

# Project 2: Supersonic Engine Analysis

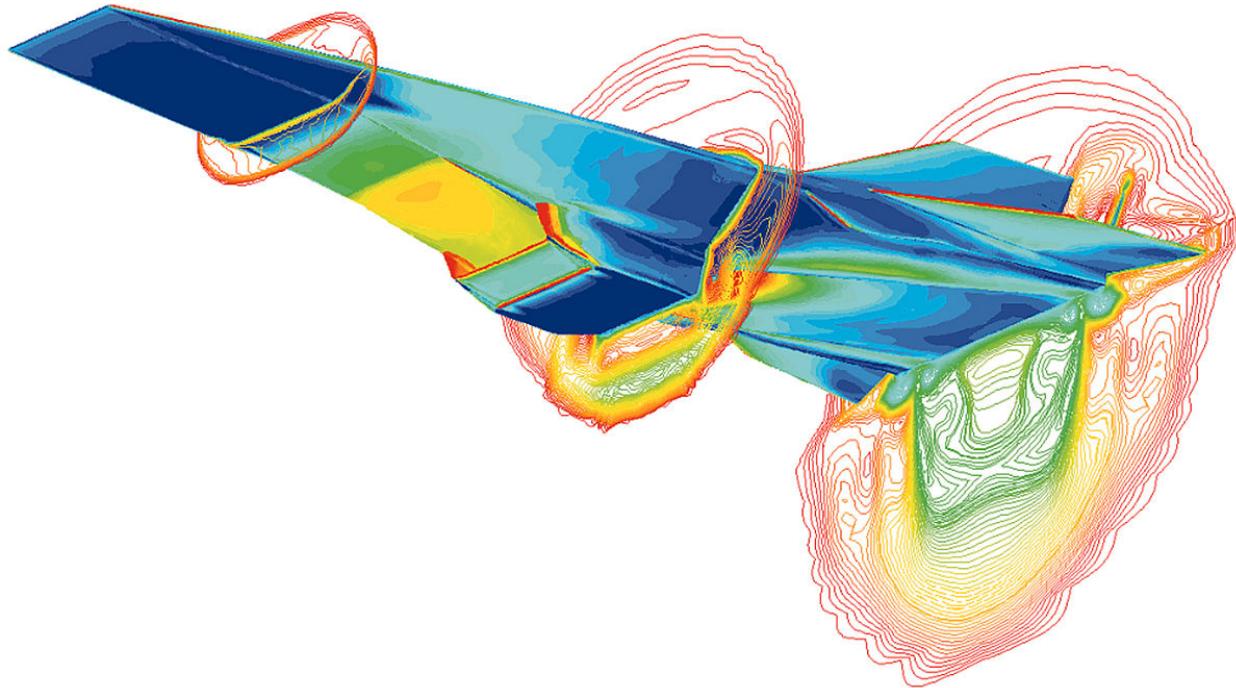
Aerospace 523: Computational Fluid Dynamics I

Graduate Aerospace Engineering

University of Michigan, Ann Arbor

By: Dan Card, dcard@umich.edu

Date: December 6, 2020



---

NASA X-43 Hypersonic Airplane



## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Numerical Method</b>	<b>6</b>
<b>3</b>	<b>Adaptation</b>	<b>9</b>
<b>4</b>	<b>Tasks and Deliverables</b>	<b>10</b>
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	23
4.4.3	Adapted Mesh ATPR Convergence	25
4.5	Adaptive Iterations	26
4.5.1	ATPR Versus Angle of Attack	26
4.5.2	Flow Fields for Varying Angle of Attacks	27
<b>Appendices</b>		<b>29</b>
<b>Appendix A</b>	<b>Python Implementation</b>	<b>29</b>
A.1	Main Driving Code	29
A.2	Finite-Volume-Element Code	34
A.3	Roe Flux Python Implementation	36
A.4	Adaptive Mesh Python Implementation	37
<b>Appendix B</b>	<b>Additional Supporting Code</b>	<b>43</b>
<b>References</b>		<b>45</b>

## 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 . . . . .	22
9	Field Plot of Mach Number for Adapted Mesh . . . . .	23
10	Field Plot of Total Pressure for Adapted Mesh . . . . .	24
11	ATPR Convergence with Cell Number . . . . .	25
12	ATPR and Angle of Attack . . . . .	26
13	Mach Field with Varying Angle of Attack . . . . .	27
14	Total Pressure Field with Varying Angle of Attack . . . . .	28

## 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 . . . . .	26

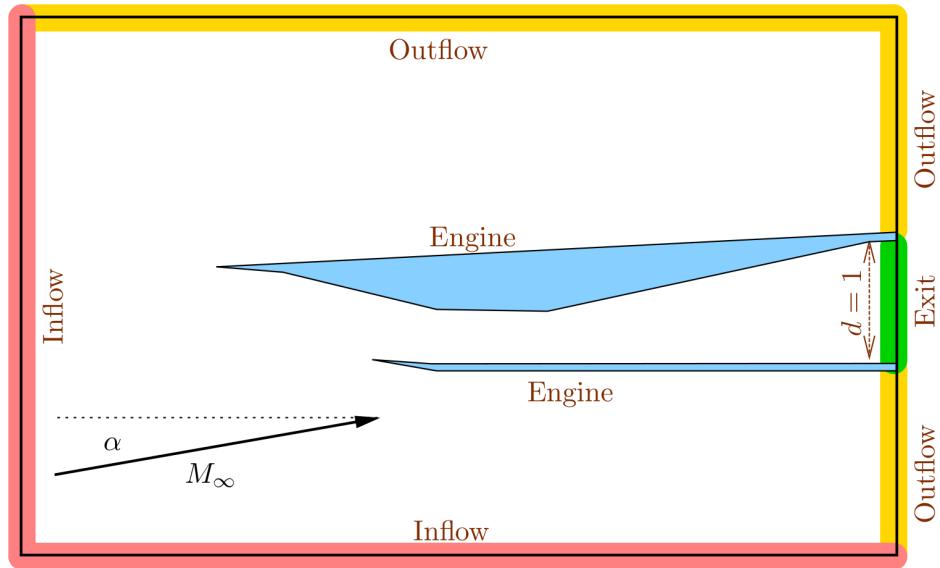
## List of Algorithms

1	Main Driving Code . . . . .	29
2	Finite-Volume-Element Code . . . . .	34
3	Roe Flux Implementation . . . . .	36
4	Adaptive Mesh Implementation . . . . .	37
5	Python Edge Hash . . . . .	43
6	Python Plot Mesh . . . . .	44

# 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

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

where  $M_\infty$  is the free-stream Mach number, and  $\alpha$  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  $\alpha \in [0, 3^\circ]$ , with a baseline value of  $\alpha = 1^\circ$ . 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,  $M_\infty$ .

**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,

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

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

$$A_i \frac{d\mathbf{u}_i}{dt} + \mathbf{R}_i = \mathbf{0} \rightarrow \frac{d\mathbf{u}_i}{dt} = -\frac{1}{A_i} \mathbf{R}_i. \quad (3)$$

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

$$\mathbf{R}_i = \sum_{e=1}^3 \hat{\mathbf{F}}(\mathbf{u}_i, \mathbf{u}_{N(i,e)}, \vec{n}_{i,e}) \Delta l_{i,e} \quad (4)$$

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:  $\Delta t_i$ . Defining the CFL number for cell  $i$  as,

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

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,

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

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

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

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:

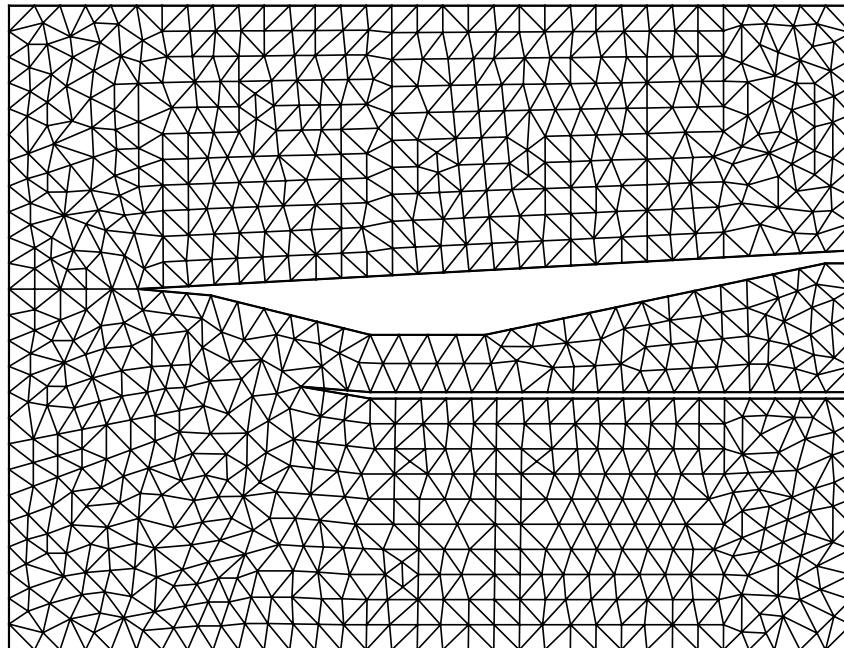
- Consistency check:  $\mathbf{F}(\mathbf{u}_L, \mathbf{u}_L, \vec{n})$  should be the same as  $\tilde{\mathbf{F}}(\mathbf{U}_L) \cdot \vec{n}$  (the analytical flux dotted with the normal).
- Flipping the direction: check that  $\mathbf{F}(\mathbf{u}_L, \mathbf{u}_R, \vec{n}) = -\mathbf{F}(\mathbf{u}_R, \mathbf{u}_L, -\vec{n})$ .
- 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

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

where  $\Delta t_i^n$  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  $e$ .

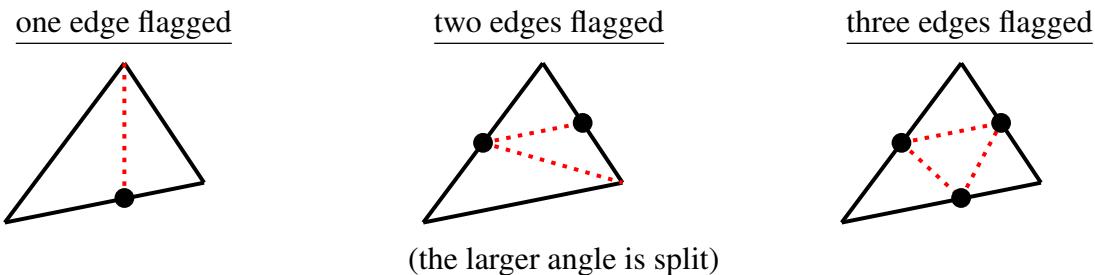
You can assume that the error indicator on the farfield boundary edges is zero. On the engine boundary (solid wall), define the error indicator by

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

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

After calculating the error indicators  $\epsilon_e$  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]

$$\hat{\mathbf{F}} = \frac{1}{2} (\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2} \left| \frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^*) \right| (\mathbf{u}_R - \mathbf{u}_L) \quad (9)$$

In this expression  $\left| \frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^*) \right|$  refers to the absolute values of the eigenvalues, i.e.  $\mathbf{R}|\Lambda|\mathbf{L}$ , 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.

$$\hat{\mathbf{F}} = \frac{1}{2} (\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2} \begin{bmatrix} |\lambda_3| \Delta \rho + C_1 \\ |\lambda_3| \Delta(\rho \vec{v}) + C_1 \vec{v} + C_2 \hat{n} \\ |\lambda_3| \Delta(\rho E) + C_1 H + C_2 (\vec{v} \cdot \hat{n}) \end{bmatrix} \quad (10)$$

Where further expansions of the constants above give,

$$\begin{aligned} [\lambda_1, \lambda_2, \lambda_3, \lambda_4] &= [u + c, u - c, u, u] \\ \vec{v} &= \frac{\sqrt{\rho_L} \vec{v}_L + \sqrt{\rho_R} \vec{v}_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, \quad H = \frac{\sqrt{\rho_L} H_L + \sqrt{\rho_R} H_R}{\sqrt{\rho_L} + \sqrt{\rho_R}} \\ C_1 &= \frac{G_1}{c^2} (s_1 - |\lambda_3|) + \frac{G_2}{c} s_2, \quad C_2 = \frac{G_1}{c} s_2 + (s_1 - |\lambda_3|) G_2 \\ G_1 &= (\gamma - 1) \left( \frac{q^2}{2} \Delta \rho - \vec{v} \cdot \Delta(\rho \vec{v}) + \Delta(\rho E) \right), \quad G_2 = -(\vec{v} \cdot \hat{n}) \Delta \rho + \Delta(\rho \vec{v}) \cdot \hat{n} \\ s_1 &= \frac{1}{2} (|\lambda|_1 + |\lambda|_2), \quad s_2 = \frac{1}{2} (|\lambda|_1 - |\lambda|_2) \end{aligned}$$

