Computational Fluid Dynamics (CFD) Simulation of CR25 Using the OpenFOAM Open-Source Package

Author:
Dara Rahmat Samii

Master of Applied Science in Mechanical Engineering
Gina Cody School of Engineering and Computer Science
Concordia University
Montreal, Canada

Email:
dara.rahmatsamii@mail.concordia.ca

July 2025

Abstract

Aether was SpaceConcordia's non-ordinary transition rocket launched on Monday, August 18th, 2025, in Timmins, Ontario, for Launch Canada. Due to its novel design and the team's limited experience with this configuration, Computational Fluid Dynamics (CFD) was utilized to provide the design team with better understanding of the forces and aerodynamic loads acting on the rocket. We hope that computer-aided techniques such as CFD become more common in student teams, and we have demonstrated that it can be a powerful tool to assess aerodynamic forces and flight performance during different stages of flight.

Launch Canada safety officials generally do not accept undergraduate CFD simulations, believing that without proper knowledge, CFD is merely "CAD in, fancy contours out." This report serves as a guide to help future practitioners perform proper mesh analysis, select appropriate solvers and boundary conditions, and configure cases to obtain verified, stable, and valid outputs that meaningfully assist other team members in designing better rockets.

I have endeavored to make this report as detailed as possible because the intended audience consists primarily of 2nd or 3rd-year undergraduates with limited experience in CFD and specifically OpenFOAM. Therefore, the first chapters describe the governing physics and theory behind each decision. In the appendix, actual OpenFOAM codes and a guide for submitting jobs to HPC systems are provided. Recognizing that undergraduate mechanical engineering students typically lack Linux experience, one full chapter is dedicated to teaching basic Linux bash commands.

It is the author's hope that readers will develop a passion for CFD and, upon discovering new methods and techniques to improve this report or the CFD codes, will document these improvements and pass them on to future generations.

For any questions, please contact me via my university email dara.rahmatsamii@mail.concordia.ca. If I have graduated and no longer have access to my university email, please contact me at darasamii@gmail.com.

Contents

List of Figures

List of Tables

Chapter 1

Grid Mesh

Generating a high-quality mesh is an essential part of any CFD simulation. In many cases, the quality of the mesh can have a greater impact on solution accuracy than the solver itself or the choice of discretization schemes. For aerodynamic simulations, having a fine inflation layer near solid boundaries is critical, as it strongly affects drag, lift, and other forces acting on the rocket. Therefore, selecting a meshing tool capable of producing sufficiently refined boundary layers is of paramount importance.

OpenFOAM provides two built-in meshing utilities: `blockMesh` and `snappyHexMesh`.[1] `blockMesh` creates structured, hexahedral meshes using simple geometry and manual patch definition. In contrast, `snappyHexMesh` works on surface geometries (e.g., STL files), automatically snapping and refining the mesh around geometry features. However, it requires an initial base mesh, typically generated by `blockMesh`, as a starting point.

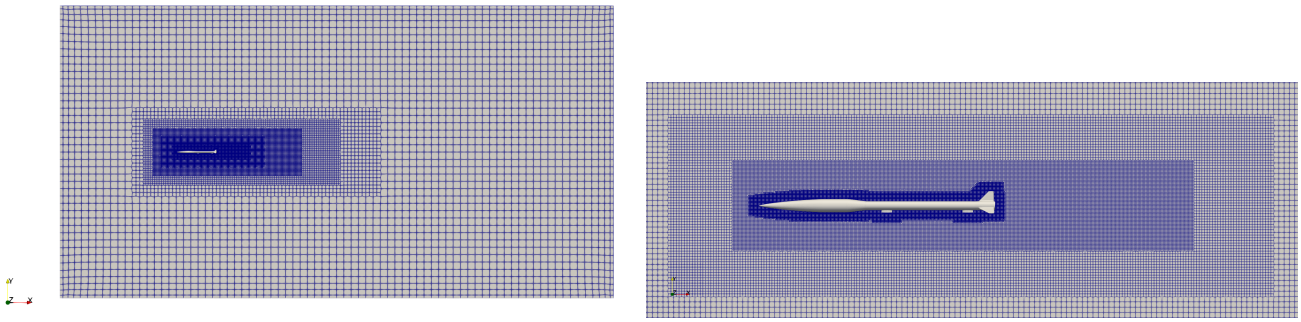
Although we initially used `snappyHexMesh`, it struggled to generate sufficiently high-quality boundary layer mesh near the rocket surface, particularly in regions with strong curvature. It was able to correctly identify outer domain regions but produced low cell quality near the wall, affecting simulation accuracy.

As a result, we transitioned to using `cfMesh`,[2] a powerful meshing library that, while not built into OpenFOAM by default, integrates seamlessly and supports syntax consistent with other OpenFOAM utilities.

The meshing procedure began with a CAD model of the rocket and an initial bounding block representing the freestream domain. The domain size was set to $38 \times 20 \times 20$ (in meters), with the rocket positioned slightly forward in the domain to ensure sufficient space for the wake and tail region.

To reduce the total cell count, a coarse base mesh was used, with progressively refined regions surrounding the rocket. Several refinement zones were introduced around the body, and each refinement region subdivided the cells by one level. The surface of the rocket was refined up to 7 levels, and any edge exceeding 2° in angular deviation was considered a feature edge and refined up to 9 levels. A schematic of the refinement box and rocket position is shown in Figure 1.1.

The generated mesh is illustrated in Figure 1.2, which shows both the full cross-section and a zoomed-in view near the rocket.



(a) Cross-section of the full mesh domain.

(b) Zoomed-in cross-section focusing on the rocket.

Figure 1.2: Visualizations of the computational mesh used for the rocket domain.

[1] `snappyHexMesh` is OpenFOAM's unstructured mesh generator for complex geometry based on STL input.

[2] `cfMesh` is an open-source meshing tool that integrates with OpenFOAM and supports fully automated mesh generation.

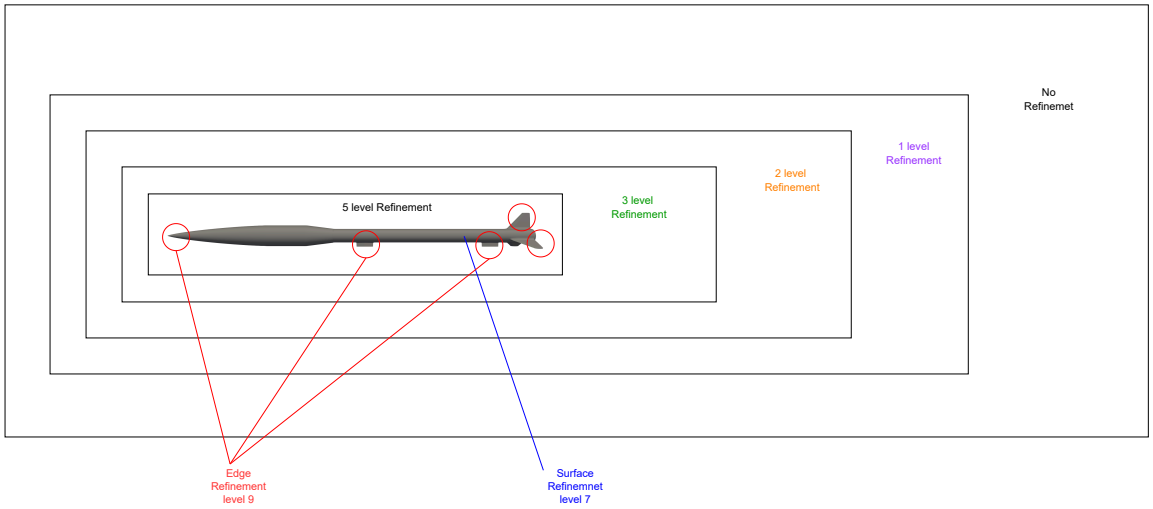
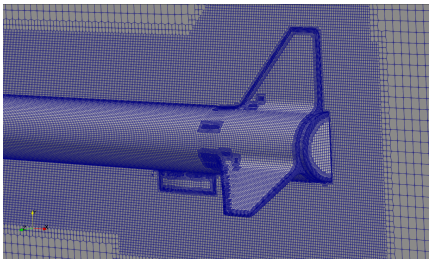
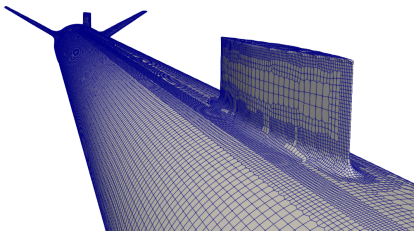


Figure 1.1: Schematic of the mesh grid around the rocket in freestream.

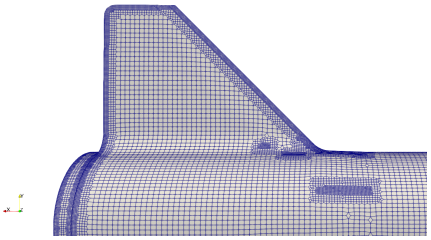
Sharp geometric features (e.g., fin edges, cone tips) often cause issues for automatic meshing algorithms, resulting in irregular cell shapes or stair-stepping artifacts that degrade solution accuracy. To mitigate this, the CAD features were explicitly extracted and passed to the meshing tool as feature edges. These were refined up to 9 levels. Close-up views of the resulting mesh quality around critical components are shown in Figure 1.3.



(a) Detailed mesh around the rocket fin.



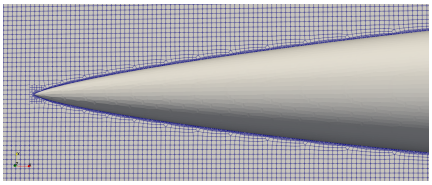
(b) Mesh near the launch lug (zoomed view).



