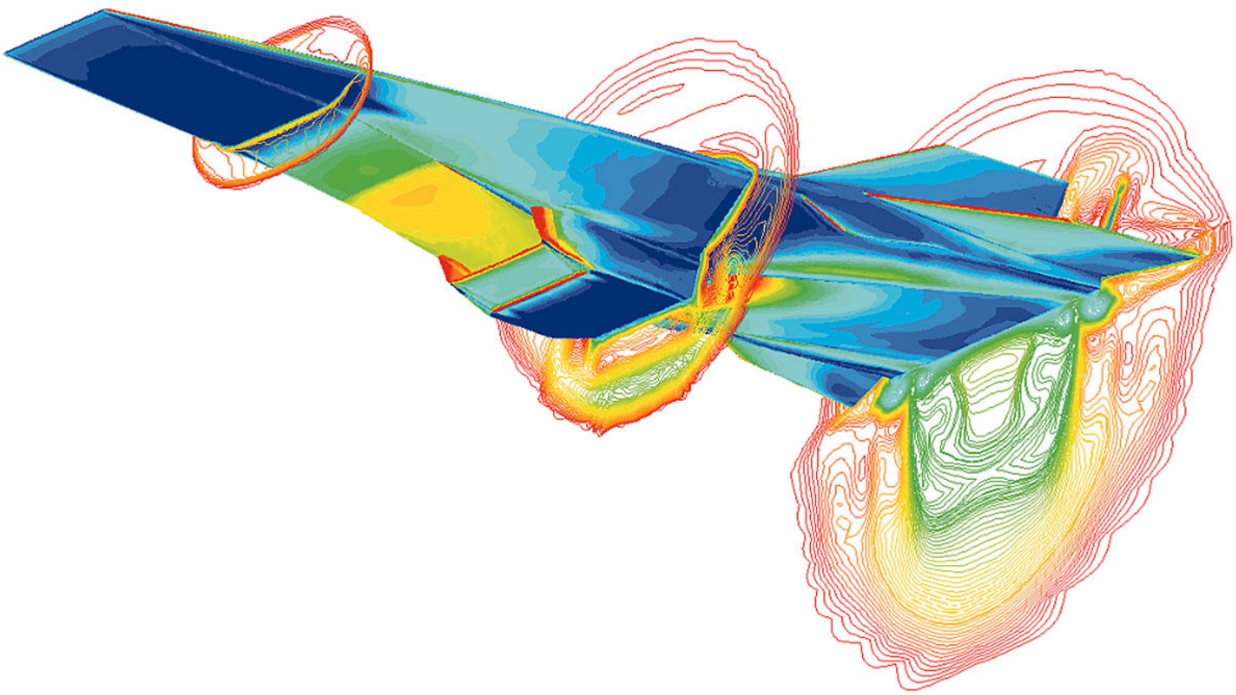
Project 2: Supersonic Engine Analysis

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

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



NASA X-43 Hypersonic Airplane

Contents

List of Figures

List of Equations

List of Tables

List of Algorithms

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.

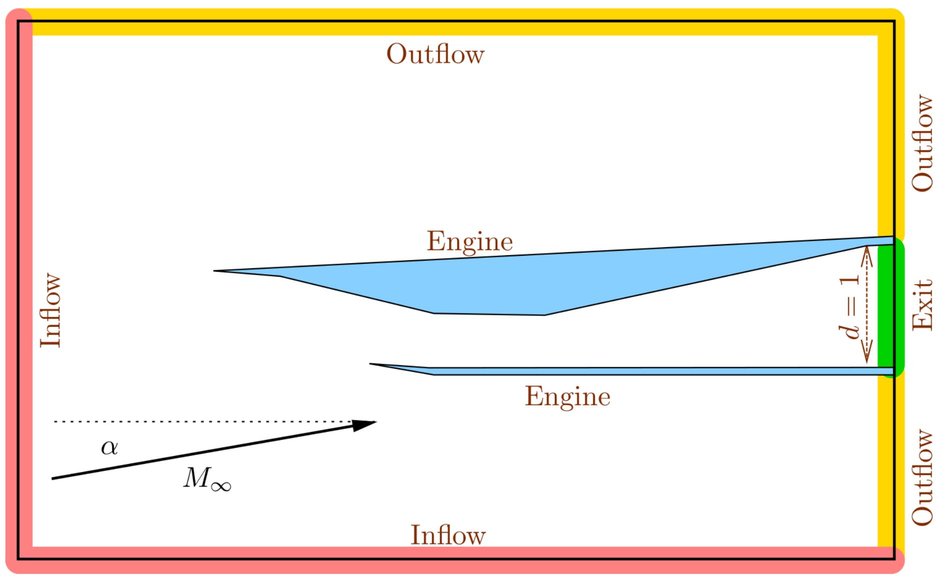


Figure 1: Engine geometry and boundary conditions.