Where the difference in states is given by,

$$\Delta \mathbf{u} = \mathbf{u}_R - \mathbf{u}_L, \quad q^2 = u^2 + v^2$$

$$\mathbf{F}_L = \tilde{\mathbf{F}}(\mathbf{u}_L) \cdot \hat{n}, \quad \mathbf{F}_R = \tilde{\mathbf{F}}(\mathbf{u}_R) \cdot \hat{n}$$

However, to prevent expansion 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  $\epsilon$  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  $\alpha = 0^\circ$ ,  $M_\infty = 0.8$ ,  $\vec{n} = [1, 0]$  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	$\rho$	$\rho u$	$\rho v$	$\rho E$
$\hat{F}(u_l, u_l, \vec{n})$	0.800	1.354	0.000	2.256
$\vec{F}(\vec{U}_l) \cdot \vec{n}$	0.800	1.354	0.000	2.256
$\Delta F$	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  $\alpha = 0^\circ$ ,  $M_\infty = 2.2$ ,  $\vec{n} = [1, 0]$  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	$\rho$	$\rho u$	$\rho v$	$\rho E$
$\hat{F}(u_l, u_r, \vec{n})$	0.800	1.354	0.000	2.256
$-F(u_r, u_l, -\vec{n})$	0.800	1.354	-0.000	2.256
$\Delta F$	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	$\rho$	$\rho u$	$\rho v$	$\rho E$
$\hat{F}(u_l, u_r, \vec{n})$	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 and create new vertices on this mesh. Next will be to iterate through and determine how many new vertices are on a new element and from here generate an updated  $U$  and  $E$ . Next will be to then use the old boundary elements with the updated vertices matrix to determine the new boundary nodes and the elements that they are located at. Finally is to generate new interior and exterior edges from `edgehash(E,B)` and then write this out to a new .gri file for later mesh refinement.[3]

### 4.2.5 Miscellaneous Code

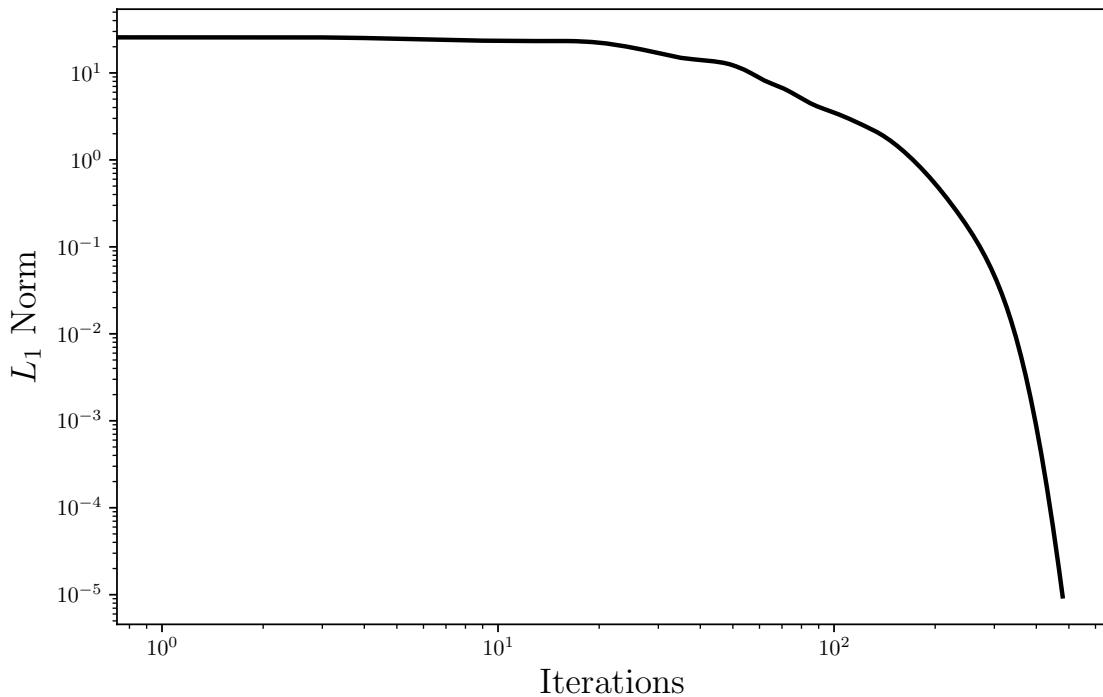
**Initial Condition** This supporting function will determine the initial condition depending on the angle of attack  $\alpha$ , and return the initial state for the solver code with the free-stream condition specified in Equation 1.

**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.

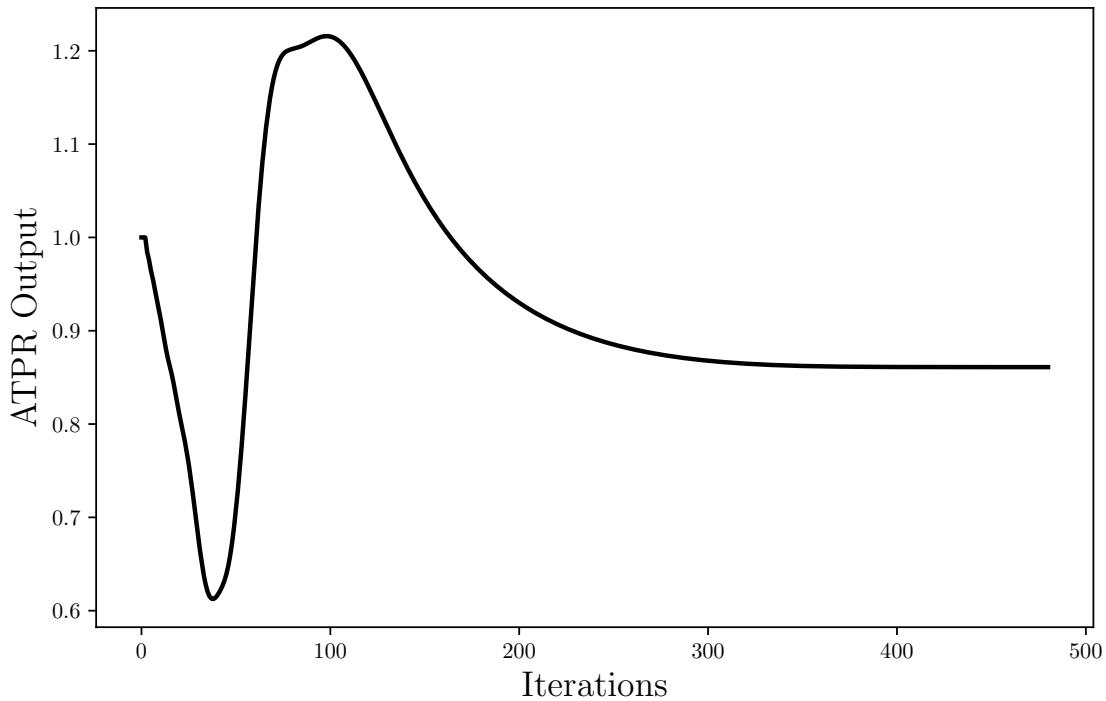
#### 4.3.1 $L_1$ Norm Convergence



**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

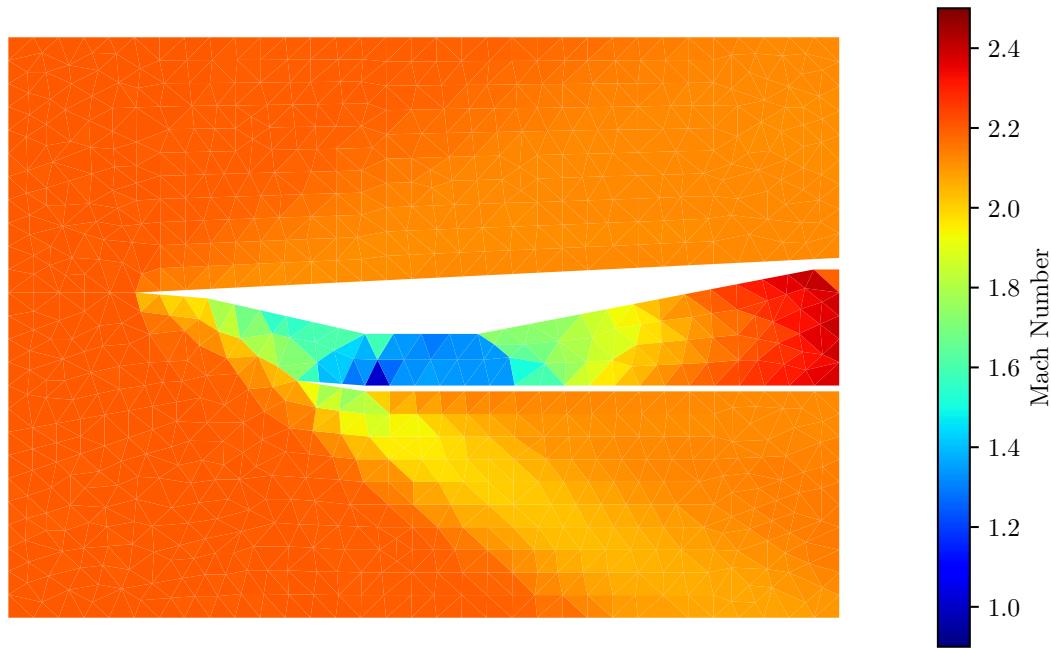


**Figure 5:** ATPR output for baseline mesh.

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.

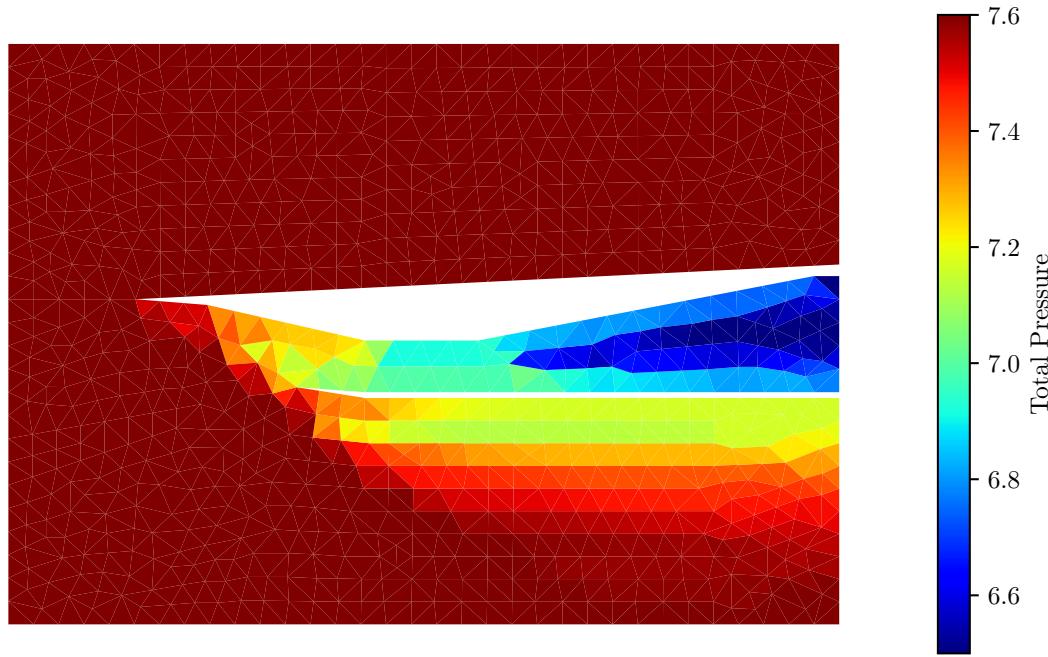
Also noteworthy is that the ATPR output dips as the simulations starts as the approximated state starts iterating towards a more physical starting condition. This can be seen as the large decrease in ATPR followed by another increase in ATPR until it starts to gradually smooth out to the physical approximation.

#### 4.3.3 Baseline Field Plots



**Figure 6:** Field plot of Mach number with  $\alpha = 1^\circ$ .

**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  $\alpha = 1^\circ$ . 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. This loss of information is shown by purely shaded regions with no visible Mach lines throughout the diffusor 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  $\alpha = 1^\circ$ .