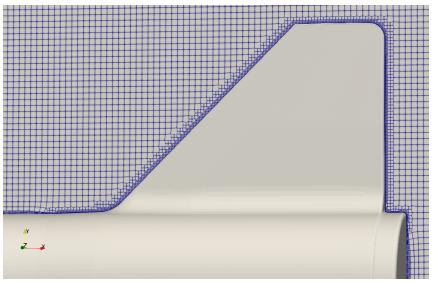
(c) Zoomed-in view of mesh at fin leading edge.

Figure 1.3: Close-up views of mesh resolution around critical rocket components.

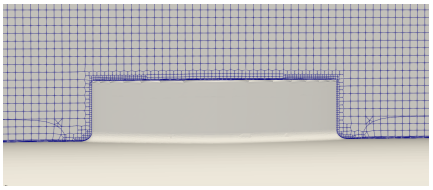
High-quality mesh with low skewness and good orthogonality is crucial to accurately capture near-wall flow physics. To resolve the boundary layer, 7 inflation layers were applied along all surfaces of the rocket, with a growth ratio of 1.2. These layers allow the mesh to resolve the steep velocity gradients present near the wall (e.g., in the viscous sublayer). Figure 1.4 shows the inflation layers in key regions of the rocket.



(a) Inflation layers near the nose cone.



(b) Inflation layers around the rocket fin.



(c) Inflation layers near the launch lug.

Figure 1.4: Visualization of inflation layer refinement in critical boundary regions.

Chapter 2

Governing Equations

The governing equations for classical aerodynamic analysis are the compressible Navier–Stokes equations. Due to the high velocities involved, the air undergoes compression, making it impossible to neglect changes in density. To account for this, a relationship between pressure and density—typically in the form of an equation of state—must be introduced.

Furthermore, high-speed flow and low-density conditions result in a high Reynolds number, which indicates the presence of turbulence. Resolving all turbulent eddies directly would require immense computational resources. As a result, turbulence models are employed to approximate their effects efficiently.

2.1 Continuity Equation

The conservation of mass for a compressible fluid is expressed by the continuity equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = 0 \tag{2.1}$$

Here, ρ represents the fluid density in $\mathrm{kg/m^3}$, t is time in seconds, and $\mathbf{U}$ is the velocity vector in m/s. The operator $\nabla\cdot$ denotes the divergence, which measures the net outflow of mass per unit volume. This equation ensures that mass is conserved in the flow domain, accounting for both temporal and spatial changes in density.

The divergence term $\nabla \cdot (\rho \mathbf{U})$ expands to the sum of the spatial derivatives of the momentum flux components in each direction:

$$\nabla \cdot (\rho \mathbf{U}) = \frac{\partial (\rho U_x)}{\partial x} + \frac{\partial (\rho U_y)}{\partial y} + \frac{\partial (\rho U_z)}{\partial z} \tag{2.2}$$

This expression represents the net rate at which mass is flowing out of an infinitesimal control volume due to fluid motion in the x, y, and z directions. If this term is positive at a point, mass is leaving the control volume faster than it enters, which must be balanced by a decrease in local density ρ over time to conserve mass.

In steady-state compressible flows, where $\partial \rho / \partial t = 0$, the continuity equation simplifies to:

$$\nabla \cdot (\rho \mathbf{U}) = 0 \tag{2.3}$$

This indicates that the mass flux into and out of any control volume must balance exactly, even though the density may vary in space.

2.2 Momentum Equation

The conservation of momentum for a compressible, viscous, and turbulent fluid is governed by the Navier–Stokes momentum equation:

$$\frac{\partial (\rho \mathbf{U})}{\partial t} + \nabla \cdot (\rho \mathbf{U} \otimes \mathbf{U}) = -\nabla p + \nabla \cdot \boldsymbol{\tau}_{\text{eff}} \tag{2.4}$$

The term $\frac{\partial(\rho\mathbf{U})}{\partial t}$ represents the local (temporal) rate of change of momentum, while the second term, $\nabla\cdot(\rho\mathbf{U}\otimes\mathbf{U})$, accounts for the convective transport of momentum due to fluid motion. In component form, this convective term can be written as:

$$\nabla\cdot(\rho\mathbf{U}\otimes\mathbf{U}) = \begin{bmatrix} \frac{\partial(\rho U_x U_x)}{\partial x} + \frac{\partial(\rho U_x U_y)}{\partial y} + \frac{\partial(\rho U_x U_z)}{\partial z} \\ \frac{\partial(\rho U_y U_x)}{\partial x} + \frac{\partial(\rho U_y U_y)}{\partial y} + \frac{\partial(\rho U_y U_z)}{\partial z} \\ \frac{\partial(\rho U_z U_x)}{\partial x} + \frac{\partial(\rho U_z U_y)}{\partial y} + \frac{\partial(\rho U_z U_z)}{\partial z} \end{bmatrix} \tag{2.5}$$

Each row corresponds to a momentum equation in the x-, y-, and z-directions, respectively. These terms represent the net flux of momentum in each direction due to the transport by the flow itself.

The right-hand side of the momentum equation consists of two terms. The first, $-\nabla p$, is the pressure gradient force, which drives acceleration in the fluid. The second, $\nabla\cdot\boldsymbol{\tau}_{\text{eff}}$, represents the divergence of the effective stress tensor, which accounts for both viscous and turbulent stresses.

The effective stress tensor is defined as:

$$\boldsymbol{\tau}_{\text{eff}} = (\mu+\mu_t)\left[\nabla\mathbf{U} + (\nabla\mathbf{U})^T - \frac{2}{3}(\nabla\cdot\mathbf{U})\mathbf{I}\right] \tag{2.6}$$

Here, μ is the molecular (dynamic) viscosity of the fluid, and μ_t is the turbulent viscosity obtained from the turbulence model (e.g., k–ω SST). The term $\nabla\mathbf{U}$ is the velocity gradient tensor, and $(\nabla\mathbf{U})^T$ is its transpose. The trace term involving $\nabla\cdot\mathbf{U}$ ensures the stress tensor remains traceless in incompressible or nearly incompressible regions. $\mathbf{I}$ is the identity tensor.

This formulation captures the effects of internal friction and turbulence within the fluid, which are essential for accurately resolving shear layers, wakes, and boundary layers, especially at high Reynolds numbers.

2.3 Energy Equation

The conservation of energy for a compressible, viscous, and turbulent fluid is typically expressed in terms of specific enthalpy. The steady-state or time-dependent energy equation, accounting for both viscous dissipation and turbulent heat transport, is given by:

$$\frac{\partial(\rho h)}{\partial t} + \nabla\cdot(\rho\mathbf{U}h) = \frac{Dp}{Dt} + \nabla\cdot\left(\frac{\mu+\mu_t}{Pr_t}\nabla h\right) + \boldsymbol{\tau}_{\text{eff}} : \nabla\mathbf{U} \tag{2.7}$$

In this equation, h is the specific enthalpy (J/kg), and the left-hand side represents the local and convective transport of energy: the term $\frac{\partial(\rho h)}{\partial t}$ accounts for the temporal change of energy, while $\nabla\cdot(\rho\mathbf{U}h)$ represents the energy convected by the fluid.

The first term on the right-hand side, $\frac{Dp}{Dt}$, is the material derivative of pressure and represents the compressibility effects (i.e., energy change due to pressure variation following the fluid motion). The operator $\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{U}\cdot\nabla$ denotes the total (or material) derivative.

The second term, $\nabla\cdot\left(\frac{\mu+\mu_t}{Pr_t}\nabla h\right)$, accounts for energy diffusion due to molecular and turbulent conduction. Here, μ is the dynamic viscosity, μ_t is the turbulent viscosity, and Pr_t is the turbulent Prandtl number (typically between 0.85 and 0.9 for air), which governs the ratio of momentum to thermal diffusivity.

The final term, $\boldsymbol{\tau}_{\text{eff}} : \nabla\mathbf{U}$, is the viscous dissipation term. The symbol : denotes the double contraction (or double dot product) between the stress tensor and the velocity gradient tensor. This term quantifies the conversion of mechanical energy into internal energy due to viscous forces—particularly significant in regions of strong shear, such as boundary layers or shock waves.

The specific enthalpy h is related to the temperature T through the constant-pressure specific heat C_p:

$$h = C_p T \tag{2.8}$$

2.4 Equation of State

To close the system of equations for compressible flow, an equation of state is required to relate thermodynamic variables. For an ideal (perfect) gas, the pressure is related to density and temperature by the ideal gas law:

$$p = \rho RT = \frac{\rho R_u T}{M} \tag{2.9}$$

In this expression, p is the pressure (Pa), ρ is the fluid density (kg/m^3), and T is the absolute temperature (K). The symbol R represents the specific gas constant (J/(kg·K)), which is defined as:

$$R = \frac{R_u}{M} \tag{2.10}$$

Here, R_u is the universal gas constant, equal to 8314 J/(mol·K), and M is the molar mass of the gas in kg/mol. For air, $M \approx 0.02897$ kg/mol, resulting in a specific gas constant of approximately $R \approx 287$ J/(kg·K).

The equation of state allows pressure to be computed from the temperature and density, which is essential when using energy-based formulations, such as in the energy equation solved by `rhoSimpleFoam`.