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 $M_\infty = 2.0$. 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

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

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

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

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 . Specifically, define the CFL number for cell i to be,

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

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 $\Delta t / A_i$ for each cell, and this can be calculated by rearranging Equation 5,

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

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: $\mathrm{CFL}_i = \mathrm{CFL} = 1.0$ 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 the 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.

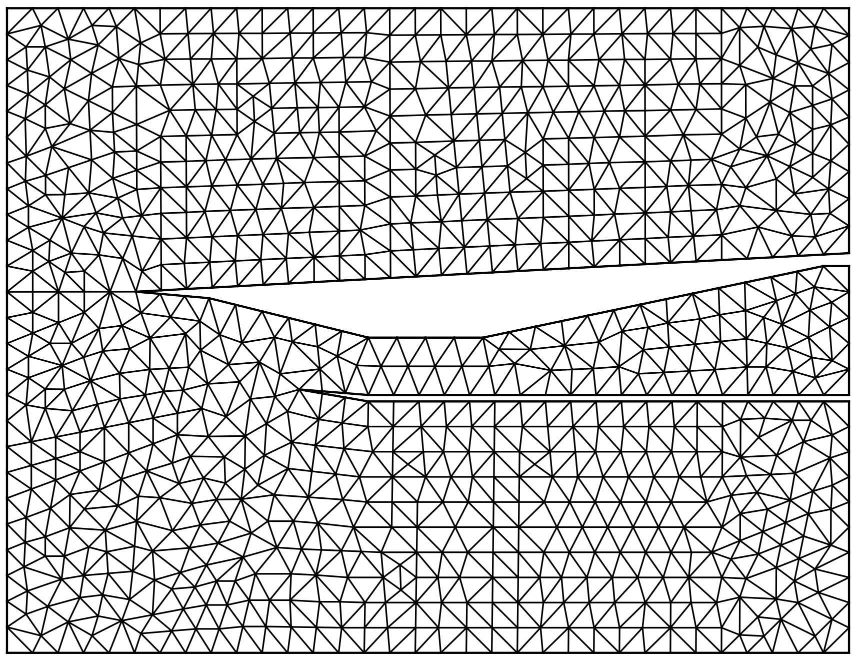


Figure 2: Baseline mesh.

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.

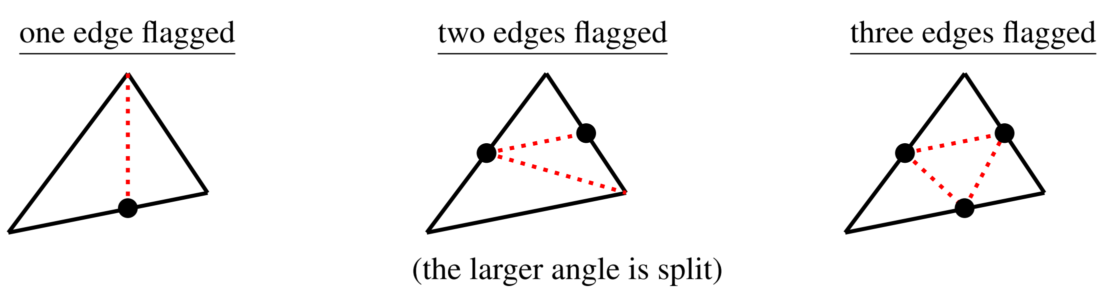


Figure 3: Refinement of triangles given edge splittings.

4 Tasks and Deliverables

4.1 Implementation of Roe Flux

[10%] Write a function that computes the Roe flux, with $\epsilon = 0.1c$ for the entropy fix. Test your flux with several cases, including subsonic and supersonic states, and varying normal vectors. In your report, describe your implementation and your tests.

4.2 Implementing Finite Volume Method

[25%] Implement a first-order finite volume method to obtain a steady-state flow solution on a given mesh. The code should:

- read and process the mesh files,
- iterate until the residual norm is less than the specified tolerance,
- calculate the average total pressure recovery output,
- monitor and log the residual norm and output convergence

You may also wish to equip your code with the ability to write restart files at the end of the simulation and periodically during a run. In your report, describe your code structure, algorithms, and data structures.

