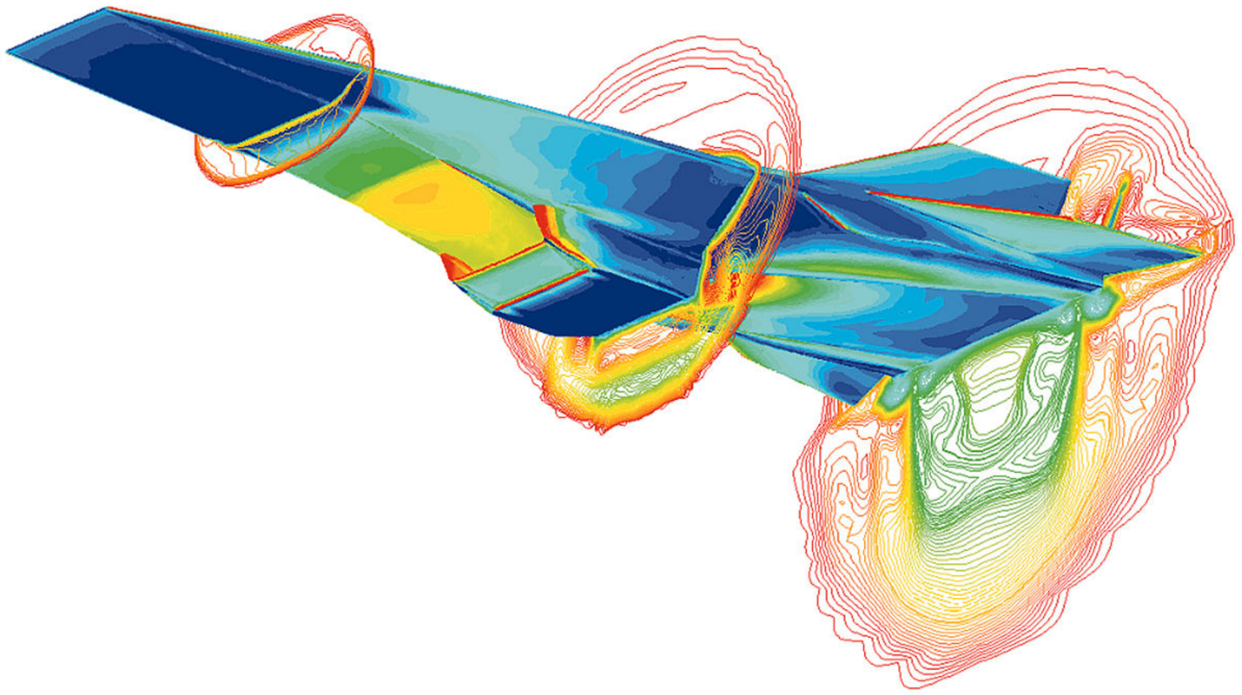
Project 2: Supersonic Engine Analysis

Aerospace 523: Computational Fluid Dynamics I
Graduate Aerospace Engineering
University of Michigan, Ann Arbor

By: Dan Card, dcard@umich.edu
Date: December 4, 2020



NASA X-43 Hypersonic Airplane

COLLEGE OF ENGINEERING
AEROSPACE ENGINEERING
UNIVERSITY OF MICHIGAN

Contents

1 Introduction . . . **4**

2 Numerical Method . . . **6**

3 Adaptation . . . **9**

4 Tasks and Deliverables . . . **10**

4.1 Roe Flux Overview . . . 10

4.1.1 Roe Flux Function . . . 11

4.1.2 Subsonic and Supersonic Implementation Tests . . . 12

4.2 Implementing Finite Volume Method . . . 13

4.2.1 Main Driving Code . . . 13

4.2.2 Finite-Volume-Element Implementation . . . 13

4.2.3 Flux Code Implementation . . . 14

4.2.4 Mesh Adaption Implementation . . . 14

4.2.5 Miscellaneous Code . . . 14

4.3 Convergences and Analysis of Baseline Mesh . . . 15

4.3.1 L_1 Norm Convergence . . . 15

4.3.2 ATPR Output . . . 16

4.3.3 Baseline Field Plots . . . 17

4.4 Implementing Mach Number Jumps . . . 19

4.4.1 Adapted Meshes . . . 19

4.4.2 Adapted Mesh Field Plots . . . 21

4.4.3 Adapted Mesh ATPR Convergence . . . 23

4.5 Adaptive Iterations . . . 24

4.5.1 ATPR Versus Angle of Attack . . . 24

4.5.2 Flow Fields for Varying Angle of Attacks . . . 25

Appendices . . . **27**

Appendix A Python Implementation . . . **27**

A.1 Main Driving Code . . . 27

A.2 Finite-Volume-Element Code . . . 30

A.3 Roe Flux Python Implementation . . . 32

A.4 Adaptive Mesh Python Implementation . . . 33

Appendix B Additional Supporting Code . . . **39**

References . . . **41**

List of Figures

1 Engine Geometry and Boundary Conditions 4
2 Scramjet Baseline Mesh 8
3 Refinement of Triangles Given Edge Splittings 9
4 L_1 Norm Convergence for Baseline Mesh 15
5 ATPR Output for Baseline Mesh 16
6 Field Plot of Mach Number for Baseline Mesh 17
7 Field Plot of Total Pressure for Baseline Mesh 18
8 Adapted Meshes Versus Baseline Mesh 20
9 Field Plot of Mach Number for Adapted Mesh 21
10 Field Plot of Total Pressure for Adapted Mesh 22
11 ATPR Convergence with Cell Number 23
12 ATPR and Angle of Attack 24
13 Mach Field with Varying Angle of Attack 25
14 Total Pressure Field with Varying Angle of Attack 26

List of Equations

1 Freestream State 4
2 Average Total Pressure Recovery 5
3 Cell Average 6
4 Flux Residual 6
5 CFL Definition 6
6 Time Stepping 6
7 Residual Vector 7
8 Forward-Euler Time Stepping Scheme 7
9 Roe Flux 10
10 Roe Flux for Euler Equations 10

List of Tables

1 Roe Flux Consistency Check 12
2 Roe Flux Flipped Direction Check 12
3 Roe Flux Supersonic Normal Velocity 12
4 ATPR Versus Angle of Attack 24

List of Algorithms

1 Main Driving Code 27
2 Finite-Volume-Element Code 30
3 Roe Flux Implementation 32
4 Adaptive Mesh Implementation 33
5 Python Edge Hash 39
6 Python Plot Mesh 40

1 Introduction

In this project you will simulate supersonic flow through a two-dimensional scramjet engine, using a first-order, adaptive, finite-volume method. Combustion will not be included, and your investigation will focus on measuring the total pressure recovery of the engine. The shock structure inside the engine is complex, and accurate simulations will require adapted meshes to resolve the shocks and expansions.

Geometry: Figure 1 shows the geometry of the engine, which consists of two sections: lower and upper. The reference length is the height of the engine channel at the exit, which is d = 1. Note that the units of the measurements are not relevant, as you will be reporting non-dimensional quantities.

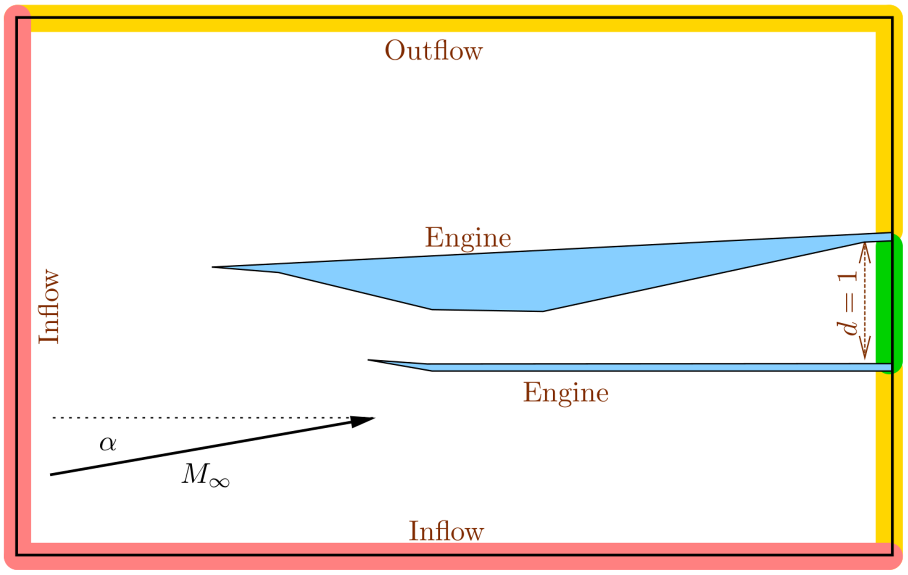


Figure 1: Engine geometry and boundary conditions.

Governing Equations: Use the two-dimensional Euler equations, with a ratio of specific heats of $\gamma = 1.4$.

Units: To avoid ill-conditioning, use "convenient" $\mathcal{O}(1)$ units for this problem, in which the freestream state is

$$\mathbf{u}_\infty = \left[\rho, \quad \rho u, \quad \rho v, \quad \rho E\right]^{\mathbf{T}} = \left[1, \quad M_\infty \cos(\alpha), \quad M_\infty \sin(\alpha), \quad \frac{1}{\gamma(\gamma-1)} + \frac{M_\infty^2}{2}\right]^{\mathbf{T}} \tag{1}$$

where M_∞ is the free-stream Mach number, and α is the angle of attack.

Initial and Boundary Conditions: The computational domain consists of the region around the engine. The inflow portion of the far-field rectangle consists of the left and bottom boundaries. On these boundaries apply free-stream "full-state" conditions, with a free-stream Mach number of $M_\infty = 2.2$. You will investigate angles of attack in the range $\alpha \in [0, 3°]$, with a baseline value of $\alpha = 1°$. On the outflow and engine exit boundaries, assume that the flow is supersonic, which means that no boundary state is needed – the flux is computed from the interior state. On the engine surface, apply the inviscid wall boundary condition.

When initializing the state in a new run, i.e. not when restarting from an existing state, you can set all cells to the same state, based on the free-stream Mach number, M_∞.

Output: Shocks inside the engine are necessary to slow the flow down and compress it for combustion, but they also lead to a loss in total pressure (lost work). A figure of merit is then the *average total pressure recovery* (ATPR), defined by an integral of the engine exit of the ratio of the total pressure to the freestream total pressure,

$$\text{ATPR} = \frac{1}{d}\int_0^d \frac{p_t}{p_{t,\infty}}\,dy, \quad p_t = p\left(1 + \frac{\gamma-1}{2}M^2\right)^{\gamma/(\gamma-1)}, \tag{2}$$

where p is the pressure, p_t is the total pressure, and y measures the vertical distance along the engine exit.

2 Numerical Method

Use the first-order finite volume methods to solve for the flow through the engine. March the solution to steady state using local time stepping, starting from either an initial uniform flow, or from an existing converged or partially-converged state.

Discretization: From the notes, cell i's average, $(\mathbf{u_i})$, evolves in time according to

$$A_i \frac{d\mathbf{u_i}}{dt} + \mathbf{R_i} = \mathbf{0} \rightarrow \frac{d\mathbf{u_i}}{dt} = -\frac{1}{A_i}\mathbf{R_i}. \tag{3}$$

where the flux residual $\mathbf{R_i}$ for a triangular cell is

$$\mathbf{R_i} = \sum_{e=1}^{3} \hat{\mathbf{F}}(\mathbf{u_i}, \mathbf{u_{N(i,e)}}, \vec{n}_{i,e})\Delta l_{i,e} \tag{4}$$

Recall that $N(i,e)$ is the cell adjacent to cell i across edge e, and $\vec{n}_{i,e}, \Delta l_{i,e}$ are the outward normal and length on edge e of cell i. Discretize Equation 3 with forward Euler time integration and use local time stepping to drive the solution to steady state.

Local Time Stepping: To implement local time stepping, a vector of time steps is calculated, one time step for each cell: Δt_i. Defining the CFL number for cell i as,

$$\text{CFL}_i = \frac{\Delta t_i}{2A_i}\sum_{e=1}^{3}|s|_{i,e}\Delta l_{i,e}, \tag{5}$$

where A_i is the area of the cell, the summation is over the three edges of a cell, and $|s|_{i,e}$ is the maximum propagation speed for edge e.