2.5 Sutherland's Law for Dynamic Viscosity

To increase the physical accuracy of the simulation, the temperature-dependent dynamic viscosity of a gas is modeled using *Sutherland's law*, which accounts for molecular motion and intermolecular collisions in a dilute gas. This model is particularly relevant in compressible flow simulations where temperature variations are significant. In regions affected by shock waves—such as near the nose cone—the frictional heating is substantial, leading to sharp local increases in temperature. Switching from a constant-viscosity model to Sutherland's law does not compromise numerical stability but can significantly improve the accuracy of viscous stress calculations, particularly in high-speed aerodynamic flows.

$$\mu(T) = A_s \frac{T^{3/2}}{T + T_s} \tag{2.11}$$

In this equation, $\mu(T)$ is the dynamic viscosity as a function of temperature, given in kg/(m·s). The constant $A_s = 1.4792 \times 10^{-6}$ kg/(m·s·K$^{0.5}$) is the Sutherland constant, and $T_s = 116$ K is the Sutherland reference temperature—both values are standard for air. The exponent 3/2 reflects the kinetic theory prediction that viscosity increases with the square root of temperature due to enhanced molecular momentum transport.

Sutherland's law provides a more accurate representation of viscosity in gases than constant-viscosity models, especially in flows with steep temperature gradients, such as shock layers or boundary layers in high-speed regimes.

2.6 $k-\omega$ SST Turbulence Model

As mentioned previously, directly simulating all turbulent eddies across all spatial and temporal scales—known as Direct Numerical Simulation (DNS)—is computationally infeasible for most practical engineering problems due to the enormous resource requirements. To address this, various turbulence modeling approaches have been developed.

A widely used and practical class of models is the Reynolds-Averaged Navier–Stokes (RANS) family, which averages the governing equations over time to model the effects of turbulence. Common RANS models include the one-equation Spalart–Allmaras model, the two-equation k–ϵ, and the k–ω models.

In this study, the *Shear Stress Transport* (SST) variant of the k–ω model is employed. This model blends the advantages of both the k–ω formulation near the wall (for better treatment of boundary layers) and the k–ϵ formulation in the free stream (for improved far-field behavior). The SST model introduces a blending function to switch between these formulations smoothly based on the local flow conditions.

2.6.1 Turbulent Kinetic Energy (k)

In the k–ω SST formulation, the transport equation for the turbulent kinetic energy k is given by:

$$\frac{\partial(\rho k)}{\partial t} + \nabla \cdot (\rho \mathbf{U} k) = \nabla \cdot [(\mu + \mu_t \sigma_k) \nabla k] + P_k - \beta^* \rho \omega k \tag{2.12}$$

Here, k is the turbulent kinetic energy measured in m^2/s^2, and ρ is the fluid density. The term μ is the dynamic viscosity, and μ_t is the turbulent (eddy) viscosity. The parameter σ_k is the turbulent Prandtl number for k, and ω is the specific dissipation rate in s^{-1}. The constant β^* is a model coefficient. The term P_k represents the production of turbulent kinetic energy, which acts as a source term in the equation.

The production term P_k is defined as:

$$P_k = \tau_{ij} \frac{\partial U_i}{\partial x_j} \tag{2.13}$$

In this expression, τ_{ij} is the Reynolds stress tensor (in Pascals), U_i is the i^{th} component of the velocity vector, and x_j is the j^{th} spatial coordinate. This formulation expresses how turbulence is generated through the interaction of fluctuating velocity components and mean velocity gradients.

2.6.2 Specific Dissipation Rate (ω)

The transport equation for the specific dissipation rate ω in the k–ω SST turbulence model is formulated as:

$$\frac{\partial(\rho\omega)}{\partial t} + \nabla \cdot (\rho \mathbf{U} \omega) = \nabla \cdot [(\mu + \mu_t \sigma_\omega) \nabla \omega] + \frac{\lambda}{\nu_t} P_k - \beta \rho \omega^2 + 2(1 - F_1) \frac{\rho \sigma_{\omega 2}}{\omega} \nabla k \cdot \nabla \omega \tag{2.14}$$

In this equation, ω is the specific dissipation rate (1/s), and μ and μ_t denote the dynamic and turbulent viscosities, respectively. The turbulent Prandtl number for ω is given by σ_ω, which is blended depending on the flow region. The term λ is a model coefficient, also blended, and ν_t is the kinematic turbulent viscosity (m^2/s). The source term $\frac{\lambda}{\nu_t} P_k$ accounts for production of ω, while the dissipation is controlled by the $-\beta\rho\omega^2$ term. The blending function F_1 is used to transition smoothly between the near-wall k–ω model and the k–ϵ formulation in the free stream. It is defined by:

$$F_1 = \tanh\left((\arg)^4 \right) \tag{2.15}$$

where the argument arg is calculated as:

$$\arg = \min\left[\max\left(\frac{\sqrt{k}}{\beta^* \omega d}, \frac{500\nu}{d^2 \omega} \right), \frac{4\rho\sigma_{\omega 2} k}{CD_{k\omega} d^2} \right] \tag{2.16}$$

In this formulation, d is the distance to the nearest wall (in meters), ν is the kinematic viscosity, and $CD_{k\omega}$ is the cross-diffusion term. The functions min and max ensure that the blending is physically reasonable in all regions. The tanh function provides a smooth transition between the limits of the model near the wall and far from it.

2.7 Air Properties

Air is the primary working fluid in aerodynamic simulations of the rocket. The thermophysical properties of air required for solving the governing equations are summarized in Table 2.4.

Table 2.1: Thermophysical Properties of Air

Property	**Value**	**Units**
Molar mass (M)	28.9	g/mol
Specific heat at constant pressure (C_p)	1005	J/(kg·K)
Formation enthalpy (H_f)	0	J/kg
Sutherland constant (A_s)	1.4792×10^{-6}	kg/(m·s·K$^{0.5}$)
Sutherland temperature (T_s)	116	K
Prandtl number (Pr)	0.7	-
Specific gas constant (R)	287.0	J/(kg·K)

2.8 Boundary Conditions

Properly defining boundary conditions is critical for achieving accurate and stable CFD simulations. In the present simulation, different sets of boundary conditions are applied to the rocket surface and the surrounding computational domain to reflect the physical behavior of the flow.

The key boundary condition types are described below:

Table 2.2: Boundary conditions used in the simulation.

Variable	**Rocket Surface (Wall)**	**Surrounding Domain**
Velocity **U**	`noSlip`	`inletOutlet`
Temperature T	`zeroGradient`	`inletOutlet`
Pressure p	`zeroGradient`	`freestream`
Turbulent kinetic energy k	`kqRWallFunction`	`inletOutlet`
Specific dissipation rate ω	`omegaWallFunction`	`inletOutlet`
Turbulent viscosity ν_t	`nutkWallFunction`	-
Turbulent thermal diffusivity α_t	`alphatWallFunction`	-

noSlip Applied to **U** (velocity) on the rocket surface. This enforces zero velocity at the wall, modeling the physical condition that fluid adheres to solid surfaces (i.e., $\mathbf{U} = 0$ at the wall).

zeroGradient Used for T (temperature) and p (pressure) on the rocket wall, this condition specifies that the normal derivative of the variable is zero, i.e., no heat or pressure flux across the wall:

$$\frac{\partial T}{\partial n} = 0 \qquad \text{and} \qquad \frac{\partial p}{\partial n} = 0$$

inletOutlet A versatile mixed boundary condition used on the outer boundaries for **U**, T, k, and ω. It behaves like an inlet when the flow enters the domain (applying a fixed value) and like a zero-gradient outlet when flow exits. It allows natural inflow/outflow behavior in freestream simulations.

freestream Used for pressure p on the domain boundaries. This condition blends between fixed value and zeroGradient based on the flow direction and magnitude, stabilizing compressible simulations with freestream inflow/outflow.

kqRWallFunction Applied to k on the rocket wall. It calculates the near-wall turbulent kinetic energy based on the wall shear stress and roughness. This function adapts the value of k using an empirical relation suited for high Reynolds number wall flows.

omegaWallFunction Applied to ω at the wall. It sets the specific dissipation rate using near-wall turbulence theory, ensuring compatibility with k–ω models:

$$\omega = \frac{6\nu}{\beta_1 y^2}$$

where y is the distance to the nearest wall, and β_1 is a model constant.

nutkWallFunction Applied to turbulent viscosity ν_t. This function estimates ν_t using the values of k and wall distance:

$$\nu_t = C_\mu \frac{k^{1/2} y}{1 + \frac{y k^{1/2}}{E\nu}}$$

alphatWallFunction Applied to turbulent thermal diffusivity α_t, this condition links α_t to the turbulent viscosity via the turbulent Prandtl number:

$$\alpha_t = \frac{\nu_t}{Pr_t}$$

where Pr_t is the turbulent Prandtl number.

2.9 Summary

Table 2.3: Governing equations.