4.3 Convergences

[15%] Run your code on the baseline mesh at $\alpha = 1^{\circ}$, starting with a uniform freestream state at this α. Perform the following post-processing:

- Plot the convergence of the given residual L_1 norm versus time step iterations.
- Plot the convergence of the ATPR output versus time step iterations.
- For the converged solution, make two field plots: one of the Mach number, and one of the total pressure.

4.4 Implementing Mach Number Jumps

[25%] Implement the mesh adaptation algorithm based on the Mach number jumps. Run your algorithm for $\alpha = 1^\circ$ with at least 5 adaptive iterations, and perform the following post-processing:

- Plot the sequence of your adapted meshes.
- Plot the Mach number and total pressure fields on your finest mesh.
- Plot the ATPR output versus number of cells in the mesh – one data point per adaptive iteration.

Discuss the results, including areas targeted for adaption and the convergence of the output.

4.5 Adaptive Iterations

[15%] Perform adaptive iterations for $\alpha = [0.5, 1, 1.5, 2, 2.5, 3]$ degrees. Run the same number of adaptive iterations for each α at least 5. Plot the ATPR output from your finest mesh versus alpha, and discuss the trend. Include flowfield plots to augment your discussion.

Appendices

A Additional Supporting Code

Algorithm 1: Python Edge Hash

```python
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 2: Python Plot Mesh

```
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.tripcolor(V[:,0], V[:,1], triangles=E)
    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')
    dosave = not not fname
    plt.axis('equal')
    plt.axis('off')
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout(); plt.show(block=(not dosave))
    if (dosave): plt.savefig(fname, bbox_inches='tight')
    plt.close(f)

#-----------------------------------------------------------
def main():
    Mesh = readgri('mesh0.gri')
    plotmesh(Mesh, [])

if __name__ == "__main__":
    main()
```

Algorithm 3: Python Read Grid

```
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, bgroup) 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

#-----------------------------------------------------------
def readgri(fname):
    f = open(fname, 'r')
    Nn, Ne, dim = [int(s) for s in f.readline().split()]
    # read vertices
    V = np.array([[float(s) for s in f.readline().split()] for n in range(Nn)])
    # read boundaries
    NB = int(f.readline())
    B = []; Bname = []
    for i in range(NB):
        s = f.readline().split(); Nb = int(s[0]); Bname.append(s[2])
        Bi = np.array([[int(s)-1 for s in f.readline().split()] for n in range(Nb)])
        B.append(Bi)
    # read elements
    Ne0 = 0; E = []
    while (Ne0 < Ne):
        s = f.readline().split(); ne = int(s[0])
        Ei = np.array([[int(s)-1 for s in f.readline().split()] for n in range(ne)])
        E = Ei if (Ne0==0) else np.concatenate((E,Ei), axis=0)
        Ne0 += ne
    f.close()
    # make IE, BE structures
    IE, BE = edgehash(E, B)
    Mesh = {'V':V, 'E':E, 'IE':IE, 'BE':BE, 'Bname':Bname }
    return Mesh

#-----------------------------------------------------------
def writegri(Mesh, fname):
    V = Mesh['V']; E = Mesh['E']; BE = Mesh['BE']; Bname = Mesh['Bname'];
    Nv, Ne, Nb = V.shape[0], E.shape[0], BE.shape[0]
    f = open(fname, 'w')
    f.write('%d␣%d␣2\n'%(Nv, Ne))
    for i in range(Nv):
        f.write('%.15e␣%.15e\n'%(V[i,0], V[i,1]));
```

```
    nbg = 0
    for i in range(Nb): nbg = max(nbg, BE[i,3])
    nbg += 1
    f.write('%d\n'%(nbg))
    for g in range(nbg):
        nb = 0
        for i in range(Nb): nb += (BE[i,3] == g)
        f.write('%d 2 %s\n'%(nb, Bname[g]))
        for i in range(Nb):
            if (BE[i,3]==g): f.write('%d %d\n'%(BE[i,0]+1, BE[i,1]+1))
    f.write('%d 1 TriLagrange\n'%(Ne))
    for i in range(Ne):
        f.write('%d %d %d\n'%(E[i,0]+1, E[i,1]+1, E[i,2]+1))
    f.close()


#-----------------------------------------------------------
def main():
    Mesh = readgri('xflow/v2/capsule.gri')
    writegri(Mesh, 'xflow/v2/test.gri')

if __name__ == "__main__":
    main()
```