Time stepping requires the value of $\Delta t_i / A_i$ for each cell, and this can be calculated by re-arranging Equation 5,

$$\frac{\Delta t_i}{A_i} = \frac{2\text{CFL}_i}{\sum_{e=1}^{3}|s|_{i,e}\Delta l_{i.e}}. \tag{6}$$

The easiest method to calculate the right-hand-side is to calculate the summation of $|s|_e \Delta l_{i,e}$ during the flux evaluations. Note that the propagation speed $|s|_{i,e}$ should be calculated by the flux function. In local time stepping, the CFL number for each cell is the same: $\text{CFL}_i = \text{CFL} = 1.0$ is a good choice for this project.

Residuals and Convergences: Assess convergence by monitoring the undivided L_1 norm of the residual vector, defined as

$$|\mathbf{R}|_{L_1} = \sum_{\text{cells } i} \sum_{\text{states } k} |R_{i,k}| \tag{7}$$

That is, take the sum of the absolute values of all of the entries in your residual vector (you will be summing the 4 conservation equation residuals in each cell). You should not divide by the number of entries/cells, as the residuals already represent integrated quantities over the cells, so the sum will behave properly with mesh refinement. Deem a solution converged when $|\mathbf{R}|_{L_1} < 10^{-5}$.

Numerical Flux: Use the Roe flux for the interface flux and to impose the full-state far-field boundary condition. This flux is described in the course notes. You will need to verify your flux once implemented, using the following tests:

- Consistency check: $\mathbf{F}(\mathbf{u_L}, \mathbf{u_L}, \vec{n})$ should be the same as $\tilde{\mathbf{F}}(\mathbf{U_L}) \cdot \vec{n}$ (the analytical flux dotted with the normal).
- Flipping the direction: check that $\mathbf{F}(\mathbf{u_L}, \mathbf{u_R}, \vec{n}) = -\mathbf{F}(\mathbf{u_R}, \mathbf{u_L}, -\vec{n})$.
- States with supersonic normal velocity: the flux function should return the analytical flux from the upwind state. The downwind state should not have any effect on flux.

Time Stepping: Use the forward-Euler method to drive the solution to steady state. With local time-stepping, the update on cell i at iteration n can be written as

$$\mathbf{u}_i^{n+1} = \mathbf{u}_i^n - \frac{\Delta t_i^n}{A_i} \mathbf{R_i}(\mathbf{U^n}), \tag{8}$$

where Δt_i^n is the local time step computed from the state at time step n.

Mesh: You are provided with a baseline mesh of 1670 cells, shown in Figure 2. This mesh will not provide very accurate flow solutions, but it will serve as the starting point for adaptation. The included `readme.txt` file describes the structure of the text-based `.gri` mesh file. You are also given python and Matlab codes for reading the `.gri` mesh file and for plotting/processing the mesh.

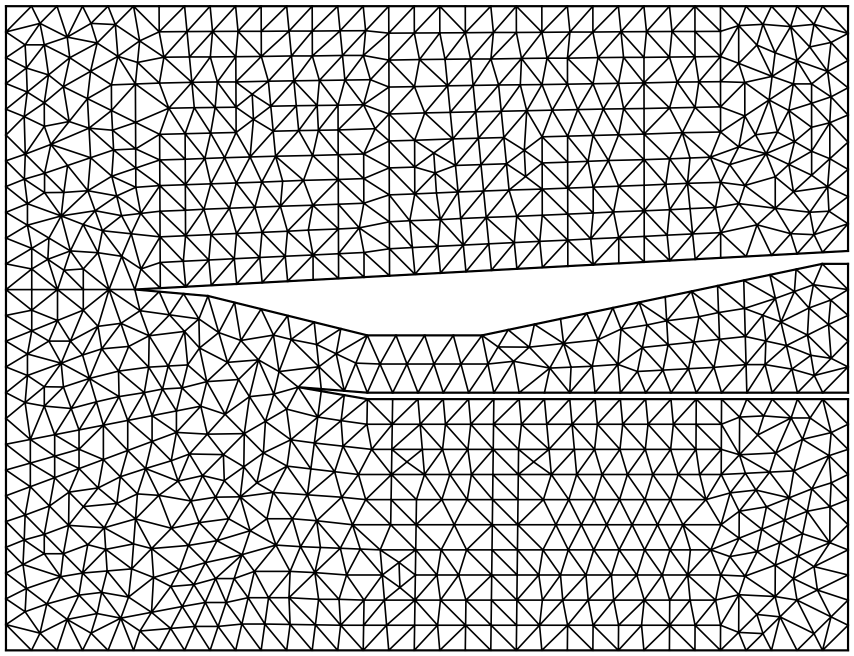


Figure 2: Scramjet baseline mesh.

Output Calculation: The average total pressure recovery output in Equation 2 requires an integral over the engine exit. Approximate this integral by summing over the edges on the exit boundary. For each edge, use the state from the adjacent cell to calculate the total pressure.

3 Adaptation

You will use mesh adaptation to improve solution quality. Adapting a mesh means locally increasing the mesh resolution in regions where errors are likely to be large. This requires a measurement of error and a method for adapting the mesh. A reasonable way to measure error is to look at jumps in the solution between cells. For example, looking at jumps in the Mach number, we can define an error indicator for each interior edge e according to

$$\text{interior: } \epsilon_e = |M_{k^+} - M_{k^-}| h_e.$$

In this formula, M_{k^+} and M_{k^-} are the Mach numbers on the two cells adjacent to edge e, and h_e is the length of edge e.

You can assume that the error indicator on the farfield boundary edges is zero. On the engine boundary (solid wall), define the error indicator by

$$\text{wall: } \epsilon_e = |M_k^{\perp}| h_e,$$

where $M_k^{\perp}$ is the Mach number of the cell's velocity component in the edge normal direction.

After calculating the error indicators ϵ_e over all edges (interior and boundary), sort the indicators in decreasing order and flag a small fraction $f = .03$ of edges with the highest error for refinement. Next, to smooth out the refinement pattern, loop over all cells: if a cell has *any* of its edges flagged for refinement, then flag *all* of its edges for refinement. This will increase the total number of edges for refinement.

Once edges are flagged as described, refine all cells adjacent to flagged edges. These cells will fall into one of three categories, shown in Figure 3, and they should be refined as indicated. At each adaptive iteration, transfer the solution to the new mesh to provide a good initial guess for the next solve.

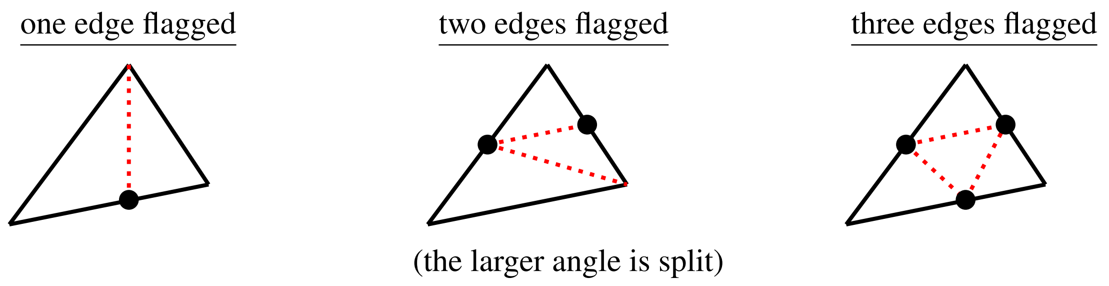


Figure 3: Refinement of triangles given edge splittings.

4 Tasks and Deliverables

In preparation for simulating the scramjet engine inlet performance I will prepare code that will implement Roe Flux to approximate the changing flow state between cells. After verification that the flux is correctly implemented then I will implement a first-order finite volume method to approximate the steady state solution and perform a convergence study on my method. Additionally, I will model Mach number jumps throughout the domain and determine the the averaged total pressure recovery. Finally, I will perform adaptive iterations to then determine the effects of the angle of attack and the averaged total pressure recovery.

4.1 Roe Flux Overview

Roe flux, is an alternative flux that carefully upwinds waves one by one and is given by Equation 9 below. [1]

$$\hat{\mathbf{F}} = \frac{1}{2}\left(\mathbf{F_L} + \mathbf{F_R}\right) - \frac{1}{2}\left|\frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^*)\right|\left(\mathbf{u_R} - \mathbf{u_L}\right) \tag{9}$$

In this expression $\left|\frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^*)\right|$ refers to the absolute values of the eigenvalues, i.e. $\mathbf{R}|\mathbf{\Lambda}|\mathbf{L}$, in the eigenvalue decomposition. $\mathbf{u}^*$ is an intermediate state that is based on $\mathbf{u_L}$ and $\mathbf{u_R}$. This intermediate choice is important for nonlinear problems, and the Roe flux uses the Roe-average state, a choice that yields exact single-wave solutions to the Riemann problem. However, for Euler equations Roe flux is given by Equation 10 below.

$$\hat{\mathbf{F}} = \frac{1}{2}(\mathbf{F_L} + \mathbf{F_R}) - \frac{1}{2}\begin{bmatrix} |\lambda|_3\Delta\rho + C_1 \\ |\lambda|_3\Delta(\rho\vec{v}) + C_1\vec{v} + C_2\hat{n} \\ |\lambda|_3\Delta(\rho E) + C_1 H + C_2(\vec{v}\cdot\hat{n}) \end{bmatrix} \tag{10}$$

Where further expansions of the constants above give,

$$\begin{bmatrix}\lambda_1, & \lambda_2, & \lambda_3, & \lambda_4\end{bmatrix} = \begin{bmatrix}u+c, & u-c, & u, & u\end{bmatrix}$$

$$\vec{v} = \frac{\sqrt{\rho_L}\vec{v}_L + \sqrt{\rho_R}\vec{v}_R}{\sqrt{\rho_L}+\sqrt{\rho_R}}, \qquad H = \frac{\sqrt{\rho_L}H_L + \sqrt{\rho_R}H_R}{\sqrt{\rho_L}+\sqrt{\rho_R}}$$

$$C_1 = \frac{G_1}{c^2}(s_1 - |\lambda|_3) + \frac{G_2}{c}s_2, \qquad C_2 = \frac{G_1}{c}s_2 + (s_1 - |\lambda|_3)G_2$$

$$G_1 = (\gamma-1)\left(\frac{q^2}{2}\Delta\rho - \vec{v}\cdot\Delta(\rho\vec{v}) + \Delta(\rho E)\right), \qquad G_2 = -(\vec{v}\cdot\hat{n})\Delta\rho + \Delta(\rho\vec{v})\cdot\hat{n}$$

$$s_1 = \frac{1}{2}\left(|\lambda|_1 + |\lambda|_2\right), \qquad s_2 = \frac{1}{2}(|\lambda|_1 - |\lambda|_2)$$

Where the difference in states is given by,

$$\Delta \mathbf{u} = \mathbf{u_R} - \mathbf{u_L}, \qquad q^2 = u^2 + v^2$$
$$\mathbf{F_L} = \tilde{\mathbf{F}}(\mathbf{u_L}) \cdot \hat{n}, \qquad \mathbf{F_R} = \tilde{\mathbf{F}}(\mathbf{u_R}) \cdot \hat{n}$$

However, to prevent expantion shocks, an entropy fix is required. The simple solution to this is to keep all eigenvalues away from zero such that,

$$\text{if } |\lambda|_i < \epsilon \text{ then } \lambda_i = \frac{\epsilon^2 + \lambda_i^2}{2\epsilon}, \quad \forall\ i\ \in\ [1,\ 4]$$

Where ϵ is a small fraction of the Roe-averaged speed of sound, e.g. $\epsilon = 0.1c$

4.1.1 Roe Flux Function

In this project I will implement Roe Flux into Python3 that will be further implemented when writing the finite-volume method to determine the flow through the scramjet. Essentially this function is as follows:

Inputs This function inputs the left state and the right state of a given edge. This will allow the finite-volume method solver to simply call this function when determining the fluxes in and out of a given cell. Furthermore, this function will also input the normal vector that will determine the flux in a given direction.

Generating Arguments Going further, this code then will determine the states of the left and right side such as $\rho,\ u,\ v,\ P,\ H$ to determine the flux and approximate the Roe-average state. With the left and right hand fluxes determined what's left is the Roe-averages.

Roe-Average Determining the Roe-average is done by passing all the calculated values into a separate subfunction that will determine the Roe-averages from a weighted averaged of the densities to the state properties. Additionally in this function it will calculate the wave propagating eigenvalues to remove discontinuities from the calculation.