Principle	**Equation**
Conservation of Mass	$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{U}) = 0$
Conservation of Momentum	$\frac{\partial (\rho \vec{U})}{\partial t} + \nabla \cdot (\rho \vec{U} \otimes \vec{U}) = -\nabla p + \nabla \cdot \boldsymbol{\tau}_{\text{eff}} + \rho \vec{g}$
Conservation of Energy	$\frac{\partial (\rho h)}{\partial t} + \nabla \cdot (\rho \vec{U} h) = \frac{Dp}{Dt} + \nabla \cdot \left(\frac{\mu + \mu_t}{Pr_t} \nabla h \right) + \boldsymbol{\tau}_{\text{eff}} : \nabla \vec{U}$
Equation of State	$p = \rho R T = \frac{\rho R_u T}{M}$
Sutherland's Law	$\mu(T) = A_s \frac{T^{3/2}}{T + T_s}$
k–ω SST Turbulence Model	$\frac{\partial (\rho k)}{\partial t} + \nabla \cdot (\rho \vec{U} k) = \nabla \cdot [(\mu + \mu_t \sigma_k) \nabla k] + (\tau_{ij} \frac{\partial U_i}{\partial x_j}) - \beta^* \rho \omega k$
	$\frac{\partial (\rho \omega)}{\partial t} + \nabla \cdot (\rho \vec{U} \omega) = \nabla \cdot [(\mu + \mu_t \sigma_\omega) \nabla \omega] + \frac{\lambda}{\nu_t} (\tau_{ij} \frac{\partial U_i}{\partial x_j}) - \beta \rho \omega^2 + 2(1 - F_1) \frac{\rho \sigma_{\omega 2}}{\omega} \nabla k \cdot \nabla \omega$
	$F_1 = \tanh \left((\min \left[\max \left(\frac{\sqrt{k}}{\beta^* \omega d}, \frac{500 \nu}{d^2 \omega} \right), \frac{4 \rho \sigma_{\omega 2} k}{C D_{k\omega} d^2} \right])^4 \right)$
	$C D_{k\omega} = \max \left(2 \rho \sigma_{\omega 2} \frac{1}{\omega} \nabla k \cdot \nabla \omega, 10^{-20} \right)$

Table 2.4: Thermophysical Properties of Air

Property	**Value**	**Units**
Molar mass (M)	28.9	g/mol
Specific heat at constant pressure (C_p)	1005	J/(kg·K)
Formation enthalpy (H_f)	0	J/kg
Sutherland constant (A_s)	1.4792×10^{-6}	kg/(m·s·K$^{0.5}$)
Sutherland temperature (T_s)	116	K
Prandtl number (Pr)	0.7	-
Specific gas constant (R)	287.0	J/(kg·K)

Table 2.5: Boundary conditions used in the simulation.

Variable	**Rocket Surface (Wall)**	**Surrounding Domain**
Velocity **U**	`noSlip`	`inletOutlet`
Temperature T	`zeroGradient`	`inletOutlet`
Pressure p	`zeroGradient`	`freestream`
Turbulent kinetic energy k	`kqRWallFunction`	`inletOutlet`
Specific dissipation rate ω	`omegaWallFunction`	`inletOutlet`
Turbulent viscosity ν_t	`nutkWallFunction`	-
Turbulent thermal diffusivity α_t	`alphatWallFunction`	-

Chapter 3

Numerical Methods

In this chapter, numerical methods, strategies, and concepts are discussed.

3.1 Discretization Schemes

This section provides explanations of the discretization schemes summarized in Table **??**. These schemes are used to approximate the differential operators that appear in the governing equations solved by `rhoSimpleFoam`.

There is a trade-off between stability and accuracy. As more accurate discretization schemes are chosen, stability may decrease, leading to oscillations in the residuals and potential simulation divergence. On the other hand, using more stable schemes leads to steadily decreasing residuals, but introduces greater numerical diffusion, which can result in physically incorrect and less accurate solutions.

To address this issue, a two-phase strategy was implemented. During the first 300 iterations, stable but low-accuracy discretization schemes listed in Table 3.1 were used. Once the residuals decreased significantly, the schemes were switched to those in Table 3.2. In other words, the solution obtained using the stable schemes was used as the initial guess for the more accurate simulation. This approach allowed the overall solution to be stabilized first, reducing the likelihood of divergence, and then improved in terms of physical correctness and accuracy by applying more accurate discretization schemes.

Table 3.1: Summary of the stable discretization schemes used for solving the governing equations.

Category	**Terms/Variables**	**Discretization Schemes**
Time	$\frac{\partial \Phi}{\partial t}$	Euler
Gradients	$\nabla \Phi$	Cell limited linear
Convection	$\nabla \cdot (\phi \mathbf{U})$, $\nabla \cdot (\phi \tilde{\nu})$	Bounded linear upwind limited gradient
	$\nabla \cdot (\phi e)$, $\nabla \cdot (\phi E_{kp})$, $\nabla \cdot (\phi_d p)$	Upwind
	$\nabla \cdot (\phi h)$, $\nabla \cdot (\phi K)$	Bounded upwind
	$\nabla \cdot (\phi k)$, $\nabla \cdot (\phi \omega)$	Bounded upwind
	$\nabla \cdot (\boldsymbol{\tau})$	Linear
Laplacians	$\nabla^2 \Phi$	Linear limited
Interpolation	Φ_f	Linear
Surface gradients	$\frac{\partial \phi}{\partial n}$	Limited

3.1.1 Time Discretization

The transient term $\frac{\partial \phi}{\partial t}$ is discretized using the **Euler** scheme, which is a first-order implicit method. It is unconditionally stable and commonly used in steady-state solvers for pseudo-time advancement:

$$\left(\frac{\partial \phi}{\partial t}\right) \approx \frac{\phi^{n+1} - \phi^n}{\Delta t}$$

Table 3.2: Summary of the accurate discretization schemes used for solving the governing equations.

Category	**Terms/Variables**	**Discretization Schemes**
Time	$\frac{\partial \phi}{\partial t}$	Euler
Gradients	$\nabla \Phi$	Cell limited linear
Convection	$\nabla \cdot (\phi \mathbf{U})$	Bounded linear upwind limited gradient face
	$\nabla \cdot (\phi h)$, $\nabla \cdot (\phi K)$	Bounded linear upwind limited gradient
	$\nabla \cdot (\phi k)$, $\nabla \cdot (\phi \omega)$,$\nabla \cdot (\phi \tilde{\nu})$	Bounded linear upwind limited gradient
	$\nabla \cdot (\phi e)$, $\nabla \cdot (\phi E_{kp})$	Linear upwind limited gradient
	$\nabla \cdot (\boldsymbol{\tau})$	Linear
	$\nabla \cdot (\phi_d p)$	Upwind
Laplacians	$\nabla^2 \Phi$	Linear limited
Interpolation	Φ_f	Linear
Surface gradients	$\frac{\partial \Phi}{\partial n}$	Limited

3.1.2 Gradient Schemes

Gradient terms like $\nabla \phi$ and ∇p are approximated using the **cell-limited linear** scheme. The base **linear** method uses Gauss's theorem:

$$\nabla \phi \approx \frac{1}{V_P} \sum_f \phi_f \mathbf{S}_f$$

where V_P is the cell volume and ϕ_f is the interpolated value at the face.

Cell-limited means that a limiter is applied to the gradient to reduce non-physical oscillations in cells with sharp variations. The limiter restricts the gradient to remain within local bounds, enhancing numerical stability without significantly compromising accuracy.

3.1.3 Convection Schemes

Upwind Scheme The **upwind** scheme uses the value of ϕ from the upstream (donor) cell:

$$\phi_f = \begin{cases} \phi_P & \text{if } \mathbf{U}_f \cdot \mathbf{S}_f > 0 \\ \phi_N & \text{otherwise} \end{cases}$$

This scheme is first-order accurate but robust and highly diffusive.

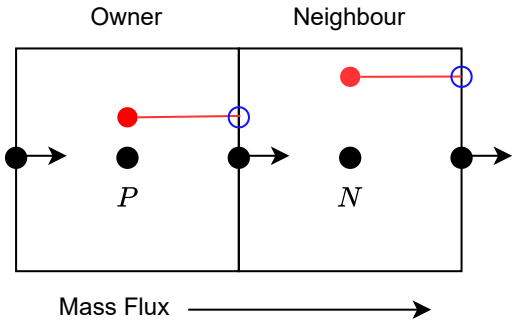


Figure 3.1: Schematic of Upwind scheme

Linear Scheme The **linear** scheme, also known as central differencing, computes the face value ϕ_f as a weighted interpolation between neighboring cell centers (typically linear in mesh geometry):

$$\phi_f = \frac{1}{2}(\phi_P + \phi_N)$$

This scheme is second-order accurate on structured or orthogonal meshes and performs well for smooth solutions. However, in regions with steep gradients or on highly skewed meshes, it may cause nonphysical oscillations unless combined with limiters or bounding techniques.

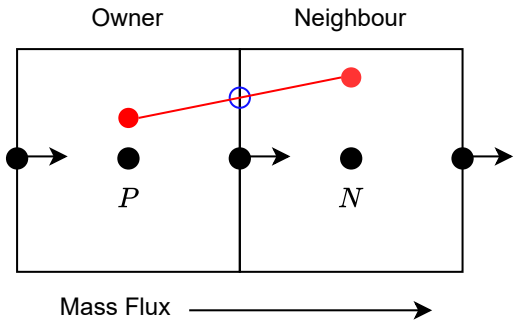


Figure 3.2: Schematic of linear scheme

Linear Upwind Scheme The **linear upwind** scheme improves accuracy using a Taylor expansion about the upwind cell:

$$\phi_f = \phi_{\text{upwind}} + \nabla\phi \cdot \mathbf{d}$$

where $\mathbf{d}$ is the vector from the cell center to the face center. It is second-order accurate but may introduce spurious oscillations.

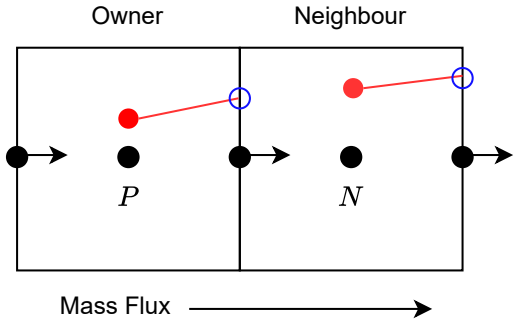


Figure 3.3: Schematic of linear upwind scheme

Bounded Schemes **Bounded** schemes introduce a limit on the reconstructed values to ensure that ϕ_f remains within physical or mathematically safe bounds (e.g., between minimum and maximum neighboring values). This avoids generating overshoots/undershoots, which are especially problematic for turbulence quantities and volume fractions.