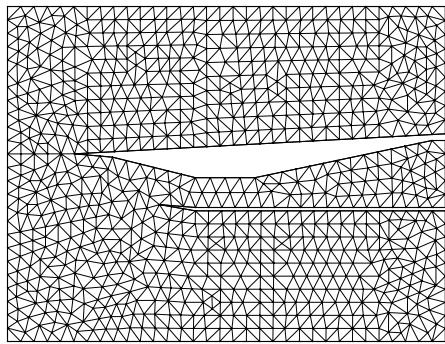
**Field Plot of Total Pressure** Above in Figure 7, is the field plot of the total pressure at  $M_\infty$  at an angle of  $\alpha = 1^\circ$ . Similar to Figure 6, there are some visible oblique shocks at the inlet of the engine. But similar to the mach field plot, much of the information is lost within 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

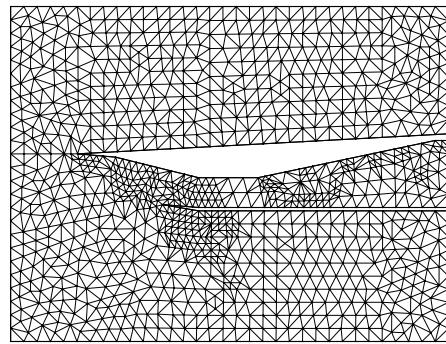
In this section I will implement an adaptive mesh function that will flag edges shown in Figure 3 given the discrepancy in Mach number across cell edges. The purpose of this function is to refine the mesh in the areas that are more prone to error; most notably the areas where there is large jumps in 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

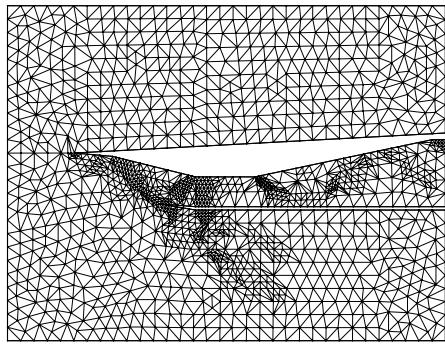
In this section, I will implement the Mach jumps into the code to refine the meshes to lower the error in the approximated solution. Using my code I will create a new mesh after each iterative solution and then refine the mesh that will be used on the next approximation. Shown on the following page in 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.



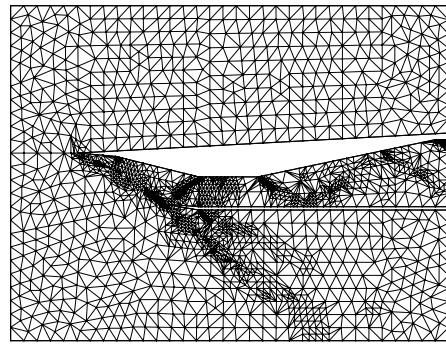
(a) Baseline mesh.



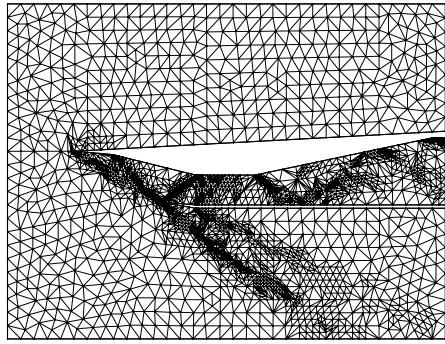
(b) Adapted mesh, iteration 1.



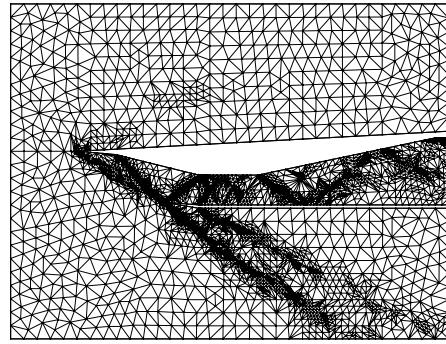
(c) Adapted mesh, iteration 2.



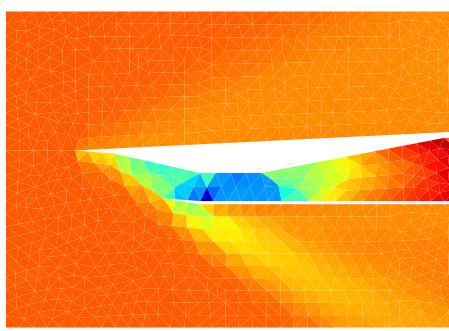
(d) Adapted mesh, iteration 3.



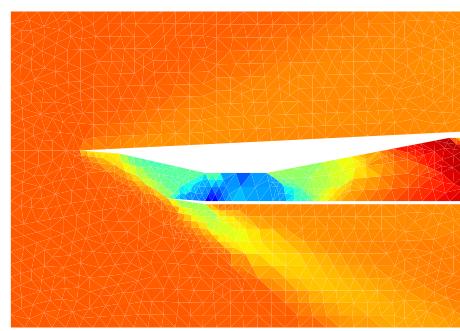
(e) Adapted mesh, iteration 4.



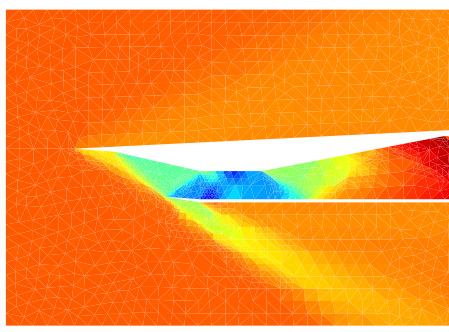
(f) Adapted mesh, iteration 5.



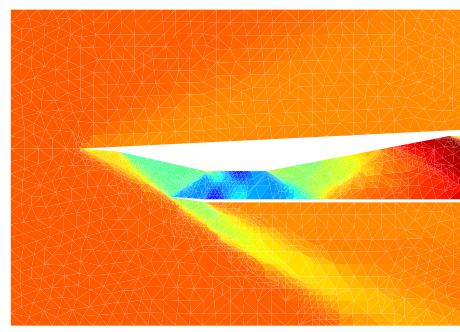
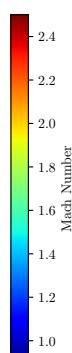
(g) Baseline Mach field.



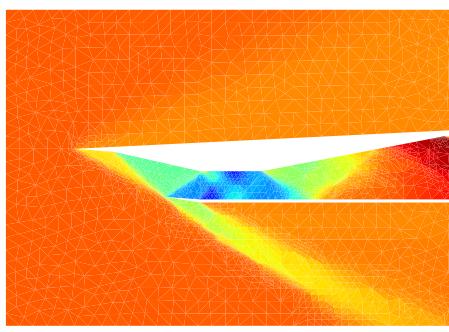
(h) Mach field, iteration 1.



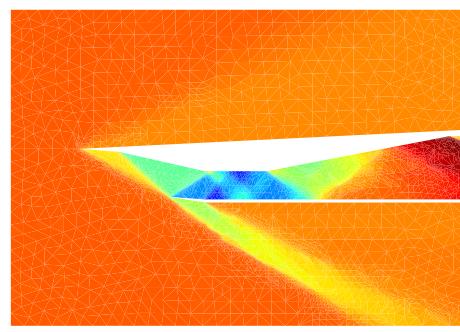
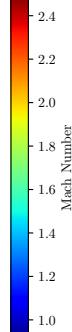
(i) Mach field, iteration 2.



(j) Mach field, iteration 3.

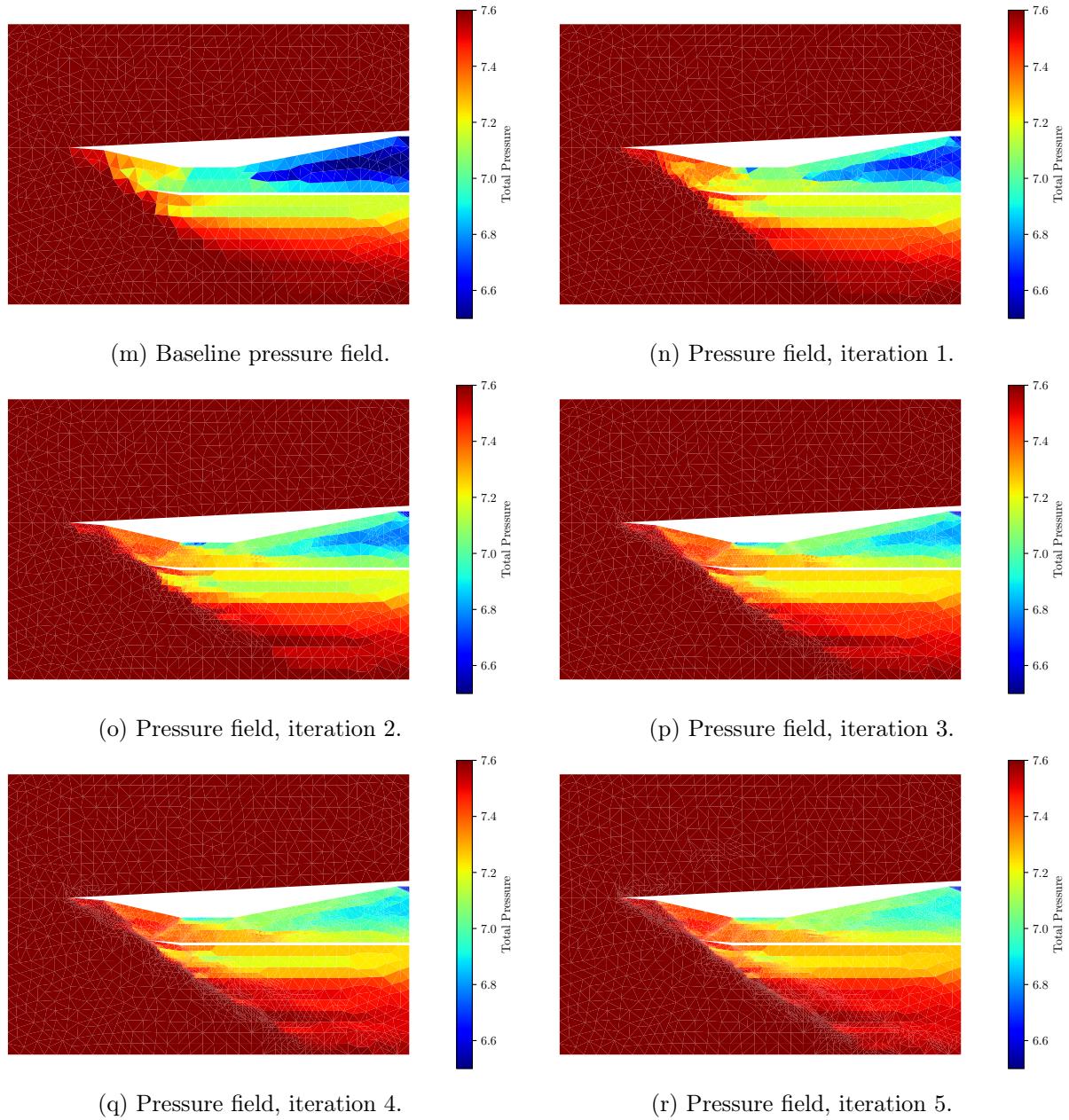


(k) Mach field, iteration 4.



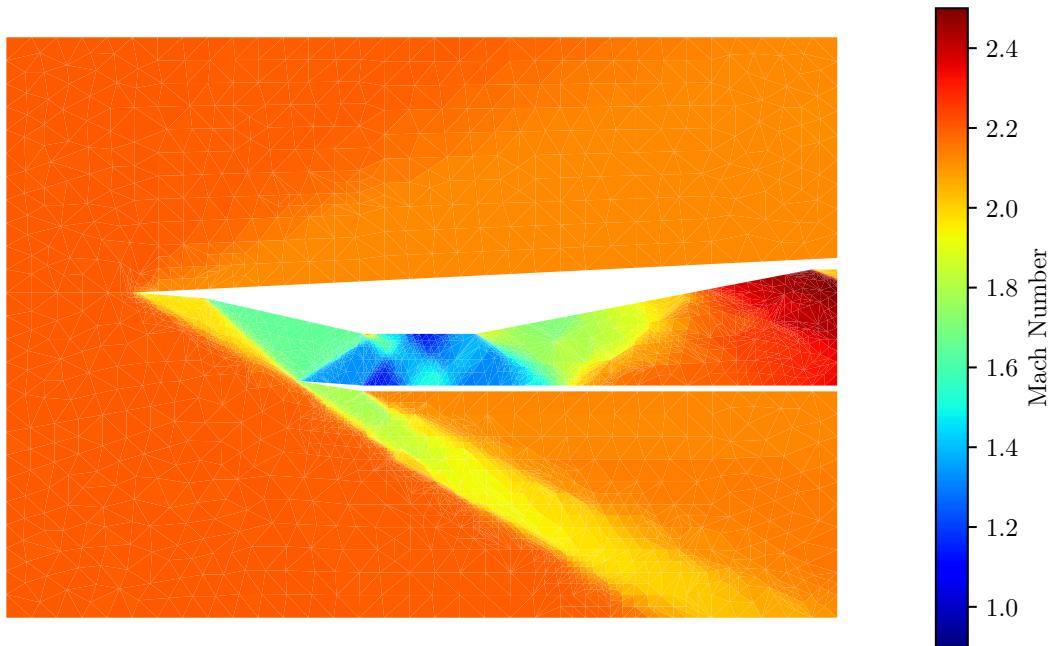
(l) Mach field, iteration 5.