Final Calculation Then with the Roe-Average and the fluxes determined, simply conducted the average of the fluxes subtracted by half the sum of the running waves.[2]

4.1.2 Subsonic and Supersonic Implementation Tests

Consistency Check: First and foremost is a simple check to see if the Roe flux at steady state is equal to the flux of a single state vector acting in the same direction of the normal. In this I simply returned the values in Python3 and tabulated the results in order to check the consistency. In this test I assumed $\alpha = 0°,\ M_\infty = 0.8,\ \vec{n} = \begin{bmatrix}1, & 0\end{bmatrix}$ and used this initial state for u_l. Performing the consistency check I get Table 1 below aligning with theory.

Table 1: Roe Flux consistency check.

Flux	ρ	ρu	ρv	ρE
$\hat{F}(u_l, u_l, \vec{n})$	0.800	1.354	0.000	2.256
$\vec{F}(\vec{U}_l) \cdot \vec{n}$	0.800	1.354	0.000	2.256
ΔF	0.00e+00	0.00e+00	0.00e+00	0.00e+00

Direction Flipping Next is to check that there is agreement with flipping the states and the norm vector and returning the same results without error. In this test case I will assume that the left state will be $\alpha = 0°,\ M_\infty = 2.2,\ \vec{n} = \begin{bmatrix}1, & 0\end{bmatrix}$ initially and for the right state the same but with $M_\infty = 2.4$ initially. Tabulating the results gives Table 2 below.

Table 2: Roe Flux flipped direction check.

Flux	ρ	ρu	ρv	ρE
$\hat{F}(u_l, u_r, \vec{n})$	0.800	1.354	0.000	2.256
$-F(u_r, u_l, -\vec{n})$	0.800	1.354	-0.000	2.256
ΔF	0.00e+00	0.00e+00	0.00e+00	0.00e+00

Supersonic Normal Velocity Conducting the supersonic normal velocity test for with Roe Flux is a test shown below in Table 3. In this test I compare $\hat{F}$ to $F_l,\ F_R$ and determine any discrepancies. This function returns the analytical flux from the upwind state and the downwind state does not have any effect on the flux. In this case, I assumed that the upwind had a free-stream $M_\infty = 2.2$ and a down-stream $M_\infty = 2.5$.

Table 3: Roe Flux supersonic normal velocity.

Flux	ρ	ρu	ρv	ρE
$\hat{F}(u_l, u_r, \vec{n})$	1.556	3.927	0.505	7.654
F_L	1.556	3.927	0.505	7.654
F_R	1.768	4.924	0.505	9.944

4.2 Implementing Finite Volume Method

The structure of my code will have several key parts. First and foremost in my code is the driving code which will call into the functions that will solve and approximate the steady-state solution. There are 4 main code implementations, one that calls the appropriate solver code, the finite-volume-element code, the Roe-Flux code, then the mesh adaption code. Other additional codes will be discussed but from a low-level perspective.

4.2.1 Main Driving Code

Firstly, the main driving code is responsible for generating the plots and tables discussed in this report. This code is responsible for testing the Roe-Flux cases from the prior section and outputting the results in a table format in this report. Furthermore, this main code will call the solving code and will generate the steady-state solution and generate figures of the field plots in the upcoming sections.

4.2.2 Finite-Volume-Element Implementation

This code section will input a given mesh, process `V`, `E`, `BE`, `IE` and generate the initial free-stream state $\mathbf{u}_\infty$ that will start the initial approximation of the steady-state. This code will start with a `while` loop iterating until the solution's residuals are less than the specified project tolerance.

Within this loop, the code will run through the interior edges(`IE`) and will determine the fluxes from the normal and then add/subtract these fluxes and lengths into the corresponding residual for the specified element and neighboring element. Furthermore, the same will be applied for the wave-speed and lengths being added for the appropriate element and neighboring element.

After the interior elements have been looped over, next will be to loop over the exterior elements (`BE`) and impose boundary conditions that will generate a physical solution. Then in this loop the code will determine which group the given edge is in and then impose the corresponding boundary condition. These boundary conditions will be free-stream – where the exterior is equal to the initial condition, outflow – where the exterior is equal to the interior state, or inviscid where it is assumed that no density or energy is transferred but momentum can still flux.

4.2.3 Flux Code Implementation

As discussed in Section 4.1.1, the Roe flux will input a “left” state and a “right” state following a normal vector to determine the flux. It will determine the state values used for Euler’s equations and then determine the Roe-Averages then finally determine the flux and return the approximation. This approximation will be used to determine the residuals in the approximation of the steady-state.

4.2.4 Mesh Adaption Implementation

In order to increase the accuracy of the approximated solution I will write a function `adapt` that will determine the error between cells from the Mach number and increase the resolution in the cells that make up the top 3% of the errors for a given converged solution. In this code it will flag the cells and keep track of how many times a given cell has been flagged for a large discrepancy in the Mach number and then will refine the mesh.

After determining which meshes will be refined, the code will re-arrange the boundary and vertices to “adapt” the mesh to include add later . . .

4.2.5 Miscellaneous Code

Initial Condition This supporting function will determine the initial condition depending on the angle of attack α, and return the initial state for the solver code with the free-stream condition specified in Equation 1.

ATPR Calculation This additional code will input the state at each iteration in the solver code and will determine the ATPR from a numerical integration along the exit of the engine. In this code, it will determine the free-stream total pressure as well as the total pressure in a given cell and then sum the total value of ATPR that will be solved for for each iteration.

4.3 Convergences and Analysis of Baseline Mesh

After implementing my finite-volume code I will perform several convergence studies and look to the results of my solver to determine the accuracy of my implementation. In this section I will perform an L_1 residual norm, look at the accuracy of the solver from the results of the ATPR over time iterations, and then finally analyze the total pressure field, as well as the Mach field.

4.3.1 L_1 Norm Convergence

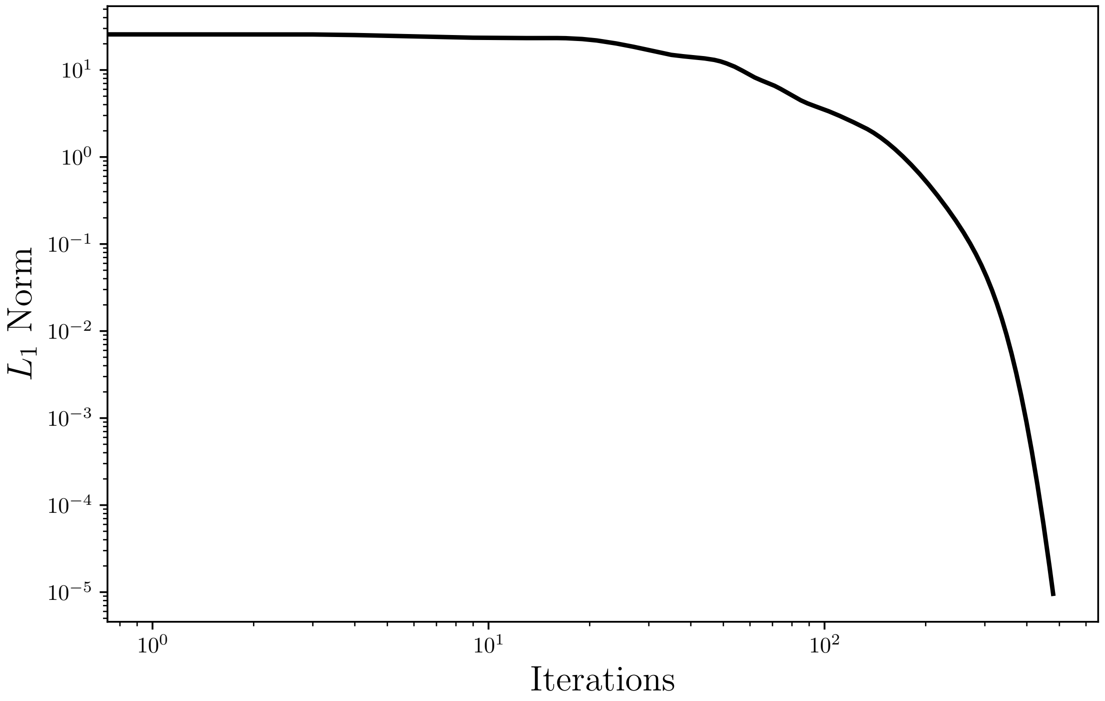


Figure 4: L_1 norm convergence versus time step iterations.

Shown above in Figure 4, is the convergence of L_1 norm as my code progresses through time-step iterations. As shown, and verified above this method will converge to an approximate answer in which the L_1 error is less than 10^{-5} to deem an accurate answer. The convergence rate is not given, since this method is conducting local-time step iterations which would not return a physical answer.

4.3.2 ATPR Output

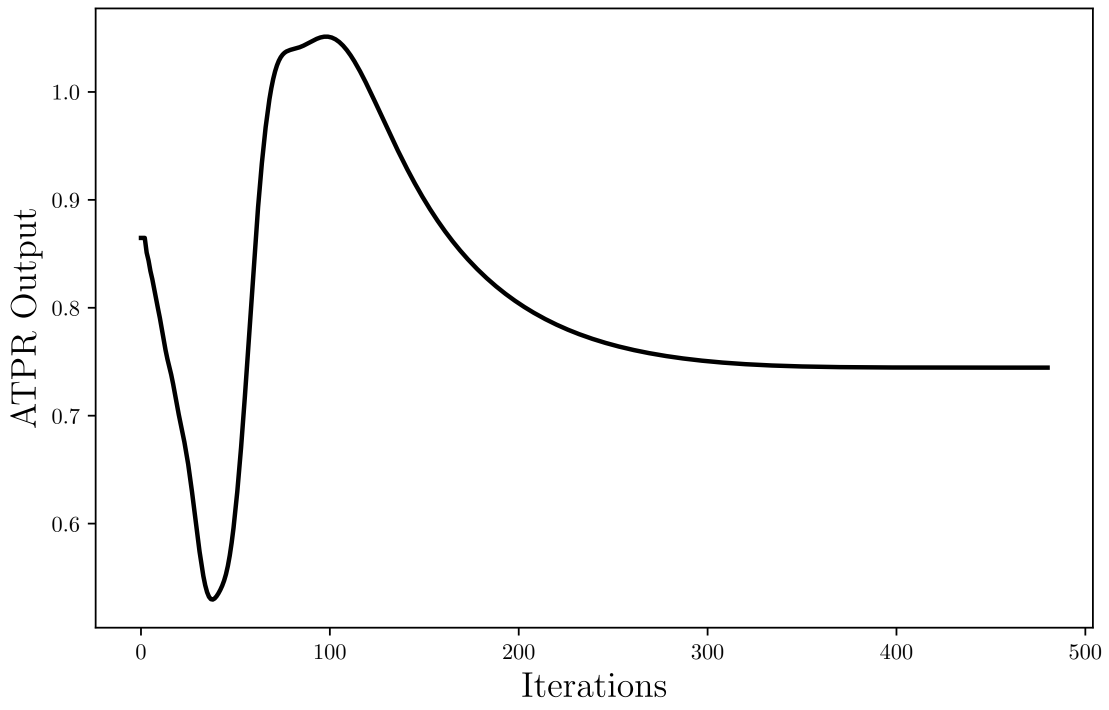


Figure 5: ATPR output for baseline mesh.

Next, was to check and confirm that the solution is giving a physical answer returning an ATPR that is less than one at the exit of the engine. The reason for the less than one is due to the fact that shocks are forming at the inlet of the engine resulting in a loss of total pressure due to entropy that cannot be recovered. Using Equation 2, with the approximated state values I get Figure 5 above confirming that the solution is converging to a value that physically makes sense.