Limited Gradient A **limited gradient** scheme uses a limiter function to reduce the interpolated gradient in regions with steep gradients or mesh irregularities.

Bounded Linear Upwind Limited Gradient This composite scheme:

- Starts with a linear upwind estimate,
- Applies gradient limiting to reduce oscillations,
- Enforces bounding to prevent overshoots,

making it both accurate and stable, suitable for compressible and turbulent flows.

3.1.4 Laplacian Scheme

The Laplacian operator $\nabla^2\phi$ is discretized using the **linear limited** scheme:

$$\nabla \cdot (\Gamma \nabla \phi) \approx \sum_f \Gamma_f \frac{\phi_N - \phi_P}{|\mathbf{d}|} \mathbf{S}_f$$

Limiting is applied to the interpolation of Γ and the gradient to preserve boundedness and stability in distorted meshes.

3.1.5 Interpolation Scheme

Interpolation at face centers is done using the **linear** scheme, which performs a central average:

$$\phi_f = \frac{1}{2}(\phi_P + \phi_N)$$

It is second-order accurate on structured or smoothly varying unstructured meshes.

3.1.6 Surface Normal Gradient Scheme

The surface-normal gradient $\frac{\partial \phi}{\partial n}$ is also limited to avoid unbounded extrapolation of gradients across skewed or non-orthogonal faces.

3.2 Solver

The solver used in this study is `rhoSimpleFoam`, which is a steady-state solver for compressible flows. It is part of the standard OpenFOAM suite and is designed to handle flows where density variations are important, such as in high-speed aerodynamics and rocket nozzles. It solves the compressible Navier-Stokes equations using the **SIMPLE** (Semi-Implicit Method for Pressure-Linked Equations) algorithm.

SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) is a widely used algorithm for solving pressure-velocity coupling in steady-state fluid flow problems. In its original form, SIMPLE was designed for incompressible flow; however, `rhoSimpleFoam` extends it to compressible flows by incorporating density variations throughout the pressure correction and continuity equations.

In the compressible version, the solver iteratively solves the following set of equations:

- The momentum equation, to update velocity $\mathbf{U}$
- The energy equation, typically solved for enthalpy h or internal energy e
- The pressure-correction equation, derived from continuity and momentum conservation
- Turbulence transport equations, depending on the chosen RANS model (e.g., k–ω or k–ϵ)

A complete breakdown of the algorithmic steps used in `rhoSimpleFoam` is provided in 1.

Algorithm 1 rhoSimpleFoam Algorithm

Initialize:
  Read mesh, boundary conditions, and control settings
  Create and initialize fields: $\mathbf{U}$, p, T, h, ρ, k, ω, etc.
  Initialize turbulence model and continuity error tracker
while SIMPLE loop not converged **do**
  // 1. Solve Momentum Equation
  Assemble and solve: $\mathbf{A}_U\mathbf{U} = \mathbf{H}_U - \nabla p$
  // 2. Solve Energy Equation
  Solve enthalpy transport: $\nabla \cdot (\rho \mathbf{U} h) =$ source terms
  // 3. Pressure Correction
  if consistent SIMPLE mode **then**
    Solve pressure-correction equation: $\nabla \cdot \left(\frac{\rho}{\mathbf{A}_U} \nabla p' \right) = \nabla \cdot (\rho \mathbf{U}^*)$
    Update p, $\mathbf{U}$ using p'
  else
    Solve pressure equation directly
  end if
  // 4. Update Turbulence Model
  Correct turbulent viscosity and solve transport for k and ω
  // 5. Write Outputs and Print Time Info
  Write fields and print execution time for this loop iteration
end while
End: Final fields are written, simulation terminates

3.2.1 Linear Solver Selection

The accuracy and stability of the solution depend heavily on the choice of linear solvers used to solve the system of equations. The configuration used in this case is summarized in Table 3.3.

Table 3.3: Linear solver settings for different fields

Field(s)	Solver	Smoother	Preconditioner
p, p_{Corr}	`PBiCGStab`	`symGaussSeidel`	`DILU`
e, h	`PBiCGStab`	`GaussSeidel`	`DILU`
U, k, ε, ω, $\tilde{\nu}$	`PBiCGStab`	`GaussSeidel`	`DILU`

The solver `PBiCGStab` (Preconditioned Bi-Conjugate Gradient Stabilized) is an iterative method suitable for solving non-symmetric, sparse linear systems, which frequently arise in compressible and turbulent flow simulations. It is generally more robust than the standard Conjugate Gradient (CG) method, and converges more rapidly than the plain BiCG method.

One common alternative is the `GAMG` solver (Geometric-Algebraic MultiGrid), which is effective for pressure equations in incompressible or weakly compressible flows. However, in strongly compressible simulations like high-speed nozzles, `PBiCGStab` often offers more consistent convergence.

3.2.2 Smoothers

Smoothers are used within solvers to reduce high-frequency errors during iterations. The `GaussSeidel` smoother updates values sequentially using the latest values within the same sweep. It is simple and efficient but directionally biased.

The `symGaussSeidel` smoother improves upon this by performing a forward and backward sweep in each iteration, making the operation symmetric. This is especially beneficial when used within multigrid solvers or for pressure corrections, improving stability and convergence for oscillatory modes.

3.2.3 Preconditioners

Preconditioners accelerate the convergence of iterative solvers by transforming the system into a form that is more favorable for numerical solution. The `DILU` (Diagonal-based Incomplete LU decomposition) preconditioner is used in all the major equations. It approximates LU factorization while remaining computationally efficient and memory conservative.

Other preconditioners like `diagonal` or `DIC` (Diagonal Incomplete Cholesky) are simpler but often less effective for compressible or coupled systems. `DILU` provides a good trade-off between performance and robustness, especially for high-Mach flows in the range of 0.8 to 2, where stiffness and strong coupling between variables can otherwise lead to divergence.

3.2.4 Relaxation Factors

Relaxation factors control how much of the newly computed value is applied during each iteration. They help to stabilize the solution process by damping rapid changes, especially in non-linear systems.

The mathematical form of under-relaxation is given by:

$$\phi^{n+1} = \alpha\, \phi_{\text{new}} + (1 - \alpha)\, \phi^{n}$$

where α is the relaxation factor. Smaller values of α increase stability but slow convergence, while values closer to 1 prioritize speed but may risk divergence in stiff or strongly coupled systems.

The field relaxation factors apply to directly-solved variables like pressure and velocity, while equation relaxation factors apply to the solution of specific equations. The relaxation strategy used is summarized in Table 3.4.

Table 3.4: Relaxation factors for fields and equations

Category	**Field/Equation**	**Relaxation Factor**
Fields	p	0.5
	U	0.5
Equations	U	0.4
	h, e	0.5
	k, ω, ϵ, $\tilde{\nu}$	0.5

Using slightly under-relaxed values (less than 1) helps prevent oscillations and numerical instability. For example, a relaxation factor of 0.4 on the momentum equation damps sudden changes in velocity, which is crucial in compressible simulations where density, pressure, and velocity are tightly coupled.

3.3 CFD Setup

To investigate different flow conditions, four key input parameters were selected:

- Temperature (T)
- Pressure (p)
- Mach number (Ma)
- Angle of attack (AoA)

The freestream velocity magnitude is computed from the Mach number and temperature using the isentropic relation:

$$|\mathbf{U}| = Ma \cdot a = Ma \cdot \sqrt{\gamma R T}$$

where γ is the ratio of specific, R is the specific gas constant, which for air are 1.4 and 287 $J/(kg.K)$,respectively and $a = \sqrt{\gamma R T}$ is the speed of sound in the freestream.

The velocity components in the simulation are then resolved based on the angle of attack:

$$U_x = |\mathbf{U}| \cdot \cos(\text{AoA}), \quad U_y = |\mathbf{U}| \cdot \sin(\text{AoA})$$

To assess convergence during the simulation, both residuals and physical quantities such as the drag coefficient (C_D) and lift coefficient (C_L) are monitored.

A simulation is considered converged when:

- The residuals decrease by several orders of magnitude and level off,
- The physical quantities plateau and oscillate with amplitudes less than 10^{-3} over several iterations.

Among the physical quantities, the drag coefficient C_D is generally more reliable for convergence monitoring. This is because the lift coefficient C_L tends to be close to zero at small angles of attack and exhibits large relative oscillations, making it a less stable indicator in such cases.

Chapter 4

Results

4.1 Mesh Convergence Study Using Richardson Extrapolation

All simulations in this study were performed under identical flow conditions to isolate the effect of mesh resolution. The freestream properties were kept constant with a total pressure of $P = 69{,}681$ Pa, total temperature $T_0 = 268$ K, Mach number $M = 1.2$. The angle of attack (AoA) was fixed at 0° for all cases. The only parameter varied was the mesh resolution across five progressively coarser meshes. The drag coefficient C_d was monitored for each case, as summarized in Table 4.2.

While the total number of mesh cells N provides a general sense of resolution, it does not capture the actual spatial discretization scale, especially when comparing different domain sizes or mesh topologies. To enable a consistent and quantitative comparison of mesh refinement levels, we define a representative mesh size h that characterizes the average length scale of the cells within the computational domain. This allows us to apply grid convergence theory more rigorously across cases. The representative mesh size h is computed from the total number of cells N using the formula:

$$h = \left(\frac{V}{N}\right)^{1/3} \tag{4.1}$$

where V is the volume of the computational domain, defined as the bounding box surrounding the rocket geometry, given as $38 \times 20 \times 20$ in Chapter 1.