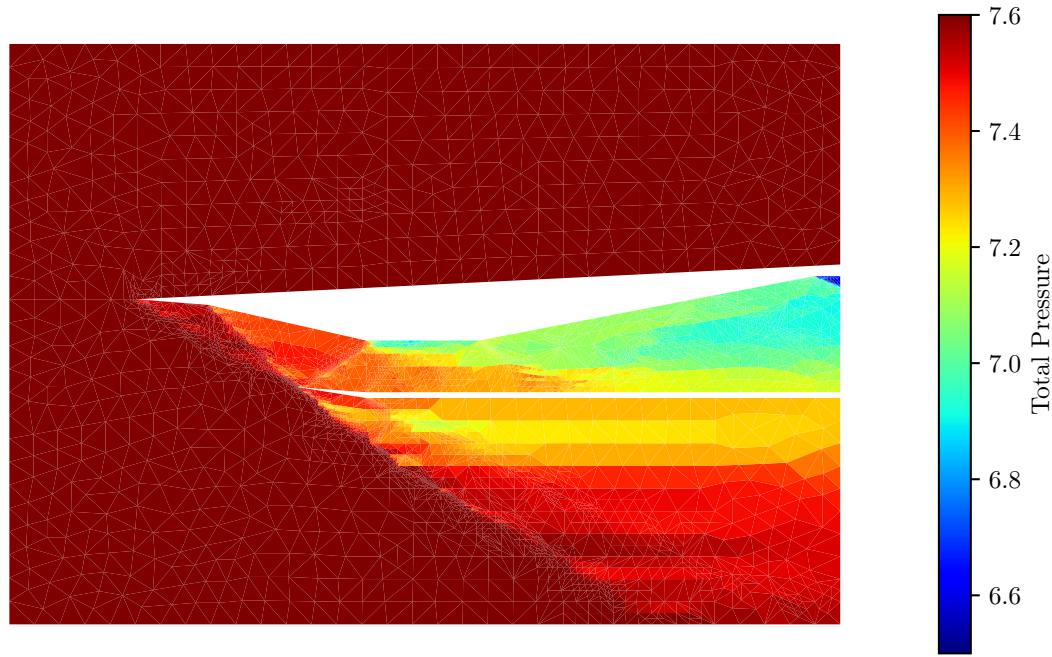
**Figure 8:** Adapted meshes versus baseline mesh.

#### 4.4.2 Adapted Mesh Field Plots



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

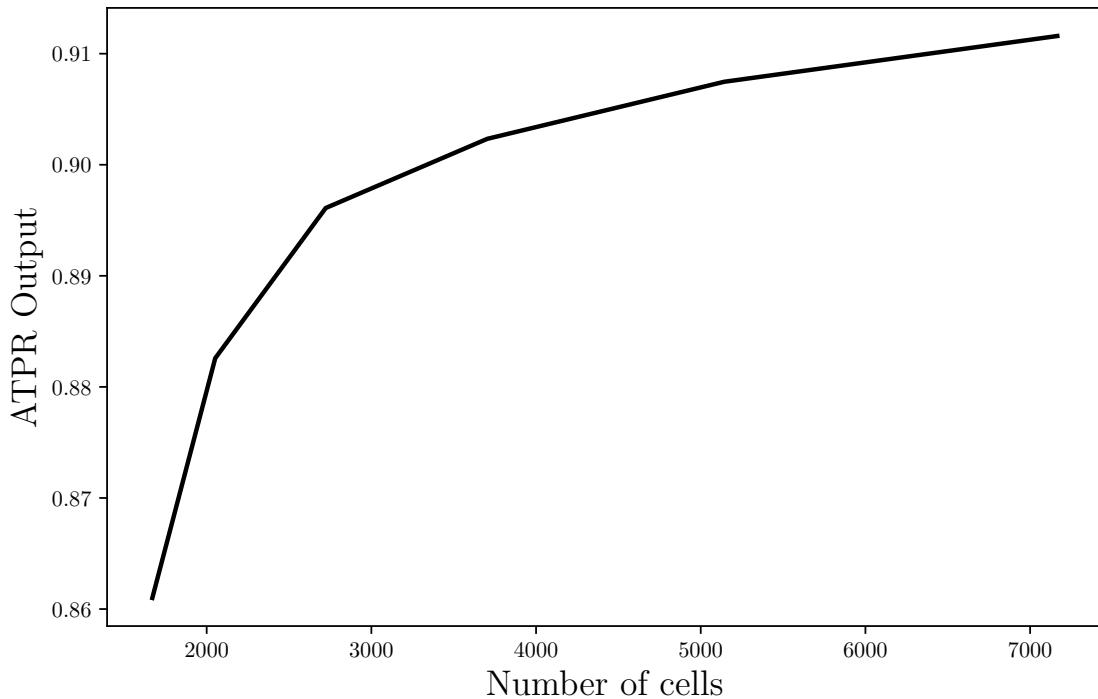
**Finest Mesh Field Plot of Mach Number** Shown above in Figure 9, is the Mach field for the most refined mesh after 5 adaptive iterations. Comparing the results from this refined mesh to that in Figure 6 shows more refinement resolution to the mach number throughout the inlet of the engine and the shock train within the engine. In this case these shock trains throughout the engine can be viewed as the differences in the Mach number in given regions indicating an oblique shock.



**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 better qualitatively how the total pressure decreases throughout the nozzle towards the exit of the engine. However, what is shown better qualitatively through this figure is that the total pressure does not decrease as much as the coarse mesh had indicated and as a result the ATPR output will increase to a more physical answer.

#### 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. As shown above, the ATPR output will converge towards an approximated solution as the cells increase the resolution of the mesh. Looking above, after the 5<sup>th</sup> adaptation the ATPR output will start to reach diminishing returns on the increases on accuracy resulting in longer run times (for me the finest mesh took approximately  $1\frac{1}{2}$  hours to run, whereas the coarsest mesh was approximately 2 minutes).

Given the plot above, the converge rate from the baseline to the first iteration is  $\mathcal{O}(2.49)$ , from first iteration to second is  $\mathcal{O}(5.37)$  which is not first-order since ATPR is a nonlinear function of the state and we are not uniformly refining the mesh.

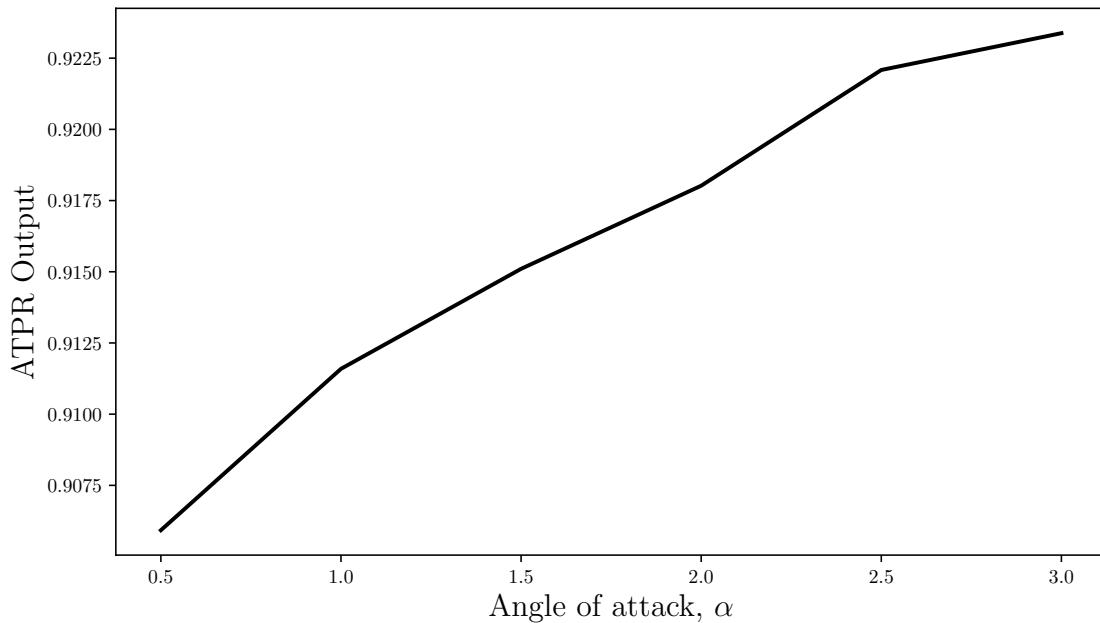
This increase in ATPR can be view qualitatively in Figures 8m-8r as the rise of the total pressure at the exit of the engine after each mesh adaptation. It is this rise in the total pressure that attributes to the rise in the ATPR output with the given number of cells.

However, increasing the number of cells greatly increases the runtime of the approximated solution even though the solutions are much more accurate. This is because to determine nodal locations it scales  $\mathcal{O}(N^2)$  resulting in an almost exponential increase in runtime resulting from the number of cells in the mesh.

## 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  $\alpha$  on the Mach field plot, the total pressure field plot, and the ATPR output.

### 4.5.1 ATPR Versus Angle of Attack



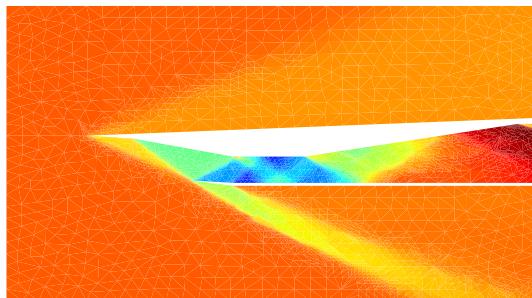
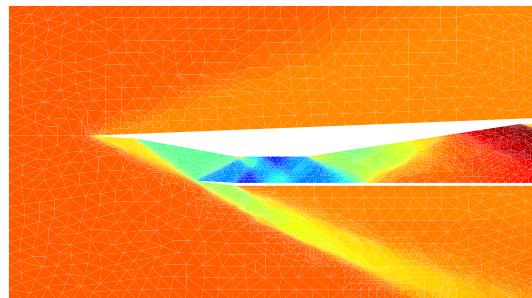
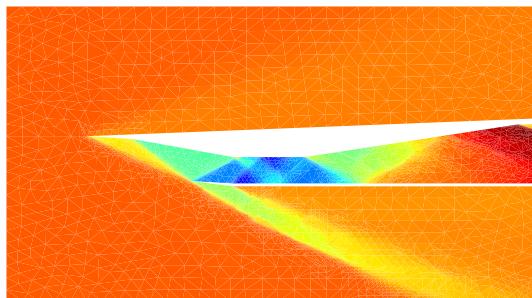
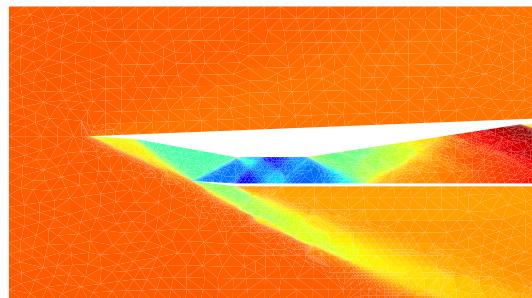
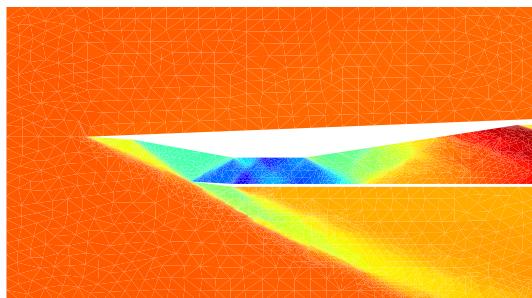
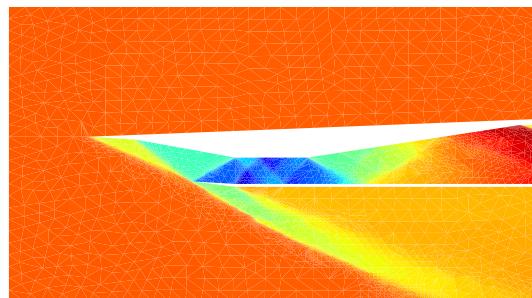
**Figure 12:** Effects of varying  $\alpha$  on the ATPR output.