4.3.3 Baseline Field Plots

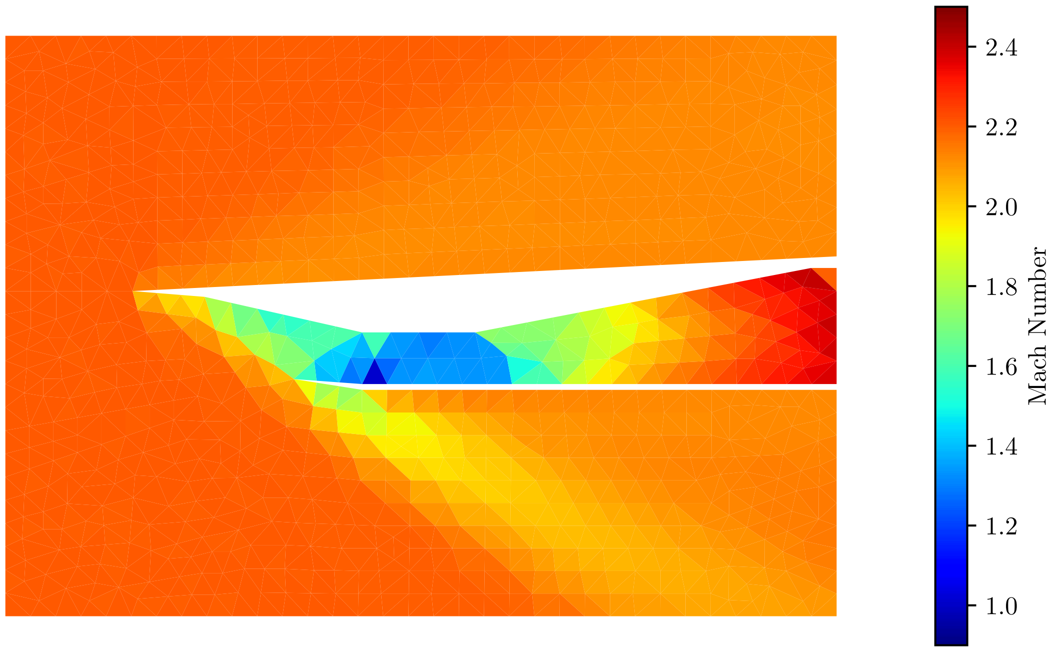


Figure 6: Field plot of Mach number with $\alpha = 1°$.

Field Plot of Mach Number Above in Figure 6, is the field plot of the mach number at $M_\infty = 2.2$ at an angle of $\alpha = 1°$. This plot shows the free-stream mach number at the steady-state with visible oblique shocks at the inlet of the engine. However, due to the coarseness of the mesh, much information is lost within the interior of the engine resulting from the train of shocks inside the inlet of the engine. The next section aims to refine this mesh to return a more refined result.

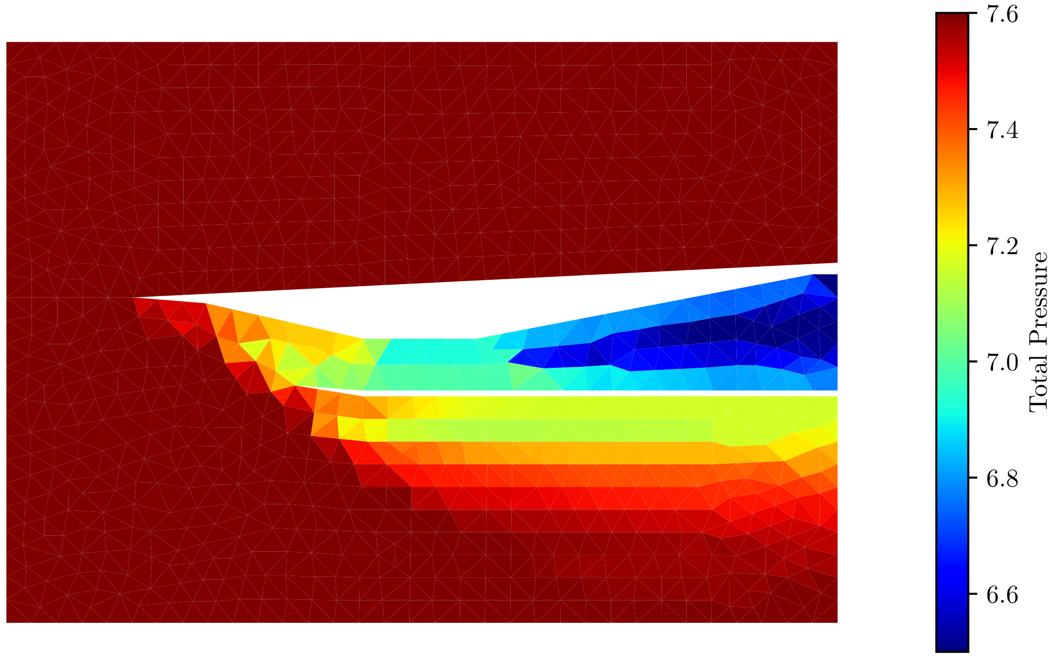


Figure 7: Field plot of total pressure with $\alpha = 1^\circ$.

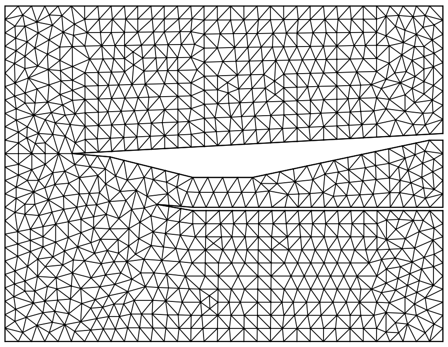
Field Plot of Total Pressure Above in Figure 7, is the field plot of the total pressure at M_∞ at an angle of $\alpha = 1^\circ$. Similar to Figure 6, there are some visible oblique shocks at the inlet of the engine. But similar to the mach field plot, much of the information is lost within the interior of the engine requiring more refinement of the mesh to return a more accurate solution. What can be found is that the total pressure decreases throughout the inlet of the engine which is consistent with theory through the losses associated with the shocks.

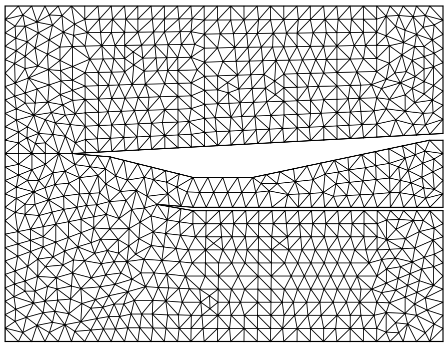
4.4 Implementing Mach Number Jumps

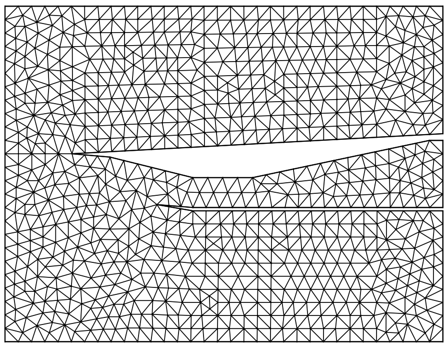
In this section I will implement an adaptive mesh function that will flag edges shown in Figure 3 given the discrepancy in Mach number across cell edges. The purpose of this function is to refine the mesh in the areas that are more prone to error; most notably the areas where there is large jumps in the Mach number like across shocks. These shocks will be located at the inlet and then trained throughout the interior of the engine. In this section I will refine the mesh and then look at the results of the Mach field, the total pressure, and finally the ATPR at the end of each refinement iteration.

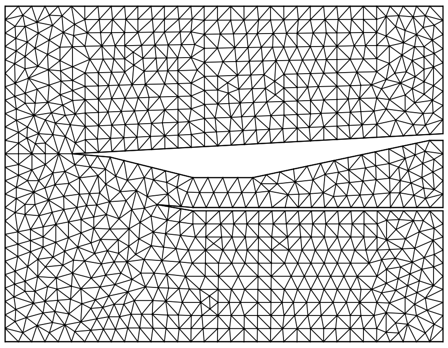
4.4.1 Adapted Meshes

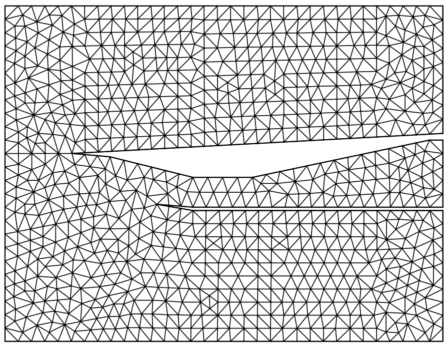
In this section, I will implement the Mach jumps into the code to refine the meshes to lower the error in the approximated solution. Using my code I will create a new mesh after each iterative solution and then refine the mesh that will be used on the next approximation. Shown on the following page in Figure 8, are the meshes after each refinement. Most notable, is that the mesh refines at the location at which oblique shocks are forming – the interior and inlet of the engine. This refinement makes sense intuitively since shocks provide a discontinuity in the flow which naturally cause large errors in calculation.


(a) Baseline mesh.


(b) Adapted mesh, iteration 1.


(c) Adapted mesh, iteration 2.


(d) Adapted mesh, iteration 3.


(e) Adapted mesh, iteration 4.

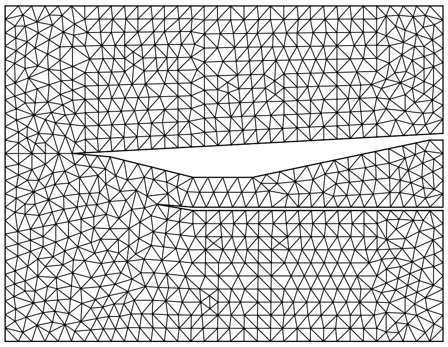

(f) Adapted mesh, iteration 5.

Figure 8: Adapted meshes versus baseline mesh.

4.4.2 Adapted Mesh Field Plots

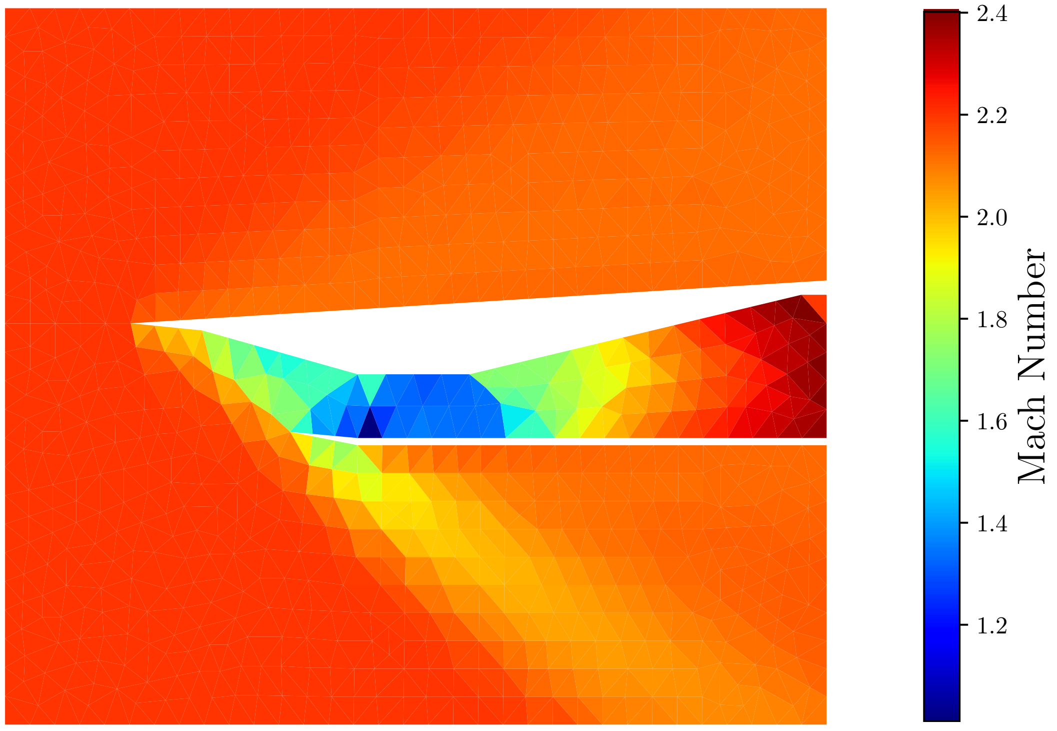


Figure 9: Field plot of Mach number with $\alpha = 1^\circ$ for the finest mesh.

Finest Mesh Field Plot of Mach Number Shown above in Figure 9, is the Mach field for the most refined mesh after 5 adaptive iterations. Comparing the results from this refined mesh to that in Figure 6 shows what . . .