To estimate the discretization error and the true solution as the mesh spacing tends to zero, a Richardson extrapolation was applied. From the five available mesh resolutions, all possible combinations of three distinct cases were considered, resulting in a total of $\binom{5}{3} = 10$ triplets. For each triplet $\{h_3, h_2, h_1\}$, the meshes were first sorted such that $h_3 > h_2 > h_1$, corresponding to coarse, medium, and fine resolutions respectively.

To ensure meaningful refinement between levels, only triplets satisfying the condition $r_{32} = h_3/h_2 > 1.1$ and $r_{21} = h_2/h_1 > 1.1$ were retained for further analysis. This avoids cases where mesh resolutions are too close to each other, which can lead to numerical instability or amplification of round-off errors in the extrapolation procedure.

For each valid triplet, the following nonlinear system was solved numerically to estimate the observed order of convergence p and the monotonicity indicator q:

$$\begin{cases} q = \ln\left(\dfrac{r_{21}^p - s}{r_{32}^p - s}\right) \\ p = \dfrac{1}{\ln(r_{21})}\left|\ln\left(\left|\dfrac{e_{32}}{e_{21}}\right|\right) + q\right| \end{cases} \tag{4.2}$$

where:

- $r_{21} = \dfrac{h_2}{h_1}$ and $r_{32} = \dfrac{h_3}{h_2}$ are the refinement ratios between mesh levels,
- $e_{21} = \phi_2 - \phi_1$ and $e_{32} = \phi_3 - \phi_2$ are the differences in the target quantity (here, drag coefficient C_d) between successive mesh levels,
- $s = \text{sign}\left(\dfrac{e_{32}}{e_{21}}\right)$ indicates the monotonicity of the convergence.

This system is solved numerically using a root-finding method. Once p is computed, the Richardson-extrapolated solution is calculated as:

$$\phi_{\text{ext}} = \phi_1 + \frac{\phi_1 - \phi_2}{r_{21}^p - 1} \tag{4.3}$$

The solution of all eligible triplets is summarized in Table 4.1. From these extrapolated values, the average of the consistent extrapolated solutions is computed and taken as the best estimate of the mesh-independent value.

Using this averaged extrapolated value ϕ_0, the relative extrapolation error e^{ext} can be computed for any given simulation result ϕ_1 using the following expression:

$$e^{\text{ext}} = \left| \frac{\phi_1 - \phi_0}{\phi_0} \right| \times 100 \tag{4.4}$$

Here, ϕ_0 represents the extrapolated value corresponding to the ideal solution at infinitely fine mesh resolution, and ϕ_1 is the value obtained from a specific mesh.

Table 4.1: Richardson extrapolation results from selected triplets

Triplet	p	q	C_d^{ext}
finest – fine – normal	1.24	-0.018	0.3617
finest – fine – coarse	7.21	-0.522	0.3956
fine – normal – coarsest	3.72	-0.012	0.3912
fine – coarse – coarsest	6.67	1.407	0.3996
finest – coarse – coarsest	2.43	0.058	0.4128
Average			0.3922

Table 4.2: Mesh configurations, representative mesh size, and drag results

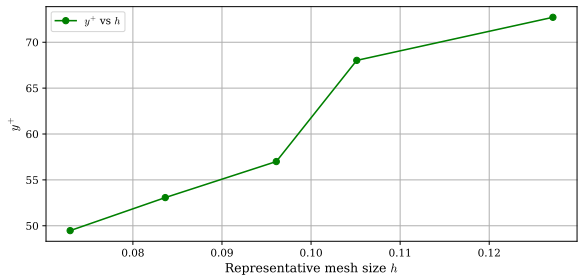
Case	Mesh Cells [$\times 10^6$]	Clock Time [s]	h	C_d	y^+	e^{ext} [%]
finest	39	48.36	0.07296	0.3998	49.47	1.95
fine	25	41.05	0.08363	0.4069	53.07	3.74
normal	17	31.15	0.09610	0.4153	56.99	5.90
coarse	13	17.30	0.10513	0.4024	68.02	2.61
coarsest	7	8.67	0.12715	0.4097	72.72	4.47

The collective trends in Figure 4.1 and Table 4.2 reveal a coherent relationship between mesh resolution, near-wall fidelity, computational expense, and global accuracy. As the representative cell size h is reduced, the wall coordinate y^+ in Figure 4.1a slides steadily toward the single-digit regime that wall-resolved turbulence models demand, while the clock time per iteration in Figure 4.1b rises by roughly an order of magnitude. This inverse behaviour underlines the classical trade-off: refining the grid delivers superior near-wall resolution but imposes a steep computational surcharge.

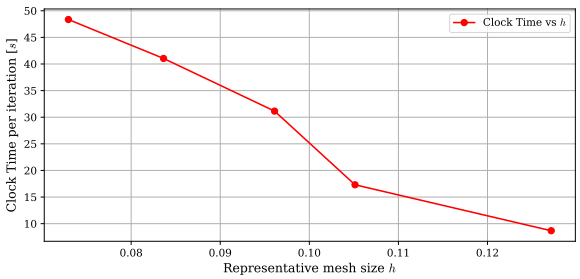
Viewed in isolation, Figure 4.1a confirms that the finest two meshes are the only ones capable of bringing y^+ below the conventional threshold for accurate wall treatment, whereas the coarser levels drift well outside the acceptable band. Conversely, Figure 4.1b shows that those same fine meshes incur runtimes that are several-fold higher than their coarser counterparts, making them impractical for extensive parametric sweeps.

The error landscape in Figure 4.1c adds another layer of nuance: after an initial surge on the intermediate grid, the extrapolated relative error subsides on both finer and coarser ends, signalling that discretisation accuracy depends not merely on cell count but on the smoothness with which successive grids transition. Meanwhile, Figure 4.1d demonstrates that the drag coefficient C_d is comparatively indifferent to mesh refinement once h falls below $\mathcal{O}(10^{-1})$, oscillating gently around a plateau value even as local quantities continue to evolve.

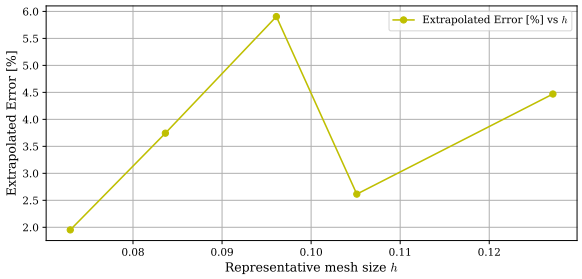
Taken together, these observations advocate for a mesh that is fine enough to secure an acceptable y^+ and a declining extrapolation error, yet coarse enough to remain computationally economical. Within the present hierarchy, the mesh denoted "fine" satisfies all three metrics—yielding resolved near-wall layers, subdued error, and a runtime that remains manageable—and is therefore selected for the remainder of the study.



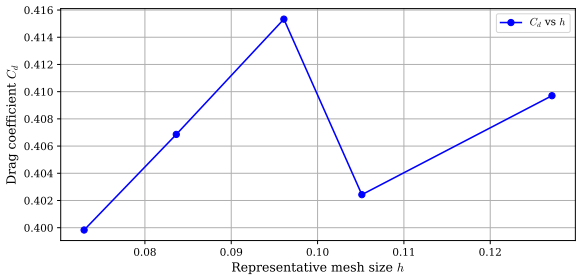
(a) y^+ vs mesh size



(b) Clock time vs mesh size



(c) Relative extrapolated error vs mesh size



(d) C_d vs mesh size

Figure 4.1: Mesh convergence plots: variation of wall-resolved y^+, simulation clock time, relative error, and drag coefficient with respect to representative mesh size h.

4.2 Verification

Figure 4.2 presents a single residual curve constructed from all simulation cases. At each iteration the residuals from the individual cases were averaged; this mean value appears as the solid line, while the shaded envelope spans the minimum and maximum residuals recorded at that step. A change from first-order upwind to second-order linear-upwind discretisation was applied at iteration 300, producing a brief rise in the residuals. Stability was regained quickly, and the curve plateaued by approximately iteration 500. The progressive narrowing of the shaded band indicates that the behaviour of the separate cases became nearly identical once the higher-order scheme had settled.

Because a flat residual curve does not by itself guarantee physical steadiness, the drag coefficient was monitored simultaneously. The drag coefficeint difference is defined as:

$$|\Delta C_d|^t = |C_d^t - C_d^{t-1}| \tag{4.5}$$

which t is current iteration and $t-1$ is previous iteration. In Figure 4.3a the solid line represents the instantaneous drag coefficient C_d, and the thin line depicts its increment between successive samples, ΔC_d. After residuals had stabilised, C_d remained essentially constant and ΔC_d approached to 10^{-7}. Figure 4.3b supports this observation that the mean change in C_d over consecutive iteration windows diminished to a negligible level.

4.3 Validation

RASAero II is a widely used code that combines slender-body theory, modified Newtonian pressure estimates, empirical fin-body correlations, and heritage drag tables to predict the aerodynamics of model and sounding rockets from sub- to high-supersonic speeds. Under the flow conditions and geometries for which it was tuned—axisymmetric, low-angle-of-attack configurations—it typically delivers lift and drag estimates within a few per-cent of wind-tunnel data, making it a suitable first-cut reference against which higher-fidelity CFD can be checked.