Shown above in Figure 12, is the effect of varying the angle of attack on ATPR output. As shown from the Figure above, the ATPR increases with the angle of attack  $\alpha$  since at an angled attack the flow enters the engine more optimally and the oblique shocks have a less prominent effect along the interior of the engine (which can be seen in Figure 13 on the following page). Looking to Table 4, below for the tabulated values for more accuracy.

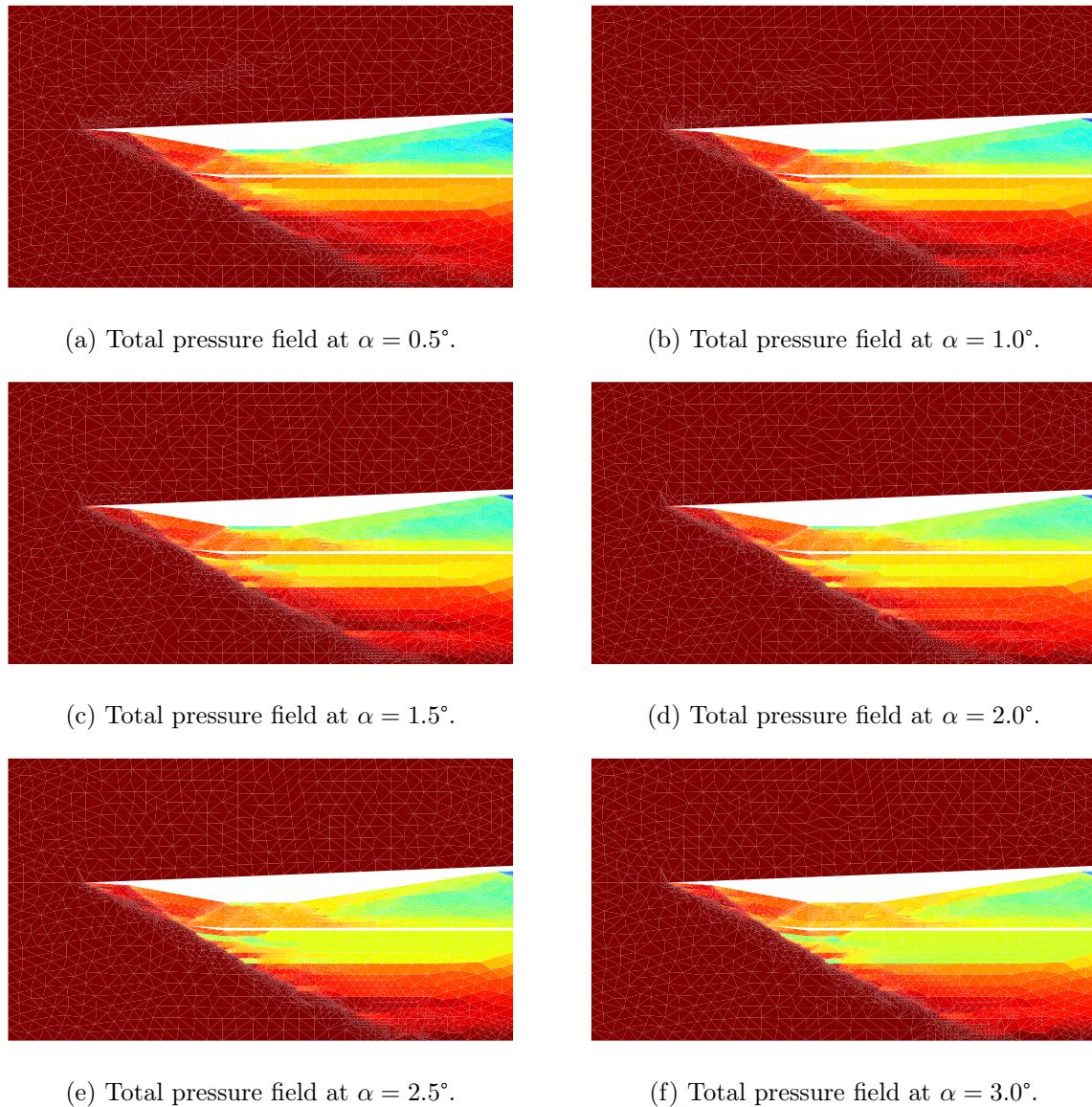
**Table 4:** ATPR versus angle of attack  $\alpha$ .

$\alpha$	ATPR
0.5°	0.906
1.0°	0.912
1.5°	0.915
2.0°	0.918
2.5°	0.922
3.0°	0.923

#### 4.5.2 Flow Fields for Varying Angle of Attacks

(a) Mach field at  $\alpha = 0.5^\circ$ .(b) Mach field at  $\alpha = 1.0^\circ$ .(c) Mach field at  $\alpha = 1.5^\circ$ .(d) Mach field at  $\alpha = 2.0^\circ$ .(e) Mach field at  $\alpha = 2.5^\circ$ .(f) Mach field at  $\alpha = 3.0^\circ$ .**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. As shown above, there is not a large difference between the angle of attack configurations, but the main differences can be seen inside the interior of the engine. Within the interior of the engine we can see the shock trains lessening in effect resulting in a lower loss in total pressure that can improve the ATPR.



**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. In this analysis, there is not a large difference (at least not as much as the Mach field), which alters the total pressure within the nozzle.

However, the most notable differences will be the total pressure below the engine and at the exit of the engine. Underneath the engine the difference is caused by a larger stagnation of pressure due to the increased angle of attack. At the exit of the engine for higher angles of attack the total pressure losses will be lower resulting in a higher total pressure than if there were more losses due to the shocks.

# Appendices

## A Python Implementation

### A.1 Main Driving Code

**Algorithm 1:** Main Driving Code

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib import rc
4 import time
5
6 # Project specific functions
7 from readgri import readgri
8 from plotmesh import plotmesh
9 from flux import RoeFlux
10 from fvm import solve
11 from adapt import adapt
12
13 plt.rc('text', usetex=True)
14 plt.rc('font', family='serif')
15
16 def getIC(alpha, mach):
17     # Constants
18     gam = 1.4; alpha = np.deg2rad(alpha)
19
20     # Initial state
21     uinf = np.transpose(np.array([1, mach*np.cos(alpha), mach*np.sin(alpha), 1/(gam
22         *(gam-1)) + mach**2/2]))
23
24     return uinf
25
26 def FVMIC(alpha, Ne):
27     # Constants and the initial state
28     alpha = np.deg2rad(alpha); Minf = 2.2; gam = 1.4
29     uinf = np.array([1, Minf*np.cos(alpha), Minf*np.sin(alpha), 1/(gam*(gam-1)) +
30         Minf**2/2])
31
32     # Loop through and form the initial state
33     u0 = np.zeros((Ne, 4))
34     for i in range(4):
35         u0[:,i] = uinf[i]
36     u0[abs(u0) < 10**-10] # Adjust for machine precision
37
38     return u0
39
40 def post_process(u):
41     gam = 1.4 # Constant
42     uvel = u[:,1]/u[:,0]; vvel = u[:,2]/u[:,0] # Velocity
43
44     q = np.sqrt(uvel**2 + vvel**2) # Magnitude of velocity
45     p = (gam-1)*(u[:,3]-0.5*u[:,0]*q**2) # Pressure
46     H = (u[:,3] + p)/u[:,0] # Enthalpy
47     c = np.sqrt((gam-1.0)*(H - 0.5*q**2)) # Speed of sound
48
49     # Calculate Mach number and the total pressure
50     mach = q/c
51     pt = p*(1 + 0.5*0.4*mach**2)**(gam/(gam-1))
52
53     def test_flux():
54         alpha = 0

```

```

55 |     ul = getIC(alpha, 0.8); ur = getIC(alpha, 0.8)
56 |     n = np.array([np.cos(np.deg2rad(alpha)), np.sin(np.deg2rad(alpha))])
57 |
58 | # Consistency Check
59 | F, analytical, FR, ls = RoeFlux(ul, ul, n); diff = abs(F - analytical)
60 | print('Roe_Flux_Tests:\nConsistency_Check\n' + 50*'-' + '\n', F, '\n', analytical)
61 |
62 | f = open('q1/consistency', 'w')
63 | f.write(r'Flux_{\rho}_{\rho u}_{\rho uv}_{\rho E}\hline')
64 | f.write(r'\hat{F}(u_1,u_1,\vec{n})%.3f%.3f%.3f\\'%(F[0],F[1],
65 |     F[2],F[3]))
66 | f.write(r'\vec{F}(\vec{U}_1)\cdot\vec{n}.3f%.3f%.3f\\'%
67 |     analytical[0],analytical[1],analytical[2],analytical[3]))
68 | f.write(r'\Delta F%.2e%.2e%.2e%(diff[0],diff[1],diff[2],diff
69 |     [3]))')
70 | f.close()
71 |
72 | # Flipping with Direction
73 | F1, FL, FR, ls = RoeFlux(ul,ur, n); Fr, FL, FR, ls = RoeFlux(ur,ul, -n); Fr *=
74 |     -1; diff = abs(F1-Fr)
75 | print('\n\nFlipping_Direction\n' + 50*'-' + '\n', F1, '\n', Fr)
76 |
77 | f = open('q1/flipped', 'w')
78 | f.write(r'Flux_{\rho}_{\rho u}_{\rho uv}_{\rho E}\hline')
79 | f.write(r'\hat{F}(u_1,u_r,\vec{n})%.3f%.3f%.3f\\'%(F1[0],F1
80 |     [1],F1[2],F1[3]))
81 | f.write(r'-F(u_r,u_1,-\vec{n})%.3f%.3f%.3f\\'%(Fr[0],Fr[1],Fr
82 |     [2],Fr[3]))
83 | f.write(r'\Delta F%.2e%.2e%.2e%(diff[0],diff[1],diff[2],diff
84 |     [3]))')
85 | f.close()
86 |
87 | # Free-stream
88 | F1, FL, FR, ls = RoeFlux(ul,ul, n); Fr, FL, FR, ls = RoeFlux(ur,ur, n); diff =
89 |     abs(F1-Fr)
90 | print('\n\nFree_Stream_Test\n' + 50*'-' + '\n', F1, '\n', Fr)
91 |
92 | # Supersonic Normal Velocity
93 | alpha = 0
94 | ul = getIC(alpha, 2.2); ur = getIC(alpha, 2.5)
95 | F, FL, FR, ls = RoeFlux(ul,ur, np.array([np.sqrt(2)/2,np.sqrt(2)/2]))
96 | print('\n\nSupersonic_Normal_Velocity\n'+50*'-'+'\n', F, '\n', FL, '\n', FR)
97 |
98 | f = open('q1/supersonic_normal', 'w')
99 | f.write(r'Flux_{\rho}_{\rho u}_{\rho uv}_{\rho E}\hline')
100 | f.write(r'\hat{F}(u_1,u_r,\vec{n})%.3f%.3f%.3f\\'%(F[0],F[1],
101 |     F[2],F[3]))
102 | f.write(r'$F_L%.3f%.3f%.3f\\'%(FL[0],FL[1],FL[2],FL[3]))
103 | f.write(r'$F_R%.3f%.3f%.3f\\'%(FR[0],FR[1],FR[2],FR[3]))
104 | f.close()
105 |
106 | def run_fvm():
107 |     mesh = readgri('mesh0.gri'); E = mesh['E']
108 |     u = FVMIC(1, E.shape[0])
109 |
110 |     start = time.time()
111 |     u, err, ATPR, V, E, BE, IE = solve(1, u, mesh); end = time.time(); print('Elapsed
112 |         Time%.2f'%(end - start))
113 |     mach, pt = post_process(u)
114 |
115 |     plt.figure(figsize=(8,5))
116 |     plt.plot(np.arange(err.shape[0]), err, lw=2, color='k')
117 |     plt.xlabel(r'Iterations', fontsize=16)
118 |     plt.ylabel(r'$L_1$ Norm', fontsize=16)
119 |     plt.xscale('log'); plt.yscale('log')
120 |     plt.savefig('q3/l1_err.pdf', bbox_inches='tight')
121 |     plt.show()
122 |
123 |     plt.figure(figsize=(8,5))
124 |     plt.plot(np.arange(ATPR.shape[0]), ATPR, lw=2, color='k')
125 |     plt.xlabel(r'Iterations', fontsize=16)

```