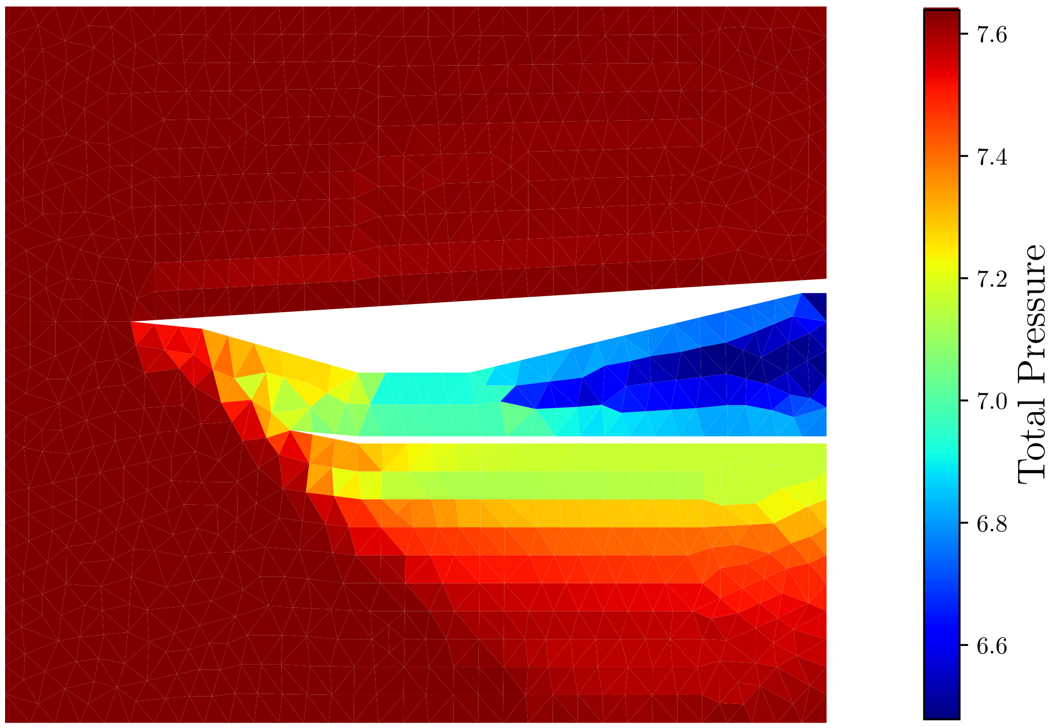


Figure 10: Field plot of total pressure with $\alpha = 1^\circ$ for the finest mesh.

Finest Mesh Field Plot of Total Pressure Shown above in Figure 10, is the total pressure field for the most refined mesh after 5 adaptive iterations. Comparing the results from this refined mesh to that in Figure 7 shows what ...

4.4.3 Adapted Mesh ATPR Convergence

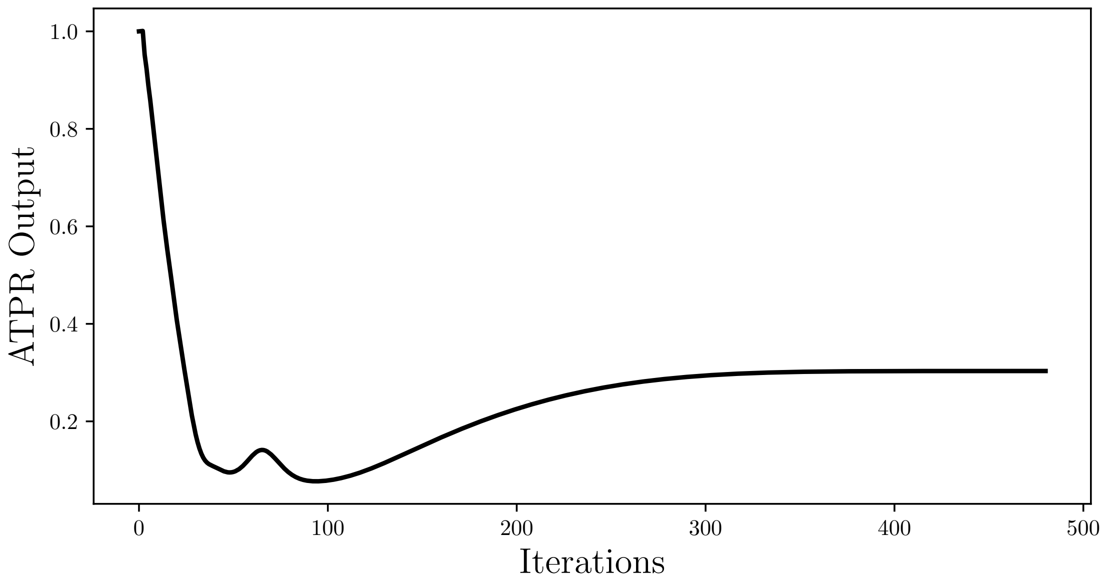


Figure 11: ATPR output versus number of cells in mesh.

Shown above in Figure 11, is the ATPR output versus the number of cells and its effect on the convergence of the ATPR. Discuss . . .

4.5 Adaptive Iterations

In the final section, I will vary the angle of attack as well as performing adaptive mesh refinements to determine the effects of α on the Mach field plot, the total pressure field plot, and the ATPR output.

4.5.1 ATPR Versus Angle of Attack

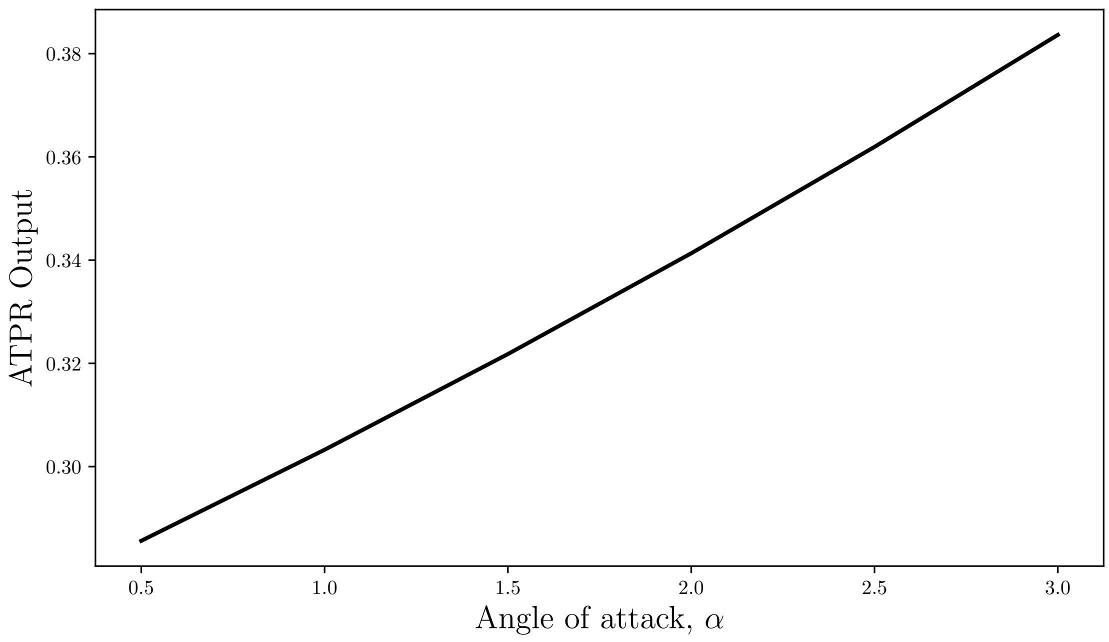


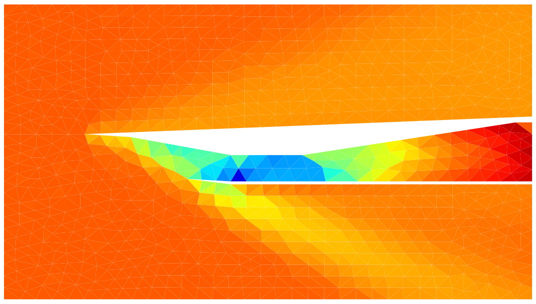
Figure 12: Effects of varying α on the ATPR output.

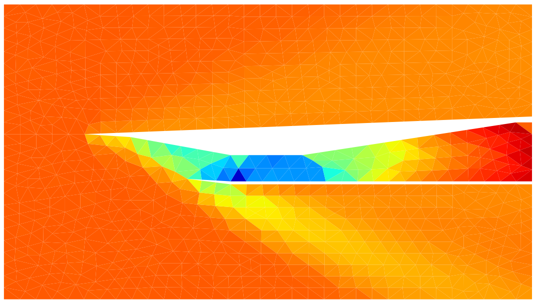
Shown above in Figure 12, is the effect of varying the angle of attack on ATPR output. Discuss . . . . Looking to Table 4, below for a Python print out of the values shown above for more accuracy.

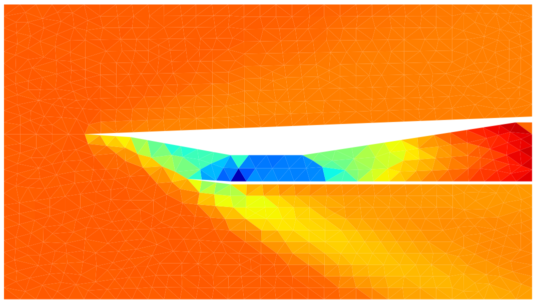
Table 4: ATPR versus angle of attack α.

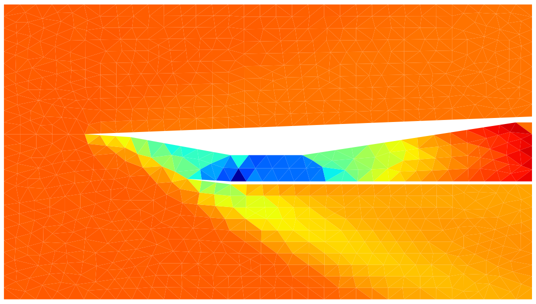
α	ATPR
0.5°	0.286
1.0°	0.303
1.5°	0.322
2.0°	0.341
2.5°	0.362
3.0°	0.384

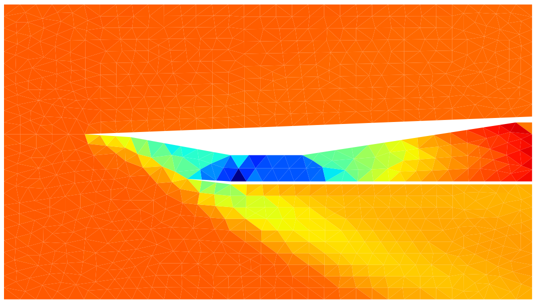
4.5.2 Flow Fields for Varying Angle of Attacks


(a) Mach field at $\alpha = 0.5°$.


(b) Mach field at $\alpha = 1.0°$.


(c) Mach field at $\alpha = 1.5°$.


(d) Mach field at $\alpha = 2.0°$.


(e) Mach field at $\alpha = 2.5°$.

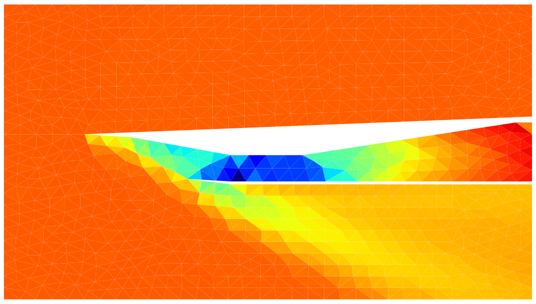
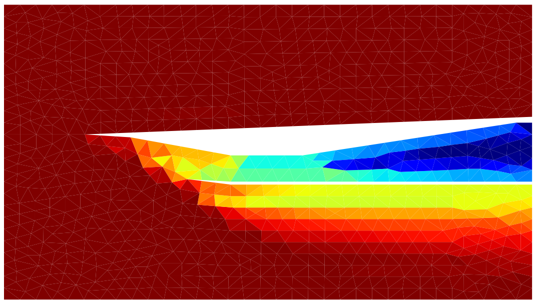
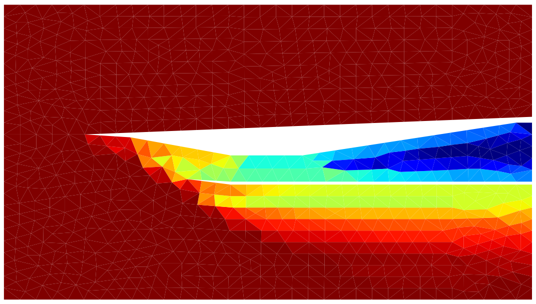
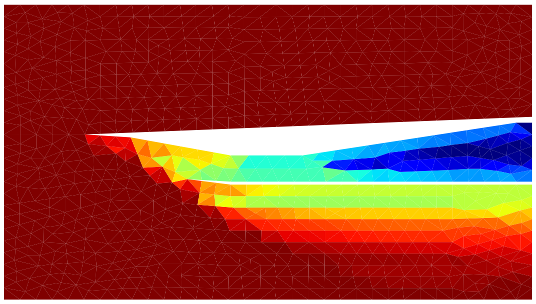

(f) Mach field at $\alpha = 3.0°$.