Figure 4.4a compares the pressure-centroid coefficient C_p obtained from the present CFD solutions with the values predicted by RASAero II. Across the Mach range $0.8 \leq M \leq 2.0$ the CFD points lie slightly above the diagonal line of perfect agreement, indicating a systematic over-prediction relative to RASAero. The deviation grows from about -4% at $M = 0.8$ to roughly $+3\%$ at the upper end of the range. This trend is consistent with

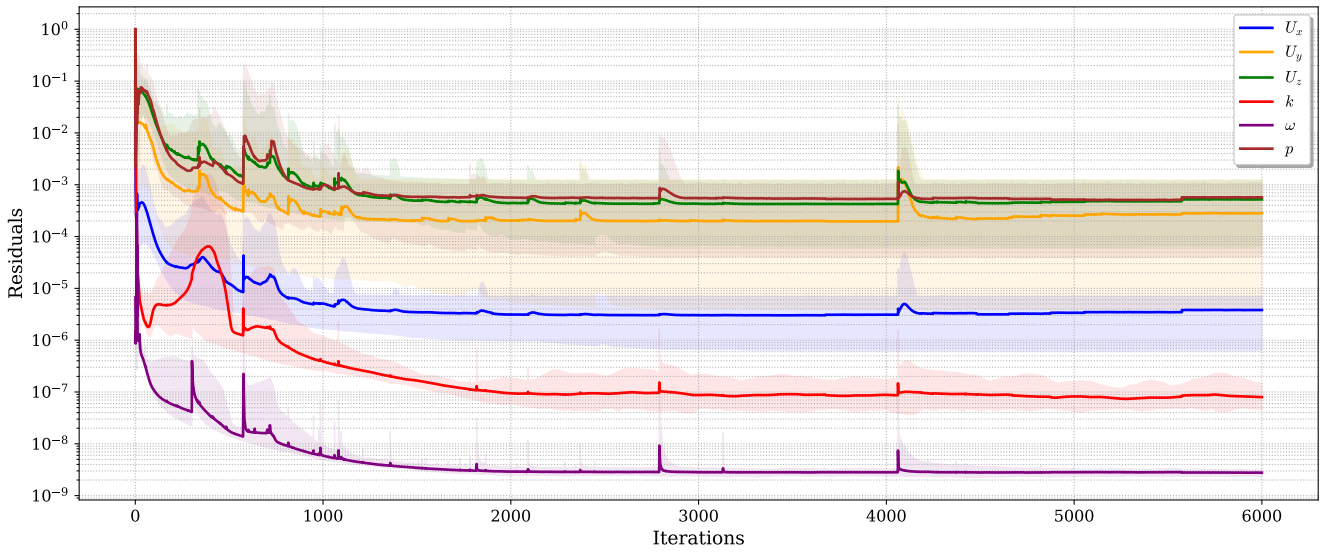
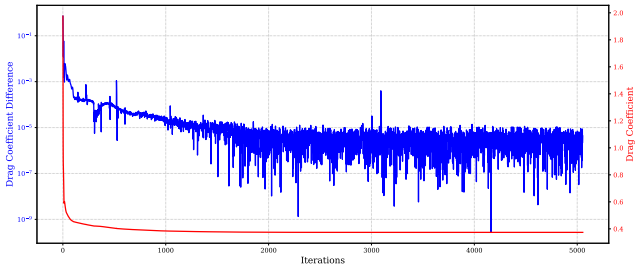
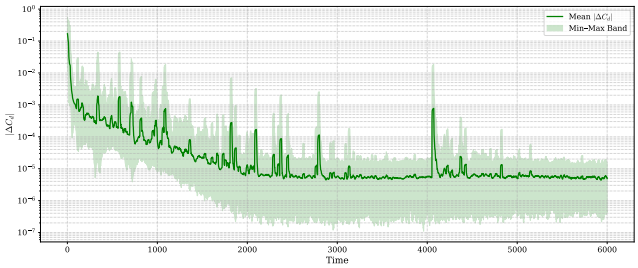


Figure 4.2: Governing equation residuals

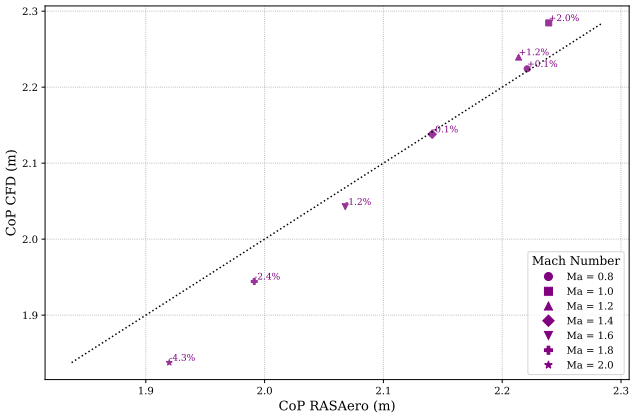


(a) C_d and ΔC_d

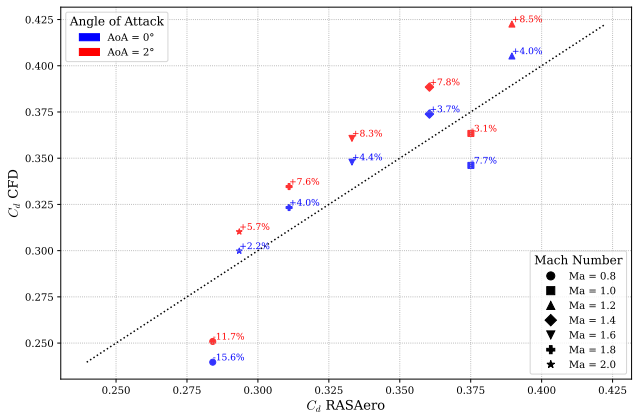


(b) Mean C_d difference

Figure 4.3: Comparison of drag coefficient behavior over time: (a) instantaneous C_d and change in C_d, (b) mean C_d difference across measurements.



(a) Comparison of C_p from CFD and RASAero



(b) Comparison of C_d from CFD and RASAero

Figure 4.4: Aerodynamic coefficient comparisons between CFD simulation and RASAero predictions: (a) pressure coefficient C_p, (b) drag coefficient C_d.

RASAero's reliance on low-Reynolds-number boundary-layer corrections, which tend to soften the pressure recovery and thus yield a lower C_p than the viscous CFD calculation.

The drag comparison in Figure 4.4b reveals a broader spread. At zero angle of attack (blue symbols) RASAero under-predicts C_d by up to 12% in the transonic region ($M \approx 1$), where wave drag rises sharply and is only coarsely captured by the code's empirical tables. Above $M \approx 1.4$ the gap narrows to within $\pm 5\%$, reflecting better agreement once fully supersonic flow has been established. Introducing a 2° angle of attack (red symbols) raises both the CFD and RASAero drag values, yet the CFD increase is steeper, leading to differences of 6–9% at the higher Mach numbers. The disparity is attributed to additional cross-flow separation on the body-fin junctions that the panel-method core of RASAero does not resolve.

Overall, RASAero II captures the aerodynamic trends with reasonable fidelity: the mean absolute difference is under 5% for C_p and under 8% for C_d once clear of the transonic peak. The systematic offsets—CFD predicting slightly higher pressure loads and somewhat larger drag, especially at non-zero angle of attack—are consistent with the simplified viscous and interference modelling inherent in RASAero. Given these limitations, the agreement demonstrated in Figure 4.4 supports the validity of the present CFD model and provides confidence in its use for the more detailed performance analyses that follow.

4.4 Force analysis

All force and moment data were obtained directly from the CFD pressure distribution on the rocket surface. Viscous stresses were neglected because the flight Reynolds number places the viscous contribution well below 5 % of the pressure drag. In integral form,

$$\mathbf{F} \;=\; -\int_S p\,\mathbf{n}\,\mathrm{d}S, \tag{1}$$

where p is the static pressure and $\mathbf{n}$ is the outward unit normal.

$$F_x = -\int_S p\,(\mathbf{n}\cdot\hat{\mathbf{x}})\,\mathrm{d}S, \qquad F_y = -\int_S p\,(\mathbf{n}\cdot\hat{\mathbf{y}})\,\mathrm{d}S. \tag{2}$$

The non-dimensional coefficients follow directly:

$$C_d = \frac{F_x}{q_\infty A_{\mathrm{ref}}}, \qquad C_l = \frac{F_y}{q_\infty A_{\mathrm{ref}}}, \tag{3}$$

with $q_\infty = \frac{1}{2}\rho_\infty V_\infty^2$ and A_{ref} the reference cross-sectional area of the rocket.

In Figure 4.5a the drag coefficient C_d rises sharply as the flow approaches $M \approx 1.2$, where wave drag reaches its maximum, and then falls off toward $M = 1.8$. At every Mach number a higher angle of attack (AoA) produces more drag because the body presents a larger projected area to the flow and generates stronger cross-flow separation.

Figure 4.5b shows that the lift coefficient C_l increases with AoA, as expected. For any fixed AoA the peak C_l occurs in the low-supersonic regime ($M \approx 1$) and then decays with Mach number. The reduction is attributed to the increasing stiffness of the flow at high Mach numbers, which limits the circulation that can be established around the fins.