```

116     plt.ylabel(r'ATPR\_Output', fontsize=16)
117     plt.savefig('q3/ATPR.pdf', bbox_inches='tight')
118     plt.show()
119
120     plt.figure(figsize=(8,4.5))
121     plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=mach, vmin=0.9, vmax=2.5,
122                   cmap='jet', shading='flat')
123     plt.axis('off')
124     plt.colorbar(label='Mach_Number')
125     plt.savefig('q3/Machfield.pdf', bbox_inches='tight')
126     plt.show()
127
128     plt.figure(figsize=(8,4.5))
129     plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=pt, vmin=6.5, vmax=7.6,
130                   cmap='jet', shading='flat')
131     plt.axis('off')
132     plt.colorbar(label='Total_Pressure')
133     plt.savefig('q3/Pfield.pdf', bbox_inches='tight')
134     plt.show()
135
136 def mesh_adapt():
137     alpha = 1
138     # First iteration - IC
139     mesh = readgri('mesh0.gri'); E = mesh['E']
140     u = FVMIC(alpha, E.shape[0]); sim_start = time.time()
141
142     print('Mesh_Adaptations\n' + 25*'')
143     ATPRlin = np.array([]); Numcells = np.array([])
144     for i in range(6):
145         print('Mesh-' + str(i) + '.f' + '\n' + '%i + 15*'')
146         mesh = readgri('q4/meshs/mesh' + str(i) + '.gri'); E = mesh['E']
147         u = FVMIC(alpha, E.shape[0])
148
149         plotmesh(mesh, 'q4/mesh' + str(i) + '.pdf')
150         start = time.time()
151         u, err, ATPR, V, E, BE, IE = solve(alpha, u, mesh); end = time.time(); print(  

152             'Elapsed_Time%.2f'%(end - start))
153         mach, pt = post_process(u)
154
155         # Generate plots
156         plt.figure(figsize=(8,4.5))
157         plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=mach, vmin=0.9, vmax
158                     =2.5, cmap='jet', shading='flat')
159         plt.axis('off')
160         plt.colorbar(label='Mach_Number')
161         plt.savefig('q4/Machfield' + str(i) + '.pdf', bbox_inches='tight')
162         plt.draw()
163         plt.pause(5)
164         plt.close()
165
166         plt.figure(figsize=(8,4.5))
167         plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=pt, vmin=6.5, vmax=7.6,
168                     cmap='jet', shading='flat')
169         plt.axis('off')
170         plt.colorbar(label='Total_Pressure')
171         plt.savefig('q4/Pfield' + str(i) + '.pdf', bbox_inches='tight')
172         plt.draw()
173         plt.pause(5)
174         plt.close()
175
176         # Append the values
177         ATPRlin = np.append(ATPRlin, ATPR[ATPR.shape[0]-1])
178         Numcells = np.append(Numcells, E.shape[0])
179
180         # Adapt the mesh
181         if i != 5:
182             u, V, E, IE, BE = adapt(u, mach, V, E, IE, BE, 'q4/meshs/mesh' + str(i+1)
183             + '.gri')

```

```

181     sim_end = time.time();
182     print('nAdaptive_Mesh' + 15*'-' + '\nElapsed_Time: %.2f sec' %(sim_end -
183         sim_start))
184
185     plt.figure(figsize=(8,5))
186     plt.plot(Numcells, ATPRlin, lw=2, color='k')
187     plt.xlabel(r'Number_of_cells', fontsize=16)
188     plt.ylabel(r'ATPR_Output', fontsize=16)
189     plt.savefig('q4/ATPR.pdf', bbox_inches='tight')
190     plt.show()
191
192 def vary_alpha():
193
194     alphas = np.arange(0.5,3.5, step=0.5)
195     ATPRlin = np.array([])
196     for alpha in alphas:
197         # First iteration - IC
198         mesh = readgri('q5/meshs/' + str(int(alpha*10)) + '/mesh0.gri'); E = mesh['E']
199         u = FVMIC(alpha, E.shape[0])
200
201         print('Mesh_Adaptations\n' + 25*'-')
202         for i in range(6):
203             print('Mesh-' + '%.f' + '\n' + 15*'-')
204             mesh = readgri('q5/meshs/' + str(int(alpha*10)) + '/mesh' + str(i) + '.gri');
205             E = mesh['E']
206             u = FVMIC(alpha, E.shape[0])
207
208             u, err, ATPR, V, E, BE, IE = solve(alpha, u, mesh)
209             mach, pt = post_process(u)
210
211             # Generate Mach field plot per Alpha - Iteration
212             plt.figure(figsize=(8,4.5))
213             plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=mach, vmin=0.9,
214                           vmax=2.5, cmap='jet', shading='flat')
215             plt.axis('off')
216             plt.savefig('q5/a' + str(int(alpha*10)) + '/mach_a' + str(int(alpha*10)) +
217                         '_' + str(int(i)) + '.pdf', bbox_inches='tight')
218             plt.pause(1)
219             plt.close()
220
221             # Generate Pressure field plot per Alpha - Iteration
222             plt.figure(figsize=(8,4.5))
223             plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=pt, vmin=6.5, vmax
224                           =7.6, cmap='jet', shading='flat')
225             plt.axis('off')
226             plt.savefig('q5/a' + str(int(alpha*10)) + '/pt_a' + str(int(alpha*10)) +
227                         '_' + str(int(i)) + '.pdf', bbox_inches='tight')
228             plt.pause(1)
229             plt.close()
230
231             # Adapt the mesh
232             if i != 5:
233                 u, V, E, IE, BE = adapt(u, mach, V, E, IE, BE, 'q5/meshs/' + str(int(
234                     alpha*10)) + '/mesh' + str(i+1) + '.gri')
235             ATPRlin = np.append(ATPRlin, ATPR[ATPR.shape[0]-1])
236
237             # Field plots per alpha iteration (Post-Refinement)
238             plt.figure(figsize=(8,4.5))
239             plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=mach, vmin=0.9, vmax
240                           =2.5, cmap='jet', shading='flat')
241             plt.axis('off')
242             plt.savefig('q5/mach_a' + str(int(alpha*10)) + '.pdf', bbox_inches='tight')
243             plt.pause(1)
244             plt.close()
245
246             plt.figure(figsize=(8,4.5))
247             plt.tripcolor(V[:,0], V[:,1], triangles=E, facecolors=pt, vmin=6.5, vmax=7.6,
248                           cmap='jet', shading='flat')
249             plt.axis('off')
250             plt.savefig('q5/pt_a' + str(int(alpha*10)) + '.pdf', bbox_inches='tight')

```

```
243 |     plt.pause(1)
244 |     plt.close()
245 |
246 |
247 | # Output ATPR values to file for LaTeX table format
248 | f = open('q5/atpr_out', 'w'); output = ''
249 | for i in range(6):
250 |     output += r'%1f\degree&.%3f\\%(alphas[i], ATPRlin[i])
251 | f.write(output)
252 | f.close()
253 |
254 | plt.figure(figsize=(9,5))
255 | plt.plot(alphas, ATPRlin, lw=2, color='k')
256 | plt.xlabel(r'Angle of attack, $\alpha$', fontsize=16)
257 | plt.ylabel(r'ATPR Output', fontsize=16)
258 | plt.savefig('q5/ATPR.pdf', bbox_inches='tight')
259 | plt.show()
260 |
261 | if __name__ == "__main__":
262 |     test_flux()
263 |     run_fvm()
264 |     mesh_adapt()
265 |     vary_alpha()
```

## A.2 Finite-Volume-Element Code

**Algorithm 2:** Finite-Volume-Element Code

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numpy import linalg as LA
4 from flux import RoeFlux
5 from readgri import readgri, writegri
6
7 def getIC(alpha, Ne):
8     alpha = np.deg2rad(alpha); Minf = 2.2; gam = 1.4
9     uinf = np.array([1, Minf*np.cos(alpha), Minf*np.sin(alpha), 1/(gam*(gam-1)) +
10                   Minf**2/2])
11    u0 = np.zeros((Ne, 4))
12    for i in range(4):
13        u0[:,i] = uinf[i]
14    u0[abs(u0) < 10**-10]
15
16    return u0
17
18 def calcATPR(u0, u, alpha, V, BE):
19     gam = 1.4
20     q = np.sqrt((u0[1]/u0[0])**2 + (u0[2]/u0[0])**2)
21     Pinf = (gam-1)*(u0[3]-0.5*u0[0]*q**2)
22     Ptinf = Pinf*(1 + 0.5*(gam-1)*(2.2)**2)**(gam/(gam-1))
23
24
25     ATPR = 0; d = 0
26     for i in range(BE.shape[0]):
27         n1, n2, e1, bgroup = BE[i,:]
28         xl = V[n1,:]; xr = V[n2,:]
29         uedge = u[e1,:]
30
31         dy = xr[1] - xl[1]
32
33         if bgroup == 1: # Exit
34             uvel = uedge[1]/uedge[0]; vvel = uedge[2]/uedge[0]
35             q = np.sqrt(uvel**2 + vvel**2)
36             P = (gam-1)*(uedge[3]-0.5*uedge[0]*q**2)
37             c = np.sqrt(gam*P/uedge[0])
38             mach = q/c
39             Pt = P*(1 + 0.5*(gam-1)*mach**2)**(gam/(gam-1))
40
41             d += dy
42             ATPR += Pt*dy/Ptinf
43
44
45     ATPR *= 1/d
46     return ATPR
47
48 def getUinf(alpha):
49     gam = 1.4; mach = 2.2
50     alpha = np.deg2rad(alpha)
51     uinf = np.transpose(np.array([1, mach*np.cos(alpha), mach*np.sin(alpha), 1/(gam *
52                   *(gam-1)) + mach**2/2]))
53
54     return uinf
55
56 def solve(alpha, u0, mesh):
57     V = mesh['V']; E = mesh['E']; BE = mesh['BE']; IE = mesh['IE']
58
59     uinf = getUinf(alpha)
60
61     u = u0.copy(); ATPR = np.array([calcATPR(uinf,u,alpha,V,BE)])
62     R = np.zeros((E.shape[0], 4)); dta = R.copy(); err = np.array([1]); itr = 0
63
64     while err[err.shape[0]-1] > 10**(-5):
65         R *= 0; dta *= 0
66         for i in range(IE.shape[0]):
```

```

66     n1, n2, e1, e2 = IE[i,:]
67     xl = V[n1,:]; xr = V[n2,:]
68     ul = u[e1,:]; ur = u[e2,:]
69
70     dx = xr - xl; deltal = LA.norm(dx)
71     nhat = np.array([dx[1], -dx[0]])/deltal
72     F, FL, FR, ls = RoeFlux(ul, ur, nhat)
73     R[e1,:] += F*deltal; R[e2,:] -= F*deltal
74     dta[e1,:] += ls*deltal; dta[e2,:] += ls*deltal
75
76     for i in range(BE.shape[0]):
77         n1, n2, e1, bgroup = BE[i,:]
78         xl = V[n1,:]; xr = V[n2,:]
79         uedge = u[e1,:]
80
81         dx = xr - xl; deltal = LA.norm(dx)
82         nhat = np.array([dx[1], -dx[0]])/deltal
83
84         if bgroup == 0: # Engine - Invscid
85             vp = np.array([uedge[1], uedge[2]])/uedge[0]
86             vb = vp - np.dot(vp, nhat)*nhat
87             pb = 0.4*(uedge[3] - 0.5*uedge[0]*(vb[0]**2 + vb[1]**2))
88             ignore, FL, FR, ls = RoeFlux(uedge, uinf, nhat)
89
90             F = pb*np.array([0, nhat[0], nhat[1], 0])
91         elif bgroup == 1 or bgroup == 2: # Exit/Outflow - Supersonic Outflow
92             F, FL, FR, ls = RoeFlux(uedge, uedge, nhat)
93         elif bgroup == 3: # Inflow
94             F, FL, FR, ls = RoeFlux(uedge, uinf, nhat)
95
96         R[e1,:] += F*deltal
97         dta[e1,:] += ls*deltal
98
99         dta = 2/dta
100        u -= np.multiply(dta, R)
101        err = np.append(err, sum(sum(abs(R))))
102
103        ATPR = np.append(ATPR, calcATPR(uinf, u, alpha, V, BE))
104        print('Iteration: %d, \tError: %.3e, \tATPR: %.3f' %(itr, err.shape[0]-1,
105          ATPR[ATPR.shape[0]-2])); itr += 1
106
return u, err[1:], ATPR, V, E, BE, IE