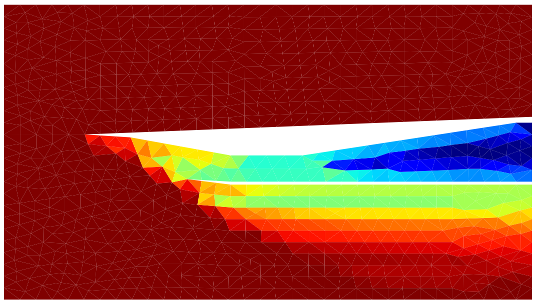
Figure 13: Varying angle of attack, and its effect on the mach field.

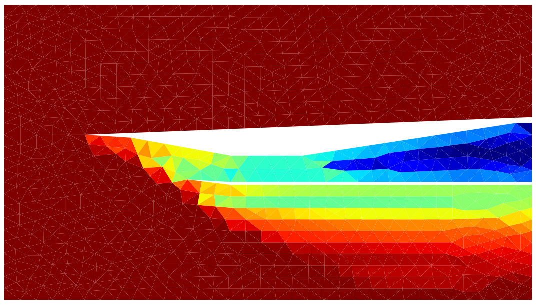
Effect on Mach Field from Varying Angle of Attack Shown above in Figure 13 are the Mach fields for varying angles of attack. Discussion . . .


(a) Total pressure field at $\alpha = 0.5°$.


(b) Total pressure field at $\alpha = 1.0°$.


(c) Total pressure field at $\alpha = 1.5°$.


(d) Total pressure field at $\alpha = 2.0°$.


(e) Total pressure field at $\alpha = 2.5°$.

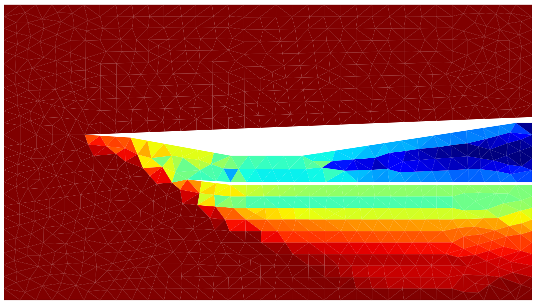

(f) Total pressure field at $\alpha = 3.0°$.

Figure 14: Varying angle of attack, and its effect on the total pressure field.

Effect on Total Pressure Field from Varying Angle of Attack Shown above in Figure 14 are the total pressure fields for varying angles of attack. Discussion . . .

Appendices

A Python Implementation

A.1 Main Driving Code

Algorithm 1: Main Driving Code

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
import time

# Project specific functions
from readgri import readgri
from plotmesh import plotmesh
from flux import RoeFlux
from fvm import solve
from adapt import adapt

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

def getIC(alpha, mach):
    gam = 1.4
    alpha = np.deg2rad(alpha)
    uinf = np.transpose(np.array([1, mach*np.cos(alpha), mach*np.sin(alpha), 1/(gam*(gam-1)) + mach**2/2]))

    return uinf

def test_flux():
    alpha = 0
    ul = getIC(alpha, 0.8); ur = getIC(alpha, 0.8)
    n = np.array([np.cos(np.deg2rad(alpha)),np.sin(np.deg2rad(alpha))])

    # Consistency Check
    F, analytical, FR, ls = RoeFlux(ul, ul, n); diff = abs(F - analytical)
    print('Roe␣Flux␣Tests:\nConsistency␣Check\n' + 50*'-' + '\n', F,'\n', analytical)

    f = open('q1/consistency', 'w')
    f.write(r'Flux␣&␣$\rho$␣&␣$\rho␣u$␣&␣$\rho␣v$␣&␣$\rho␣E$␣\\␣\hline\hline')
    f.write(r'$\hat{F}(u_l,u_l,\vec{n})$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f␣\\'%(F[0],F[1],F[2],F[3]))
    f.write(r'$\vec{F}(\vec{U}_l)\cdot␣\vec{n}␣$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f␣\\'%(analytical[0],analytical[1],analytical[2],analytical[3]))
    f.write(r'$\Delta␣F$␣&␣%.2e␣&␣%.2e␣&␣%.2e␣&␣%.2e'%(diff[0],diff[1],diff[2],diff[3]))
    f.close()

    # Flipping with Direction
    Fl, FL, FR, ls = RoeFlux(ul,ur, n); Fr, FL, FR, ls = RoeFlux(ur,ul, -n); Fr *= -1; diff = abs(Fl-Fr)
    print('\n\nFlipping␣Direction\n' + 50*'-' + '\n', Fl,'\n', Fr)

    f = open('q1/flipped', 'w')
    f.write(r'Flux␣&␣$\rho$␣&␣$\rho␣u$␣&␣$\rho␣v$␣&␣$\rho␣E$␣\\␣\hline\hline')
    f.write(r'$\hat{F}(u_l,u_r,\vec{n})$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f␣\\'%(Fl[0],Fl[1],Fl[2],Fl[3]))
    f.write(r'$-F(u_r,u_l,-\vec{n})$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f␣\\'%(Fr[0],Fr[1],Fr[2],Fr[3]))
    f.write(r'$\Delta␣F$␣&␣%.2e␣&␣%.2e␣&␣%.2e␣&␣%.2e'%(diff[0],diff[1],diff[2],diff[3]))
    f.close()

    # Free-stream
    Fl, FL, FR, ls = RoeFlux(ul,ul, n); Fr, FL, FR, ls = RoeFlux(ur,ur, n); diff = abs(Fl-Fr)
    print('\n\nFree␣Stream␣Test\n' + 50*'-' + '\n', Fl,'\n', Fr)

    # Supersonic Normal Velocity
    alpha = 0
    ul = getIC(alpha, 2.2); ur = getIC(alpha, 2.5)
    F, FL, FR, ls = RoeFlux(ul,ur, np.array([np.sqrt(2)/2,np.sqrt(2)/2]))
    print('\n\nSupersonic␣Normal␣Velocity\n'+50*'-'+'\n', F,'\n', FL,'\n', FR)

    f = open('q1/supersonic_normal', 'w')
    f.write(r'Flux␣&␣$\rho$␣&␣$\rho␣u$␣&␣$\rho␣v$␣&␣$\rho␣E$␣\\␣\hline\hline')
    f.write(r'$\hat{F}(u_l,u_r,\vec{n})$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f␣\\'%(F[0],F[1],F[2],F[3]))
    f.write(r'$F_L$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f␣\\'%(FL[0],FL[1],FL[2],FL[3]))
    f.write(r'$F_R$␣&␣%.3f␣&␣%.3f␣&␣%.3f␣&␣%.3f'%(FR[0],FR[1],FR[2],FR[3]))
    f.close()

def post_process(u):
    gam = 1.4
    uvel = u[:,1]/u[:,0]; vvel = u[:,2]/u[:,0]

    q = np.sqrt(uvel**2 + vvel**2)
    p = (gam-1)*(u[:,3]-0.5*u[:,0]*q**2)
    H = (u[:,3] + p)/u[:,0]

    c = np.sqrt((gam-1.0)*(H - 0.5*q**2))
    mach = q/c
    pt = p*(1 + 0.5*0.4*mach**2)**(gam/(gam-1))

    return mach, pt

def run_fvm():
    mesh = readgri('mesh0.gri')

    start = time.time()
    u, err, ATPR, V, E, BE, IE = solve(1,mesh); end = time.time(); print('Elapsed␣Time␣%.2f'%(end - start))
    mach, pt = post_process(u)


    plt.figure(figsize=(8,5))
    plt.plot(np.arange(err.shape[0]), err, lw=2, color='k')
    plt.xlabel(r'Iterations', fontsize=16)
    plt.ylabel(r'$L_1$␣Norm', fontsize=16)
    plt.xscale('log'); plt.yscale('log')
    plt.savefig('q3/l1_err.pdf', bbox_inches='tight')
    plt.show()

    plt.figure(figsize=(8,5))
    plt.plot(np.arange(ATPR.shape[0]), ATPR, lw=2, color='k')
    plt.xlabel(r'Iterations', fontsize=16)
    plt.ylabel(r'ATPR␣Output', fontsize=16)
    plt.savefig('q3/ATPR.pdf', bbox_inches='tight')
    plt.show()

    plt.figure(figsize=(8,4.5))
    plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=mach, vmin=0.9, vmax=2.5, cmap='jet', shading='flat')
    plt.axis('off')
    plt.colorbar(label='Mach␣Number')
    plt.savefig('q3/Machfield.pdf', bbox_inches='tight')
    plt.show()

    plt.figure(figsize=(8,4.5))
    plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=pt, vmin=6.5, vmax=7.6, cmap='jet', shading='flat')
    plt.axis('off')
    plt.colorbar(label='Total␣Pressure')
    plt.savefig('q3/Pfield.pdf', bbox_inches='tight')
    plt.show()

def mesh_adapt(alpha):

    # Plot sequence of adapted meshes
    # Plot two figs. (Mach Number and the Total Pressure) for the finest mesh
    # Plot ATPR output vs. number of cells in a mesh (last ATPR calculation per
        iteration)

    #mesh = readgri('mesh0.gri')
    #for i in range(6):
    #    plotmesh(mesh, 'q4/mesh' + str(i) + '.pdf')

    u, err, ATPR, V, E, BE, IE = solve(alpha, mesh)
    mach, pt = post_process(u)
    adapt(u, mach, V, E, IE, BE)

def vary_alpha():

    # Vary alpha from 0.5:0.5:3 degrees
    # Run same adaptive iterations for each alpha at least 5
    # Plot ATPR from finest mesh vs. alpha and discuss trend
    alphas = np.arange(0.5,3.5, step=0.5)
    atpr_out = np.zeros(6); k = 0
    for i in alphas:

        start = time.time()
        u, err, ATPR, V, E, BE, IE = solve(i, mesh); end = time.time(); print('
            Elapsed␣Time␣%.2f'%(end - start))
        mach, pt = post_process(u)

        plt.figure(figsize=(8,4.5))
        plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=mach, vmin=0.9, vmax
            =2.5, cmap='jet', shading='flat')
        plt.axis('off')
        plt.savefig('q5/mach_a' + str(int(i*10)) + '.pdf', bbox_inches='tight')
        plt.pause(0.2)
        plt.close()

        plt.figure(figsize=(8,4.5))
        plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=pt, vmin=6.5, vmax=7.6,
             cmap='jet', shading='flat')
        plt.axis('off')
        plt.savefig('q5/pt_a' + str(int(i*10)) + '.pdf', bbox_inches='tight')
        plt.pause(0.2)
        plt.close()

        atpr_out[k] = ATPR[len(ATPR)-1]; k += 1


    f = open('q5/atpr_out', 'w'); output = ''
    for i in range(6):
        output += r'%.1f\degree␣&␣%.3f␣\\'%(alphas[i], atpr_out[i])
    f.write(output)
    f.close()

    plt.figure(figsize=(9,5))
    plt.plot(alphas, atpr_out, lw=2, color='k')
    plt.xlabel(r'Angle␣of␣attack,␣$\alpha$', fontsize=16)
    plt.ylabel(r'ATPR␣Output', fontsize=16)
    plt.savefig('q5/ATPR.pdf', bbox_inches='tight')
    plt.show()

