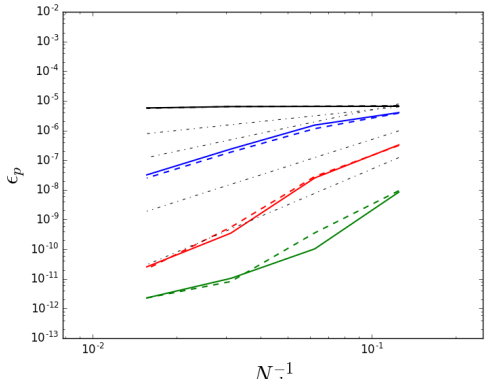
(c) Convergence in pressure error.

Figure 4.5: Convergence of the presented DG solver. The dashed lines color lines are the convergence on the regular mesh, while the sold lines are for the deformed mesh.



(a) Convergence in state variable error.

(b) Convergence in entropy error.



(c) Convergence in pressure error.

Figure 4.6: Convergence of the presented DG solver for the second set of initial conditions. The dashed lines color lines are the convergence on the regular mesh, while the sold lines are for the deformed mesh.

after the interpolation/projection onto the basis. The magnitude of the errors are additionally not much different.

Table 4.1: Convergence rates for the two different ICs.

	Projection	Interpolation
p=0	0.157	0.18
p=1	2.35	2.19
p=2	2.56	2.56
p=3	4.39	4.35

5 Summary

The development of a 2D unstructured DG solver in python was presented. The code makes use of numpy routines, slicing, and broadcasting to efficiently solve the Euler equations in a high level language. The code was tested on an Euler vortex and was shown to converge with roughly $p+1$ order of accuracy for all norms of the solution. The convergence between the projected ICs

and interpolated ICs, however, was not quite as different as I expected. I am thinking that this is because I was computing errors here I am taking the exact solution to be the initial solution after the interpolation/projection onto the basis.