```

### A.3 Roe Flux Python Implementation

**Algorithm 3:** Roe Flux Implementation

```

1 import numpy as np
2 from numpy import linalg as LA
3
4 def RoeFlux(Ul, Ur, n):
5     gam = 1.4
6
7     # Left side arguments
8     rhol = Ul[0]; ul = Ul[1]/rhol; vl = Ul[2]/rhol; rhoEl = Ul[3]
9     pl = (gam-1)*(rhoEl-0.5*rhol*(ul**2 + vl**2))
10    Hl = (rhoEl + pl)/rhol
11
12    # Right side arguments
13    rhor = Ur[0]; ur = Ur[1]/rhor; vr = Ur[2]/rhor; rhoEr = Ur[3]
14    pr = (gam-1)*(rhoEr-0.5*rhor*(ur**2 + vr**2))
15    Hr = (rhoEr + pr)/rhor
16
17    # Left and Right side fluxes
18    FL = np.array([np.dot([Ul[1],Ul[2]], n), np.dot([Ul[1]*ul+pl, Ul[2]*vl],n), np.
19                  dot([Ul[1]*vl, Ul[2]*vl+pl],n), Hl*np.dot([Ul[1],Ul[2]],n)])
20    FR = np.array([np.dot([Ur[1],Ur[2]], n), np.dot([Ur[1]*ur+pr, Ur[2]*vr],n), np.
21                  dot([Ur[1]*vr, Ur[2]*vr+pr],n), Hr*np.dot([Ur[1],Ur[2]],n)])
22
23    # Roe-Averages
24    RHS, ls = ROE_Avg(ul,vl,rhol,Hl,rhoEl, ur,vr,rhor,Hr,rhoEr, n)
25    F = 0.5*(FL + FR) - 0.5*RHS
26
27    return F, FL, FR, ls
28
29 def ROE_Avg(ul,vl,rhol,Hl,rhoEl, ur,vr,rhor,Hr,rhoEr, n):
30     gam = 1.4
31     vell = np.array([ul, vl]); velr = np.array([ur, vr])
32
33     # Calculating Roe average
34     v = (np.sqrt(rhol)*vell + np.sqrt(rhor)*velr)/(np.sqrt(rhol) + np.sqrt(rhor))
35     H = (np.sqrt(rhol)*Hl + np.sqrt(rhor)*Hr)/(np.sqrt(rhol) + np.sqrt(rhor))
36
37     # Calculating eigenvalues
38     q = LA.norm(v)
39     c = np.sqrt((gam-1.0)*(H - 0.5*q**2))
40     u = np.dot(v, n)
41     ls = abs(np.array([u+c, u-c, u]))
42
43     # Apply the entropy fix
44     ls[abs(ls) < 0.1*c] = ((0.1*c)**2 + ls[abs(ls) < 0.1*c]**2)/(2*0.1*c)
45
46     delrho = rhor - rhol; delmo = np.array([rhor*ur - rhol*ul, rhor*vr - rhol*vl]);
47     dele = rhoEr - rhoEl
48     s1 = 0.5*(abs(ls[0]) + abs(ls[1])); s2 = 0.5*(abs(ls[0]) - abs(ls[1]))
49     G1 = (gam-1.0)*(0.5*q**2*delrho - np.dot(v, delmo) + dele); G2 = -1.0*u*delrho +
50           np.dot(delmo, n)
51     C1 = G1*(c**-2)*(s1 - abs(ls[2])) + G2*(c**-1)*s2; C2 = G1*(c**-1)*s2 + (s1 -
52           abs(ls[2]))*G2
53
54     RHS = np.array([ls[2]*delrho+C1, ls[2]*delmo[0]+C1*v[0]+C2*n[0], ls[2]*delmo[1]+
55                   C1*v[1]+C2*n[1], ls[2]*dele+C1*H+C2*u])
56
57     return RHS, max(ls)

```

## A.4 Adaptive Mesh Python Implementation

**Algorithm 4:** Adaptive Mesh Implementation

```

1 import numpy as np
2 from numpy import linalg as LA
3 from readgri import writegri
4 from edgehash import edgehash
5 import matplotlib.pyplot as plt
6
7 def mach_perp(u, nhat):
8     uvel = u[1]/u[0]; v = u[2]/u[0]      # Calculate the velocity
9     q = np.dot(np.array([uvel, v]), nhat) # Determine the perpendicular speed
10    P = (1.4 - 1)*(u[3] - 0.5*u[0]*q**2) # Calculate pressure
11    H = (u[3] + P)/u[0]                  # Calculate enthalpy
12    c = np.sqrt(0.4*(H - 0.5*q**2))    # Calculate speed of sound
13    mach = q/c                         # Calculate the Mach number
14
15    return mach
16
17 def check_vert(Vvec, x):
18     check = True
19     # Loop over the vertices
20     for i in range(Vvec.shape[0]):
21         # If this vertex exists return False
22         if '%.5f'%x[0] == '%.5f'%Vvec[i,0] and '%.5f'%x[1] == '%.5f'%Vvec[i,1]:
23             check = False
24             break
25
26     return check
27
28 def isCCW(a, b, c):
29     # Princeton implementation of CCW logical
30     cross_val = (b[0] - a[0])*(c[1] - a[1]) - (c[0] - a[0])*(b[1] - a[1])
31
32     # Conditional statements
33     if cross_val > 0:
34         cross_val = 1
35     elif cross_val < 0:
36         cross_val = -1
37     else:
38         cross_val = 0
39
40     return cross_val
41
42 def vert_ind(Vvec, x):
43     check = False; ind = np.array([])
44     # Loop over the vertices
45     for i in range(Vvec.shape[0]):
46         # If this vertex exists return False
47
48         if '%.5f'%x[0] == '%.5f'%Vvec[i,0] and '%.5f'%x[1] == '%.5f'%Vvec[i,1]:
49             check = True; ind = np.append(ind, np.array([i]))
50
51     return check, ind
52
53 def genflags(u, mach, V, E, IE, BE):
54
55     # Pre-allocate flag array
56     flags = np.zeros((IE.shape[0] + BE.shape[0], 2)); flags[:,0] = np.arange(flags.
57     shape[0]); k = 0
58
59     # Iterate over the interior edges
60     for i in range(IE.shape[0]):
61         n1, n2, e1, e2 = IE[i,:]           # Nodes and elements from interior edge
62         xl = V[n1,:]; xr = V[n2,:]
63         machl = mach[e1]; machr = mach[e2] # Mach numbers at each element
64         dx = xr - xl; deltal = LA.norm(dx) # Determine the length of the edge
65         eps = abs(machr - machl)*deltal # Calculate the error
66
67         flags[k,1] += eps; k += 1          # Add the edge error

```

```

67
68     # Iterate over the boundary edges
69     for i in range(BE.shape[0]):
70         n1, n2, e1, bgroup = BE[i,:]
71         if bgroup == 0: # Engine
72             xl = V[n1,:]; xr = V[n2,:]
73             uedge = u[e1,:]
74             dx = xr - xl; deltal = LA.norm(dx) # Determine the length of the edge
75             nhat = np.array([dx[1], -dx[0]])/deltal # Determine the normal off the
76                 boundary edge
77             machperp = mach_perp(uedge, nhat) # Calculate the perpendicular Mach
78                 number
79             eps = abs(machperp)*deltal # Calculate error
80
81             flags[k,1] += eps # Add the edge error
82             k += 1
83
84     # Sort from largest to smallest errors
85     flags = flags[:,1].argsort(); flags = np.flipud(flags)
86     # Remove all outliers to be refined
87     ind = int(np.ceil(flags.shape[0] * 0.03))
88     flags[ind:(flags.shape[0]-1),1] = 0
89
90     # Sort the errors increasing the edge number to iterate
91     flags = flags[:,0].argsort()
92
93     return flags
94
95 def genV(flags, V, E, IE, BE):
96     Vcopy = V.copy(); k = 0
97     for i in range(IE.shape[0]):
98         err = flags[k,1]
99         if err > 0:
100             ig, ig, e1, e2 = IE[i,:]
101             for j in np.array([e1,e2]):
102                 n1, n2, n3 = E[j,:]
103                 x1 = V[n1,:]; x2 = V[n2,:]; x3 = V[n3,:]
104
105                 # Conditionals to prevent duplicate nodes
106                 if check_vert(Vcopy, (x2-x1)/2 +x1):
107                     Vcopy = np.append(Vcopy, np.array([(x2-x1)/2 +x1]), axis=0)
108                 if check_vert(Vcopy, (x3-x1)/2 +x1):
109                     Vcopy = np.append(Vcopy, np.array([(x3-x1)/2 +x1]), axis=0)
110                 if check_vert(Vcopy, (x3-x2)/2 +x2):
111                     Vcopy = np.append(Vcopy, np.array([(x3-x2)/2 +x2]), axis=0)
112
113             err = flags[k,1]
114             if err > 0:
115                 ig, ig, e1, ig = BE[i,:]
116                 n1, n2, n3 = E[e1,:]
117                 x1 = V[n1,:]; x2 = V[n2,:]; x3 = V[n3,:]
118
119                 # Conditionals to prevent duplicate nodes
120                 if check_vert(Vcopy, (x2-x1)/2 +x1):
121                     Vcopy = np.append(Vcopy, np.array([(x2-x1)/2 +x1]), axis=0)
122                 if check_vert(Vcopy, (x3-x1)/2 +x1):
123                     Vcopy = np.append(Vcopy, np.array([(x3-x1)/2 +x1]), axis=0)
124                 if check_vert(Vcopy, (x3-x2)/2 +x2):
125                     Vcopy = np.append(Vcopy, np.array([(x3-x2)/2 +x2]), axis=0)
126
127             k += 1
128
129     return Vcopy
130
131 def genUE(u, Vcopy, V, E, IE, BE):
132     Ecopys = E.copy(); Ucopy = u.copy()
133     for i in range(Ecopys.shape[0]):
134         # Grab the node values of each given element
135         n1, n2, n3 = Ecopys[i,:]

```

```

136     vals = np.array([(x2-x1)/2 +x1, (x3-x1)/2 +x1, (x3-x2)/2 +x2])
137
138     # Generate nodes for each element
139     nodes = np.array([])
140     for k in vals:
141         check, ind = vert_ind(Vcopy, k)
142         if check:
143             nodes = np.append(nodes, ind)
144
145     # If three flags have been flagged
146     if nodes.shape[0] == 3:
147         # Ensure that the nodes are CCW
148         if isCCW(Vcopy[int(nodes[0]),:], Vcopy[int(nodes[1]),:], Vcopy[int(nodes[2]),:]) != 1:
149             nodes = np.flip(nodes)
150
151     nodeint = np.array([n1, n2, n3])
152     if isCCW(Vcopy[int(nodeint[0]),:], Vcopy[int(nodeint[1]),:], Vcopy[int(nodeint[2]),:]) != 1:
153         nodeint = np.flip(nodeint)
154
155     # Loop through the nodes
156     for k in range(3):
157         # Start at the nodes N1 -> N2 for consistency
158         if '%.5f'%Vcopy[int(nodes[k]),0] == '%.5f'%vals[0,0] and '%.5f'%Vcopy[int(nodes[k]),1] == '%.5f'%vals[0,1]:
159             Ecopy[i,:] = np.array([n1, nodes[k], nodes[(k+2)%3]]) # Replace
160                         the ith element with new element
161
162         # Start the indices based from generalized case
163         ind1 = np.array([nodes[k], n2, nodes[(k+1)%3]])
164         ind2 = np.array([nodes[k], nodes[(k+1)%3], nodes[(k+2)%3]])
165         ind3 = np.array([nodes[(k+1)%3], nodes[(k+2)%3], n3])
166
167         # Ensure that the nodes are CCW
168         if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),:]) != 1:
169             ind1 = np.flip(ind1)
170         if isCCW(Vcopy[int(ind2[0]),:], Vcopy[int(ind2[1]),:], Vcopy[int(ind2[2]),:]) != 1:
171             ind2 = np.flip(ind2)
172         if isCCW(Vcopy[int(ind3[0]),:], Vcopy[int(ind3[1]),:], Vcopy[int(ind3[2]),:]) != 1:
173             ind3 = np.flip(ind3)
174
175         # Append new elements
176         Ecopy = np.append(Ecopy, np.transpose(np.array([[ind1[0]], [ind1[1]], [ind1[2]]])), axis=0)
177         Ecopy = np.append(Ecopy, np.transpose(np.array([[ind2[0]], [ind2[1]], [ind2[2]]])), axis=0)
178         Ecopy = np.append(Ecopy, np.transpose(np.array([[ind3[0]], [ind3[1]], [ind3[2]]])), axis=0)
179
180         # Append values of U to the updated U for initial starting
181         # condition
182         for l in range(3):
183             Ucopy = np.append(Ucopy, np.transpose(np.array([[u[i,0]], [u[i,1]], [u[i,2]], [u[i,3]]])), axis=0)
184             break
185
186     # If two edges have been flagged
187     elif nodes.shape[0] == 2:
188         node_ind = np.array([n1, n2, n3])
189
190         # Ensure CCW order
191         if isCCW(Vcopy[int(node_ind[0]),:], Vcopy[int(node_ind[1]),:], Vcopy[int(node_ind[2]),:]) != 1:
192             node_ind = np.flip(node_ind)
193
194         # Ensure CCW order
195         ccw_count = 0