if __name__ == "__main__":
    #test_flux()
    run_fvm()
    #mesh_adapt(1)
    #vary_alpha()
```

A.2 Finite-Volume-Element Code

Algorithm 2: Finite-Volume-Element Code

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
from flux import RoeFlux
from readgri import readgri, writegri

def getIC(alpha, Ne):
    alpha = np.deg2rad(alpha); Minf = 2.2; gam = 1.4
    uinf = np.array([1, Minf*np.cos(alpha), Minf*np.sin(alpha), 1/(gam*(gam-1)) +
        Minf**2/2])

    u0 = np.zeros((Ne, 4))
    for i in range(4):
        u0[:,i] = uinf[i]
    u0[abs(u0) < 10**-10]

    return u0

def calcATPR(u0, u, alpha, V, BE):
    gam = 1.4

    Pinf = (gam-1)*(u0[0,3]-0.5*u0[0,0]*((u0[1,0]/u0[0,0])**2 + (u0[2,0]/u0[0,0])
        **2))
    Ptinf = Pinf*(1 + 0.5*(gam-1)*(2.2)**2)*(gam/(gam-1))

    ATPR = 0; d = 0
    for i in range(BE.shape[0]):
        n1, n2, e1, bgroup = BE[i,:]
        xl = V[n1,:]; xr = V[n2,:]
        uedge = u[e1,:]

        dy = xr[1] - xl[1]

        if bgroup == 1: # Exit
            uvel = uedge[1]/uedge[0]; vvel = uedge[2]/uedge[0]
            q = np.sqrt(uvel**2 + vvel**2)
            P = (gam-1)*(uedge[3]-0.5*uedge[0]*q**2)
            c = np.sqrt(gam*P/uedge[0])
            mach = q/c
            Pt = P*(1 + 0.5*(gam-1)*mach**2)**(gam/(gam-1))

            d += dy
            ATPR += Pt*dy/Ptinf

    ATPR *= 1/d
    return ATPR

def solve(alpha, mesh):
    V = mesh['V']; E = mesh['E']; BE = mesh['BE']; IE = mesh['IE']

    u0 = getIC(alpha, E.shape[0]); u = u0.copy(); ATPR = np.array([calcATPR(u0,u,1,V
        ,BE)])
    R = np.zeros((E.shape[0], 4)); dta = R.copy(); err = np.array([1]); itr = 0

    while err[err.shape[0]-1] > 10**(-5):
    #for k in range(50):
        R *= 0; dta *= 0
        for i in range(IE.shape[0]):
            n1, n2, e1, e2 = IE[i,:]
            xl = V[n1,:]; xr = V[n2,:]
            ul = u[e1,:]; ur = u[e2,:]

            dx = xr - xl; deltal = LA.norm(dx)
            nhat = np.array([dx[1], -dx[0]])/deltal
            F, FL, FR, ls = RoeFlux(ul, ur, nhat)
            R[e1,:] += F*deltal; R[e2,:] -= F*deltal
            dta[e1,:] += ls*deltal; dta[e2,:] += ls*deltal

        for i in range(BE.shape[0]):
            n1, n2, e1, bgroup = BE[i,:]
            xl = V[n1,:]; xr = V[n2,:]
            uedge = u[e1,:]

            dx = xr - xl; deltal = LA.norm(dx)
            nhat = np.array([dx[1], -dx[0]])/deltal

            if bgroup == 0: # Engine - Invscid
                vp = np.array([uedge[1], uedge[2]])/uedge[0]
                vb = vp - np.dot(vp, nhat)*nhat
                pb = 0.4*(uedge[3] - 0.5*uedge[0]*(vb[0]**2 + vb[1]**2))
                ignore, FL, FR, ls = RoeFlux(uedge, u0[0,:], nhat)

                F = pb*np.array([0, nhat[0], nhat[1], 0])
            elif bgroup == 1 or bgroup == 2: # Exit/Outflow - Supersonic Outflow
                F, FL, FR, ls = RoeFlux(uedge, uedge, nhat)
            elif bgroup == 3: # Inflow
                F, FL, FR, ls = RoeFlux(uedge, u0[0,:], nhat)

            R[e1,:] += F*deltal
            dta[e1,:] += ls*deltal

        dta = 2/dta
        u -= np.multiply(dta, R)
        err = np.append(err, sum(sum(abs(R))))

        ATPR = np.append(ATPR, calcATPR(u0,u,1,V,BE))
        print('Iteration:␣%3d,\tError:␣%.3e,␣ATPR:␣%.3f'%(itr, err[err.shape[0]-1],
            ATPR[ATPR.shape[0]-2])); itr += 1

    return u, err[1:], ATPR, V, E, BE, IE
```

A.3 Roe Flux Python Implementation

Algorithm 3: Roe Flux Implementation

```python
import numpy as np
from numpy import linalg as LA

def RoeFlux(Ul, Ur, n):
    gam = 1.4

    # Left side arguments
    rhol = Ul[0]; ul = Ul[1]/rhol; vl = Ul[2]/rhol; rhoEl = Ul[3]
    pl = (gam-1)*(rhoEl-0.5*rhol*(ul**2 + vl**2))
    Hl = (rhoEl + pl)/rhol

    # Right side arguments
    rhor = Ur[0]; ur = Ur[1]/rhor; vr = Ur[2]/rhor; rhoEr = Ur[3]
    pr = (gam-1)*(rhoEr-0.5*rhor*(ur**2 + vr**2))
    Hr = (rhoEr + pr)/rhor

    # Left and Right side fluxes
    FL = np.array([np.dot([Ul[1],Ul[2]], n), np.dot([Ul[1]*ul+pl, Ul[2]*ul],n), np.
        dot([Ul[1]*vl, Ul[2]*vl+pl],n), Hl*np.dot([Ul[1],Ul[2]],n)])
    FR = np.array([np.dot([Ur[1],Ur[2]], n), np.dot([Ur[1]*ur+pr, Ur[2]*ur],n), np.
        dot([Ur[1]*vr, Ur[2]*vr+pr],n), Hr*np.dot([Ur[1],Ur[2]],n)])

    # Roe-Averages
    RHS, ls = ROE_Avg(ul,vl,rhol,Hl,rhoEl, ur,vr,rhor,Hr,rhoEr, n)
    F = 0.5*(FL + FR) - 0.5*RHS

    return F, FL, FR, ls

def ROE_Avg(ul,vl,rhol,Hl,rhoEl, ur,vr,rhor,Hr,rhoEr, n):
    gam = 1.4
    vell = np.array([ul, vl]); velr = np.array([ur, vr])

    # Calculating Roe average
    v = (np.sqrt(rhol)*vell + np.sqrt(rhor)*velr)/(np.sqrt(rhol) + np.sqrt(rhor))
    H = (np.sqrt(rhol)*Hl + np.sqrt(rhor)*Hr)/(np.sqrt(rhol) + np.sqrt(rhor))

    # Calculating eigenvalues
    q = LA.norm(v)
    c = np.sqrt((gam-1.0)*(H - 0.5*q**2))
    u = np.dot(v, n)
    ls = abs(np.array([u+c, u-c, u]))

    # Apply the entropy fix
    ls[abs(ls) < 0.1*c] = ((0.1*c)**2 + ls[abs(ls) < 0.1*c]**2)/(2*0.1*c)

    delrho = rhor - rhol; delmo = np.array([rhor*ur - rhol*ul, rhor*vr - rhol*vl]);
        dele = rhoEr - rhoEl
    s1 = 0.5*(abs(ls[0]) + abs(ls[1])); s2 = 0.5*(abs(ls[0]) - abs(ls[1]))
    G1 = (gam-1.0)*(0.5*q**2*delrho - np.dot(v, delmo) + dele); G2 = -u*delrho + np.
        dot(delmo, n)
    C1 = G1*(c**-2)*(s1 - abs(ls[2])) + G2*(c**-1)*s2; C2 = G1*(c**-1)*s2 + (s1 -
        abs(ls[2]))*G2

    RHS = np.array([ls[2]*delrho+C1, ls[2]*delmo[0]+C1*v[0]+C2*n[0], ls[2]*delmo[1]+
        C1*v[1]+C2*n[1], ls[2]*dele+C1*H+C2*u])

    return RHS, max(ls)
```

A.4 Adaptive Mesh Python Implementation

Algorithm 4: Adaptive Mesh Implementation

```
import numpy as np
from numpy import linalg as LA
from readgri import writegri
from edgehash import edgehash
import matplotlib.pyplot as plt

def mach_perp(u, nhat):
    uvel = u[1]/u[0]; v = u[2]/u[0]     # Calculate the velocity
    q = np.dot(np.array([uvel, v]), nhat) # Determine the perpendicular speed
    P = (1.4 - 1)*(u[3] - 0.5*u[0]*q**2) # Calculate pressure
    H = (u[3] + P)/u[0]                 # Calculate enthalpy
    c = np.sqrt(0.4*(H - 0.5*q**2))     # Calculate speed of sound
    mach = q/c                          # Calculate the Mach number

    return mach

def check_vert(Vvec, x):
    check = True
    # Loop over the vertices
    for i in range(Vvec.shape[0]):
        # If this vertex exists return False
        if x[0] == Vvec[i,0] and x[1] == Vvec[i,1]:
            check = False
            break

    return check

def genflags(u, mach, V, E, IE, BE):

    # Pre-allocate flag array
    flags = np.zeros(((IE.shape[0] + BE.shape[0]),2)); flags[:,0] = np.arange(flags.
        shape[0]);k = 0

    # Iterate over the interior edges
    for i in range(IE.shape[0]):
        n1, n2, e1, e2 = IE[i,:]        # Nodes and elements from interior edge
        xl = V[n1,:]; xr = V[n2,:]      # Vertice values
        machl = mach[e1]; machr = mach[e2] # Mach numbers at each element
        dx = xr - xl; deltal = LA.norm(dx) # Determine the length of the edge
        eps = abs(machr - machl)*deltal # Calculate the error

        flags[k,1] += eps; k += 1       # Add the edge error

    # Iterate over the boundary edges
    for i in range(BE.shape[0]):
        n1, n2, e1, bgroup = BE[i,:]    # Node and elements from boundary edge
        if bgroup == 0: # Engine
            xl = V[n1,:]; xr = V[n2,:]      # Vertice values
            uedge = u[e1,:]                 # State at edge
            dx = xr - xl; deltal = LA.norm(dx) # Determine the length of the edge
            nhat = np.array([dx[1], -dx[0]])/deltal # Determine the normal off the
                boundary edge
            machperp = mach_perp(uedge, nhat) # Calculate the perpendicular Mach
                number
            eps = abs(machperp)*deltal      # Calculate error

            flags[k,1] += eps               # Add the edge error
        k += 1

    # Sort from largest to smallest errors
    flags = flags[flags[:,1].argsort()]; flags = np.flipud(flags)
    # Remove all outliers to be refined
    ind = int(np.ceil(flags.shape[0] * 0.03))
    ind = int(np.ceil(flags.shape[0] * 0.1))
    flags[ind:(flags.shape[0]-1),1] = 0

    # Sort the errors increasing the edge number to iterate
    flags = flags[flags[:,0].argsort()]

    return flags

def genV(flags, V, E, IE, BE):
    Vcopy = V.copy(); k = 0
    for i in range(IE.shape[0]):
        err = flags[k,1]
        if err > 0:
            ig, ig, e1, e2 = IE[i,:]
            for j in np.array([e1,e2]):
                n1, n2, n3 = E[j,:]
                x1 = V[n1,:]; x2 = V[n2,:]; x3 = V[n3,:]

                # Conditionals to prevent duplicate nodes
                if check_vert(Vcopy, (x2-x1)/2 +x1):
                    Vcopy = np.append(Vcopy, np.array([(x2-x1)/2 +x1]), axis=0)
                if check_vert(Vcopy, (x3-x1)/2 +x1):
                    Vcopy = np.append(Vcopy, np.array([(x3-x1)/2 +x1]), axis=0)
                if check_vert(Vcopy, (x3-x2)/2 +x2):
                    Vcopy = np.append(Vcopy, np.array([(x3-x2)/2 +x2]), axis=0)
        k += 1

    for i in range(BE.shape[0]):
        err = flags[k,1]
        if err > 0:
            ig, ig, e1, ig = BE[i,:]
            n1, n2, n3 = E[e1,:]
            x1 = V[n1,:]; x2 = V[n2,:]; x3 = V[n3,:]

            # Conditionals to prevent duplicate nodes
            if check_vert(Vcopy, (x2-x1)/2 +x1):
                Vcopy = np.append(Vcopy, np.array([(x2-x1)/2 +x1]), axis=0)
            if check_vert(Vcopy, (x3-x1)/2 +x1):
                Vcopy = np.append(Vcopy, np.array([(x3-x1)/2 +x1]), axis=0)
            if check_vert(Vcopy, (x3-x2)/2 +x2):
                Vcopy = np.append(Vcopy, np.array([(x3-x2)/2 +x2]), axis=0)
        k += 1

    return Vcopy

def genUE(u, Vcopy, V, E, IE, BE):
    Ecopy = E.copy(); Ucopy = u.copy()
    for i in range(Ecopy.shape[0]):
        n1, n2, n3 = Ecopy[i,:]
        x1 = V[int(n1),:]; x2 = V[int(n2),:]; x3 = V[int(n3),:]
        vals = np.array([(x2-x1)/2 +x1, (x3-x1)/2 +x1, (x3-x2)/2 +x2])

        # Generate nodes for each element
        nodes = np.array([])
        for k in vals:
            check, ind = vert_ind(Vcopy, k)
            if check:
                nodes = np.append(nodes, ind)

        if nodes.shape[0] == 3:
            # Ensure that the nodes are CCW
            if isCCW(Vcopy[int(nodes[0]),:], Vcopy[int(nodes[1]),:], Vcopy[int(nodes
                [2]),:]) != 1:
                nodes = np.flip(nodes)

            # Loop through the nodes
            for k in range(3):
                # Start at the nodes N1 -> N2 for consistency
                if Vcopy[int(nodes[k]),0] == vals[0,0] and Vcopy[int(nodes[k]),1] ==
                    vals[0,1]:
                    Ecopy[i,:] = np.array([n1, nodes[k], nodes[(k+2)%3]]) # Replace
                        the ith element with new element

                    ind1 = np.array([nodes[k], n2, nodes[(k+1)%3]])
                    ind2 = np.array([nodes[k], nodes[(k+1)%3], nodes[(k+2)%3]])
                    ind3 = np.array([nodes[(k+1)%3], nodes[(k+2)%3], n3])
                    if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),:]) != 1:
                        ind1 = np.flip(ind1)
                    if isCCW(Vcopy[int(ind2[0]),:], Vcopy[int(ind2[1]),:], Vcopy[int(ind2[2]),:]) != 1:
                        ind2 = np.flip(ind2)
                    if isCCW(Vcopy[int(ind3[0]),:], Vcopy[int(ind3[1]),:], Vcopy[int(ind3[2]),:]) != 1:
                        ind3 = np.flip(ind3)
                    # Append new elements
                    Ecopy = np.append(Ecopy, np.transpose(np.array([[ind1[0]], [ind1[1]], [ind1[2]]])), axis=0)
                    Ecopy = np.append(Ecopy, np.transpose(np.array([[ind2[0]], [ind2[1]], [ind2[2]]])), axis=0)
                    Ecopy = np.append(Ecopy, np.transpose(np.array([[ind3[0]], [ind3[1]], [ind3[2]]])), axis=0)

                    for l in range(3):
                        Ucopy = np.append(Ucopy, np.transpose(np.array([[u[i,0]], [u[i,1]], [u[i,2]], [u[i,3]]])), axis=0)
                    break

        elif nodes.shape[0] == 2:
            node_ind = np.array([n1, n2, n3])

            if isCCW(Vcopy[int(node_ind[0]),:], Vcopy[int(node_ind[1]),:], Vcopy[int(node_ind[2]),:]) != 1:
                node_ind = np.flip(node_ind)

            dl_old = 0
            for k in range(3):
                for j in range(2):
                    dl = LA.norm(Vcopy[int(node_ind[k]),:] - Vcopy[int(nodes[j]),:])
                    if dl > dl_old:
                        dl_old = dl

                        nodetemp = nodes
                        node_indtemp = np.array([node_ind[k], node_ind[(k+1)%3], node_ind[(k+2)%3]])
                        if j == 0:
                            ind1 = np.array([node_indtemp[0], nodetemp[0], nodetemp[1]])
                            ind2 = np.array([node_indtemp[0], node_indtemp[1], nodetemp[1]])
                            ind3 = np.array([nodetemp[1], node_indtemp[2], node_indtemp[2]])
                        else:
                            ind1 = np.array([node_indtemp[0], nodetemp[0], nodetemp[1]])
                            ind2 = np.array([node_indtemp[0], node_indtemp[2], nodetemp[1]])
                            ind3 = np.array([nodetemp[1], node_indtemp[2], node_indtemp[1]])








            if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),:]) != 1:
                ind1 = np.flip(ind1)
            if isCCW(Vcopy[int(ind2[0]),:], Vcopy[int(ind2[1]),:], Vcopy[int(ind2[2]),:]) != 1:
                ind2 = np.flip(ind2)
            if isCCW(Vcopy[int(ind3[0]),:], Vcopy[int(ind3[1]),:], Vcopy[int(ind3[2]),:]) != 1:
                ind3 = np.flip(ind3)

```

```python
            Ecopy[i,:] = np.array([ind1[0], ind1[1], ind1[2]])

            Ecopy = np.append(Ecopy, np.transpose(np.array([[ind2[0]], [ind2[1]], [ind2[2]]])), axis=0)
            Ecopy = np.append(Ecopy, np.transpose(np.array([[ind3[0]], [ind3[1]], [ind3[2]]])), axis=0)

            for k in range(2):
                Ucopy = np.append(Ucopy, np.transpose(np.array([[u[i,0]], [u[i,1]], [u[i,2]], [u[i,3]]])), axis=0)

        elif nodes.shape[0] == 1:

            for k in range(3):
                if vals[k,0] == Vcopy[int(nodes[0]),0] and vals[k,1] == Vcopy[int(nodes[0]),1]:
                    if k == 0:
                        ind1 = np.array([n1, nodes[0], n3])
                        ind2 = np.array([n2, n3, nodes[0]])
                    elif k == 1:
                        ind1 = np.array([n1, nodes[0], n2])
                        ind2 = np.array([n3, n2, nodes[0]])
                    elif k == 2:
                        ind1 = np.array([n2, nodes[0], n1])
                        ind2 = np.array([n3, n1, nodes[0]])

            if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),:]) != 1:
                ind1 = np.flip(ind1)
            if isCCW(Vcopy[int(ind2[0]),:], Vcopy[int(ind2[1]),:], Vcopy[int(ind2[2]),:]) != 1:
                ind2 = np.flip(ind2)

            Ecopy[i,:] = np.array([ind1[0], ind1[1], ind1[2]])
            Ecopy = np.append(Ecopy, np.transpose(np.array([[ind2[0]], [ind2[1]], [ind2[2]]])), axis=0)


            Ucopy = np.append(Ucopy, np.transpose(np.array([[u[i,0]], [u[i,1]], [u[i,2]], [u[i,3]]])), axis=0)
    Ecopy = Ecopy.astype(int)
    return Ucopy, Ecopy

def genB(u, V, Vcopy, BE):
    Bcopy = BE.copy()
    for i in range(Bcopy.shape[0]):
        n1, n2, e1, bgroup = BE[i,:]
        xl = V[n1,:]; xr = V[n2,:]

        check, ind = vert_ind(Vcopy, 0.5*(xr-xl) + xl)
        if check:

            Bcopy[i,:] = np.array([n1, ind.item(), i, bgroup])
            Bcopy = np.append(Bcopy, np.transpose(np.array([[ind.item()],[n2], [Bcopy.shape[0]+1], [bgroup]])), axis=0)

    B0 = np.array([[-1,-1]]); B1 = B0.copy(); B2 = B0.copy(); B3 = B0.copy()
    for i in range(Bcopy.shape[0]):
        n1, n2, e, bname = Bcopy[i,:]
        if bname == 0:
            B0 = np.append(B0, np.transpose(np.array([[n1], [n2]])), axis=0)
        if bname == 1:
            B1 = np.append(B1, np.transpose(np.array([[n1], [n2]])), axis=0)
        if bname == 2:
            B2 = np.append(B2, np.transpose(np.array([[n1], [n2]])), axis=0)
        if bname == 3:
            B3 = np.append(B3, np.transpose(np.array([[n1], [n2]])), axis=0)
    B0 = B0[1:,:]; B1 = B1[1:,:]; B2 = B2[1:,:]; B3 = B3[1:,:];
    B = [B0.astype(int), B1.astype(int), B2.astype(int), B3.astype(int)]

    return B
```

```

def isboundary(nodestate, BEvec, Vvec):
    edgevals = np.array([])
    for k in range(3):
        node = Vvec[int(nodestate[k])]
        for i in range(BEvec.shape[0]):
            n1, ig, ig, ig = BEvec[i,:]     # Node and elements from boundary edge
            x1 = Vvec[n1,:]
            if node[0] == x1[0] and node[1] == x1[1]:
                edgevals = np.append(edgevals, nodestate[k])

    return edgevals

def isCCW(a, b, c):
    cross_val = (b[0] - a[0])*(c[1] - a[1]) - (c[0] - a[0])*(b[1] - a[1])

    if cross_val > 0:
        cross_val = 1
    elif cross_val < 0:
        cross_val = -1
    else:
        cross_val = 0

    return cross_val

def orientation(p, q, r):
    val = (q[1] - p[1])*(r[0]-q[0]) - (q[0] - p[0])*(r[1] - q[1])

    return val

def doIntersect(a, b, c, d):

    check = False

    m = (b[1] - a[1])/(b[0] - a[0])

    xlin = np.linspace(a[0], b[0], endpoint = True, num=25)
    for i in range(25):
        y = m*(xlin[i] - a[0]) + a[1]

        if y < max(np.array([c[1], d[1]])) and y > min(np.array([c[1], d[1]])) and
             xlin[i] < max(np.array([c[0], d[0]])) and xlin[i] > min(np.array([c
            [0], d[0]])):
            check = True

    return check

def vert_ind(Vvec, x):
    check = False; ind = np.array([])
    # Loop over the vertices
    for i in range(Vvec.shape[0]):
        # If this vertex exists return False
        if x[0] == Vvec[i,0] and x[1] == Vvec[i,1]:
            check = True; ind = np.append(ind, [i])

    return check, ind

def genArea(a,b,c):
    s = 0.5*(a+b+c)
    area = (s*(s-a)*(s-b)*(s-c)) ** 0.5

    return area

def adapt(u, mach, V, E, IE, BE):

    flags = genflags(u, mach, V, E, IE, BE)
    Vcopy = genV(flags, V, E, IE, BE)
    Ucopy, Ecopy = genUE(u, Vcopy, V, E, IE, BE)
    B = genB(u, V, Vcopy, BE)
    IEcopy, BEcopy = edgehash(Ecopy, B)

```

```python
    Mesh = {'V':Vcopy, 'E':Ecopy, 'IE':IEcopy, 'BE':BEcopy, 'Bname':['Engine', 'Exit
        ', 'Outflow', 'Inflow'] }
    writegri(Mesh, 'test1.gri')

    return u, Vcopy, Ecopy, IEcopy, BEcopy

def plotmesh(V, B, E):

    f = plt.figure(figsize=(12,12))
    plt.triplot(V[:,0], V[:,1], E, 'k-')
    plt.scatter(V[:,0], V[:,1])
    #for i in range(BE.shape[0]):
    #    plt.plot(V[BE[i,0:2],0],V[BE[i,0:2],1], '-', linewidth=2, color='black')
    plt.axis('equal'); plt.axis('off')
    f.tight_layout();
    plt.show()
```

B Additional Supporting Code

Algorithm 5: Python Edge Hash

```
import numpy as np
from scipy import sparse


#-----------------------------------------------------------
# Identifies interior and boundary edges given element-to-node
# IE contains (n1, n2, elem1, elem2) for each interior edge
# BE contains (n1, n2, elem) for each boundary edge
def edgehash(E, B):
    Ne = E.shape[0]; Nn = np.amax(E)+1
    H = sparse.lil_matrix((Nn, Nn), dtype=np.int)
    IE = np.zeros([int(np.ceil(Ne*1.5)),4], dtype=np.int)
    ni = 0
    for e in range(Ne):
        for i in range(3):
            n1, n2 = E[e,i], E[e,(i+1)%3]
            if (H[n2,n1] == 0):
                H[n1,n2] = e+1
            else:
                eR = H[n2,n1]-1
                IE[ni,:] = n1, n2, e, eR
                H[n2,n1] = 0
                ni += 1
    IE = IE[0:ni,:]
    # boundaries
    nb0 = nb = 0
    for g in range(len(B)):
        nb0 += B[g].shape[0]
    BE = np.zeros([nb0,4], dtype=np.int)
    for g in range(len(B)):
        Bi = B[g]
        for b in range(Bi.shape[0]):
            n1, n2 = Bi[b,0], Bi[b,1]
            if (H[n1,n2] == 0): n1,n2 = n2,n1
            BE[nb,:] = n1, n2, H[n1,n2]-1, g
            nb += 1
    return IE, BE
```

Algorithm 6: Python Plot Mesh

```python
import numpy as np
import matplotlib.pyplot as plt
from readgri import readgri

#-----------------------------------------------------------
def plotmesh(Mesh, fname):
    V = Mesh['V']; E = Mesh['E']; BE = Mesh['BE']

    f = plt.figure(figsize=(12,12))
    plt.triplot(V[:,0], V[:,1], E, 'k-')
    for i in range(BE.shape[0]):
        plt.plot(V[BE[i,0:2],0],V[BE[i,0:2],1], '-', linewidth=2, color='black')
    plt.axis('equal'); plt.axis('off')
    f.tight_layout();
    plt.savefig(fname, bbox_inches='tight')
    plt.close()
```

References

[1] K. Fidkowski, “Computational fluid dynamics,” September 2020.

[2] Gryphon, “Roe flux differencing scheme: The approximate riemann problem.”