```

```

194     for k in range(3):
195         if isCCW(Vcopy[int(node_ind[k)],:], Vcopy[int(nodes[0]),:], Vcopy[int(
196             nodes[1],:)] != 1:
197             ccw_count += 1
198         if ccw_count == 2:
199             nodes = np.flip(nodes)
200
201         # Determine which node can be omitted
202         for k in range(3):
203             xn1 = Vcopy[int(node_ind[k)],:]; xn2 = Vcopy[int(node_ind[(k+1)%3],
204                 :, :]; xn3 = Vcopy[int(node_ind[(k-1)%3]),:]
205             test1 = (xn2-xn1)/2 + xn1; test2 = (xn1-xn3)/2 + xn3
206
207             # Conditional to determine the starting nodes for triangle
208             if '%.5f'%test1[0] == '%.5f'%Vcopy[int(nodes[1]),0] and '%.5f'%test1
209                 [1] == '%.5f'%Vcopy[int(nodes[1]),1] and '%.5f'%test2[0] == '%.5f'
210                 '%Vcopy[int(nodes[0]),0] and '%.5f'%test2[1] == '%.5f'%Vcopy[int(
211                     nodes[0]),1]:
212                 ind1 = np.array([node_ind[k], nodes[1], nodes[0]])
213                 newnodes = np.array([node_ind[(k+1)%3], node_ind[(k+2)%3]])
214
215             # Initialize values for test
216             vn1 = Vcopy[int(nodes[0]),:]; vn2 = Vcopy[int(nodes[1]),:]
217             xn1 = Vcopy[int(newnodes[0]),:]; xn2 = Vcopy[int(newnodes[1]),:]
218
219             # Determine the angle of the first corner
220             temp1 = (vn1-xn1)/2 + xn1; temp2 = (vn2-xn1)/2 + xn1
221             theta1 = np.arccos(np.dot(temp1, temp2)/(LA.norm(temp1)*LA.norm(temp2)))
222
223             # Determine the angle of the second corner
224             temp1 = (vn1-xn2)/2 + xn2; temp2 = (vn2-xn2)/2 + xn2
225             theta2 = np.arccos(np.dot(temp1, temp2)/(LA.norm(temp1)*LA.norm(temp2)))
226
227             # Conditional to ensure no edge overlaps
228             if theta1 >= theta2:
229                 # Re-arrangement of indices to ensure connectivity
230                 ind2 = np.array([newnodes[1], nodes[0], nodes[1]])
231                 ind3 = np.array([newnodes[0], newnodes[1], nodes[1]])
232             else:
233                 # Re-arrangement of indices to ensure connectivity
234                 ind2 = np.array([newnodes[0], nodes[0], nodes[1]])
235                 ind3 = np.array([newnodes[0], newnodes[1], nodes[0]])
236
237             # Ensure that they are CCW
238             if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),
239                 :, :]) != 1:
240                 ind1 = np.flip(ind1)
241             if isCCW(Vcopy[int(ind2[0]),:], Vcopy[int(ind2[1]),:], Vcopy[int(ind2[2]),
242                 :, :]) != 1:
243                 ind2 = np.flip(ind2)
244             if isCCW(Vcopy[int(ind3[0]),:], Vcopy[int(ind3[1]),:], Vcopy[int(ind3[2]),
245                 :, :]) != 1:
246                 ind3 = np.flip(ind3)
247
248             # Overwrite E, and append to E
249             Ecopy[i,:] = np.array([ind1[0], ind1[1], ind1[2]])
250
251             Ecopy = np.append(Ecopy, np.transpose(np.array([[ind2[0]], [ind2[1]], [
252                 ind2[2]]])), axis=0)
253             Ecopy = np.append(Ecopy, np.transpose(np.array([[ind3[0]], [ind3[1]], [
254                 ind3[2]]])), axis=0)
255
256             # Append to U for initial starting guess
257             for k in range(2):
258                 Ucopy = np.append(Ucopy, np.transpose(np.array([[u[i,0]], [u[i,1]],
259                     [i,2]], [u[i,3]]))), axis=0)
260
261             # If ones edges has been flagged
262             elif nodes.shape[0] == 1:
263                 # Determine the starting node
264                 for k in range(3):

```

```

254 |         if '%.5f'%vals[k,0] == '%=.5f'%Vcopy[int(nodes[0]),0] and '%.5f'%vals[
255 |             k,1] == '%.5f'%Vcopy[int(nodes[0]),1]:
256 |             # Rearrange the nodes according to the node value
257 |             if k == 0:
258 |                 ind1 = np.array([n1, nodes[0], n3])
259 |                 ind2 = np.array([n2, n3, nodes[0]])
260 |             elif k == 1:
261 |                 ind1 = np.array([n1, nodes[0], n2])
262 |                 ind2 = np.array([n3, n2, nodes[0]])
263 |             elif k == 2:
264 |                 ind1 = np.array([n2, nodes[0], n1])
265 |                 ind2 = np.array([n3, n1, nodes[0]])
266 |
267 |             # Ensure CCW orientation
268 |             if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),
269 |                 :, :]) != 1:
270 |                 ind1 = np.flip(ind1)
271 |             if isCCW(Vcopy[int(ind2[0]),:], Vcopy[int(ind2[1]),:], Vcopy[int(ind2[2]),
272 |                 :, :]) != 1:
273 |                 ind2 = np.flip(ind2)
274 |
275 |             # Overwrite and append E
276 |             Ecopy[i,:] = np.array([ind1[0], ind1[1], ind1[2]])
277 |             Ecopy = np.append(Ecopy, np.transpose(np.array([[ind2[0]], [ind2[1]], [ind2[2]]])), axis=0)
278 |
279 |             # Append to U for initial start
280 |             Ucopy = np.append(Ucopy, np.transpose(np.array([[u[i,0]], [u[i,1]], [u[i,
281 |                 2]], [u[i,3]]])), axis=0)
282 |
283 |             # Double check to make sure CCW orientation
284 |             for i in range(Ecopy.shape[0]):
285 |                 n1, n2, n3 = Ecopy[i,:]
286 |                 ind1 = np.array([n1, n2, n3])
287 |                 if isCCW(Vcopy[int(ind1[0]),:], Vcopy[int(ind1[1]),:], Vcopy[int(ind1[2]),:]) != 1:
288 |                     Ecopy[i,:] = np.array([Ecopy[i,2], Ecopy[i,1], Ecopy[i,0]])
289 |
290 |             # Return E (as an integer array)
291 |             Ecopy = Ecopy.astype(int)
292 |
293 |             return Ucopy, Ecopy
294 |
295 |     def genB(u, V, Vcopy, BE):
296 |         Bcopy = BE.copy()
297 |         for i in range(Bcopy.shape[0]):
298 |             # Node locations of boundary edges
299 |             n1, n2, e1, bgroup = BE[i,:]
300 |             xl = V[n1,:]; xr = V[n2,:]
301 |
302 |             # Call function to determine if the vertex exists
303 |             check, ind = vert_ind(Vcopy, (xl+xr)/2)
304 |             if check:
305 |                 # If this vertex exists re-write B
306 |                 Bcopy[i,:] = np.array([n1, ind[0], e1, bgroup])
307 |                 Bcopy = np.append(Bcopy, np.transpose(np.array([[ind[0]], [n2], [Bcopy.
308 |                     shape[0]+1], [bgroup]])), axis=0)
309 |
310 |             # Re-arrange B for input to edgehash()
311 |             B0 = np.array([-1,-1]); B1 = B0.copy(); B2 = B0.copy(); B3 = B0.copy()
312 |             for i in range(Bcopy.shape[0]):
313 |                 # Node values
314 |                 n1, n2, e, bname = Bcopy[i,:]
315 |                 # Given the values of Bname append to the corresponding group
316 |                 if bname == 0:
317 |                     B0 = np.append(B0, np.transpose(np.array([[n1], [n2]]))), axis=0)

```

```

318     B3 = np.append(B3, np.transpose(np.array([[n1], [n2]])), axis=0)
319
320     # Output B#'s to B for input to edgehash()
321     B0 = B0[1:,:]; B1 = B1[1:,:]; B2 = B2[1:,:]; B3 = B3[1:,:];
322     B = [B0.astype(int), B1.astype(int), B2.astype(int), B3.astype(int)]
323
324     return B
325
326 def adapt(u, mach, V, E, IE, BE, filepath):
327
328     flags = genflags(u, mach, V, E, IE, BE)      # Flag edges along the interior and
329     exterior
330     Vcopy = genV(flags, V, E, IE, BE)            # With the flags determine the nodes
331     on the elements to split
332     Ucopy, Ecop = genUE(u, Vcopy, V, E, IE, BE) # Determine the new Elements and
333     with them the new U
334     B = genB(u, V, Vcopy, BE)                   # With the old boundary edges
335     determine which are on the edges
336     IECopy, BECopy = edgehash(Ecop, B)
337
338
339     # Prepare for input to writegri
340     Mesh = {'V':Vcopy, 'E':Ecop, 'IE':IEcopy, 'BE':BECopy, 'Bname':['Engine', 'Exit',
341     ', 'Outflow', 'Inflow']} }
342     writegri(Mesh, filepath)
343
344     return Ucopy, Vcopy, Ecop, IECopy, BECopy

```

## B Additional Supporting Code

**Algorithm 5:** Python Edge Hash

```

1 import numpy as np
2 from scipy import sparse
3
4
5 #-----
6 # Identifies interior and boundary edges given element-to-node
7 # IE contains (n1, n2, elem1, elem2) for each interior edge
8 # BE contains (n1, n2, elem) for each boundary edge
9 def edgehash(E, B):
10    Ne = E.shape[0]; Nn = np.amax(E)+1
11    H = sparse.lil_matrix((Nn, Nn), dtype=np.int)
12    IE = np.zeros([int(np.ceil(Ne*1.5)),4], dtype=np.int)
13    ni = 0
14    for e in range(Ne):
15        for i in range(3):
16            n1, n2 = E[e,i], E[e,(i+1)%3]
17            if (H[n2,n1] == 0):
18                H[n1,n2] = e+1
19            else:
20                eR = H[n2,n1]-1
21                IE[ni,:] = n1, n2, e, eR
22                H[n2,n1] = 0
23                ni += 1
24    IE = IE[0:ni,:]
25    # boundaries
26    nb0 = nb = 0
27    for g in range(len(B)):
28        nb0 += B[g].shape[0]
29    BE = np.zeros([nb0,4], dtype=np.int)
30    for g in range(len(B)):
31        Bi = B[g]
32        for b in range(Bi.shape[0]):
33            n1, n2 = Bi[b,0], Bi[b,1]
34            if (H[n1,n2] == 0): n1,n2 = n2,n1
35            BE[nb,:] = n1, n2, H[n1,n2]-1, g
36            nb += 1
37    return IE, BE

```

**Algorithm 6:** Python Plot Mesh

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from readgri import readgri
4
5 #-
6 def plotmesh(Mesh, fname):
7     V = Mesh['V']; E = Mesh['E']; BE = Mesh['BE']
8
9     f = plt.figure(figsize=(12,12))
10    plt.tripplot(V[:,0], V[:,1], E, 'k-')
11    for i in range(BE.shape[0]):
12        plt.plot(V[BE[i,0:2],0],V[BE[i,0:2],1], '--', linewidth=2, color='black')
13    plt.axis('equal'); plt.axis('off')
14    f.tight_layout();
15    plt.savefig(fname, bbox_inches='tight')
16    plt.close()
```

## References

- [1] K. Fidkowski, “Computational fluid dynamics,” September 2020.
- [2] Gryphon, “Roe flux differencing scheme: The approximate riemann problem.”
- [3] R. Sedgewik and K. Wayne, “Geometric primitives,” December 2016.