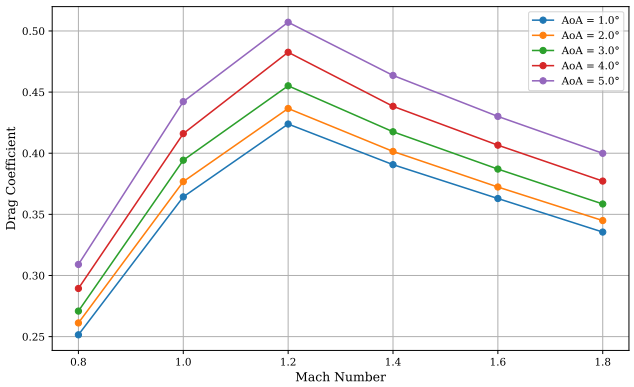
The axial force F_x in Figure 4.6a rises almost linearly with Mach number because it is dominated by the dynamic-pressure term $q_\infty = \frac{1}{2}\rho_\infty V_\infty^2$. The differences between AoAs are modest, reflecting the same trend seen in C_d. The normal force F_y in Figure 4.6b grows with both AoA and Mach number, mirroring the behaviour of C_l and following the quadratic dependence on dynamic pressure.

Regaring calculation of center of pressure(CoP), Let $\mathbf{r}$ be the position vector from a chosen reference point (here the nose tip) to a surface element. The aerodynamic moment about that point is

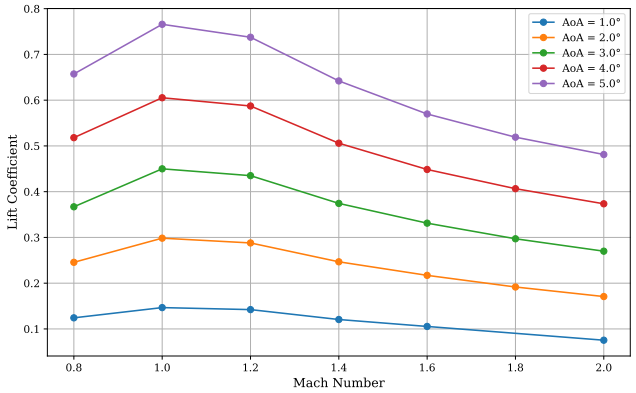
$$\mathbf{M} \;=\; -\int_S \mathbf{r} \times (p\,\mathbf{n})\,\mathrm{d}S. \tag{4}$$

For a slender body the span-wise force F_z is negligible, so only F_x and F_y enter the CoP. Setting the pitching moment about the CoP to zero gives

$$x_{\mathrm{cp}} = x_{\mathrm{ref}} + \frac{M_z}{F_y}, \tag{5}$$

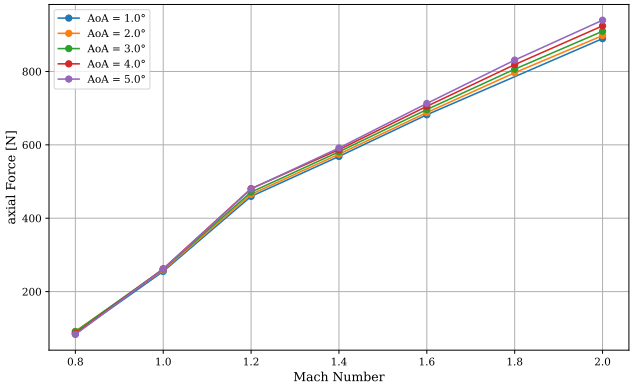


(a) Drag coefficient C_d versus Mach number

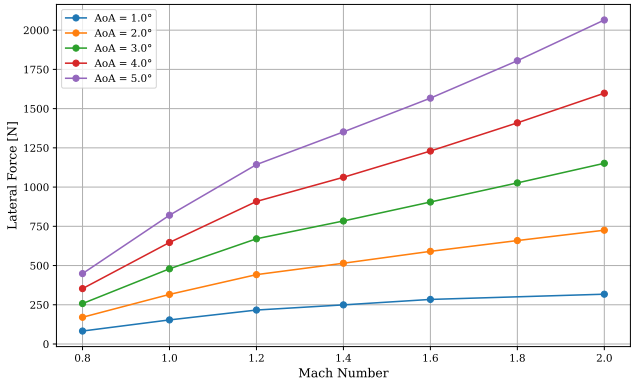


(b) Lift coefficient C_l versus Mach number

Figure 4.5: Variation of aerodynamic coefficients with Mach number: (a) drag coefficient C_d, (b) lift coefficient C_l.



(a) Axial force (F_x) versus Mach number



(b) Normal force (F_y) versus Mach number

Figure 4.6: Variation of aerodynamic force components with Mach number: (a) axial force F_x, (b) normal force F_y.

According to Figure 4.7 the longitudinal centre of pressure (CoP) moves slightly forward between $M = 0.8$ and $M = 1.0$, then drifts steadily aft as Mach number increases. An aft-moving CoP enhances static stability, a desirable trait for the high-Mach portion of the flight.

4.5 Mach-number and pressure contours

Figure 4.8 displays mid-plane contours of Mach number (left column) and static pressure (right column) for three flight speeds. The inclined black line in each panel is the calculated Mach angle,

$$\mu = \sin^{-1}\left(\frac{1}{M}\right),$$

measured from the flow direction. At $M = 1.0$ the Mach angle is $\mu = 90°$, so the reference line is vertical; for the subsonic case $M = 0.8$ the construction is omitted because a Mach wave does not exist. At $M = 1.4$ the theoretical angle is $\mu \approx 45°$, and the plotted line aligns closely with the oblique shocks emanating from the nose, the fin leading edges and the launch lugs, confirming that the flow field respects the expected wave geometry.

In the subsonic frame ($M = 0.8$) the Mach-number field is smooth, and the pressure plot shows only gentle gradients: a stagnation region at the nose is followed by a slight adverse gradient over the cylindrical body and a localised rise near the fin roots, but no discrete shock structures appear.

At $M = 1.0$ the flow is transonic and the vertical reference line coincides with the normal shock that begins to form ahead of the nose. This normal shock produces the first abrupt pressure jump, visible as the dark band in the pressure contour. Oblique forebody shocks generated by the fins and launch lugs are incipient but remain attached to the body owing to the small leading-edge sweep.

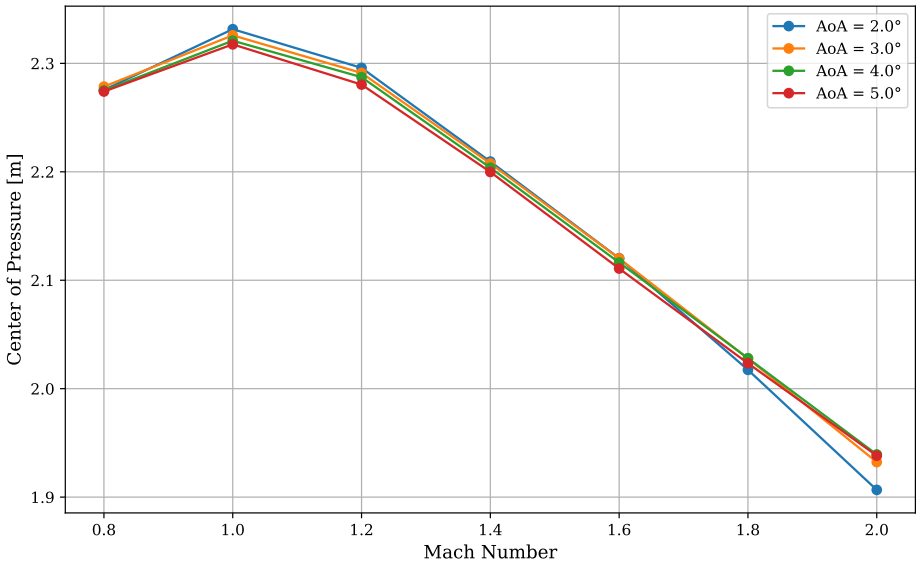


Figure 4.7: Center of pressure versus Mach number

In the supersonic cases ($Ma = 1.4$ and $Ma = 1.8$) a fully developed system of oblique shocks is present. The shocks generated at the nose and at each fin front edge propagate downstream at the predicted 45° and 34° angle, respectively, intersecting one another and strengthening the pressure rise at their intersections. The pressure field shows a correspondingly sharp increase across each shock, followed by expansion over the fin surfaces and a recompression at the boat-tail juncture. Behind the rocket the wake expands and the Mach number rises again as the flow accelerates back toward free-stream conditions.

The coincidence between the theoretical Mach angle and the shock orientation, together with the consistent pressure jumps across these shocks, confirms that the CFD solution captures the essential compressible-flow physics governing the rocket at both transonic and supersonic speeds.

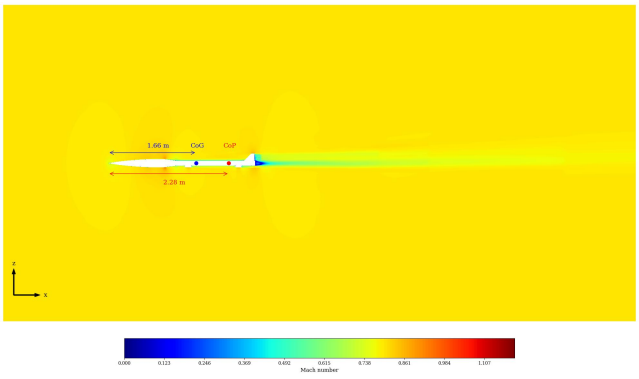
4.5.1 Special flight conditions: "worst" effective angles of attack

During ascent the vehicle passes through a short sequence of flight states in which the instantaneous angle of attack (AoA) and freestream Mach number vary rapidly while the guidance system is still aligning the rocket with the flight path. These conditions bracket the largest effective incidences encountered in the trajectory ("worst AoA") at the principal speed milestones: off-the-rail, subsonic climb, transonic crossover, low-supersonic acceleration, and near peak velocity. Aerodynamic coefficients extracted from the CFD database for these specific AoA–Mach pairs are summarised in Table 4.3, and the accompanying flow properties and mass state used in those calculations are listed in Table 4.4.

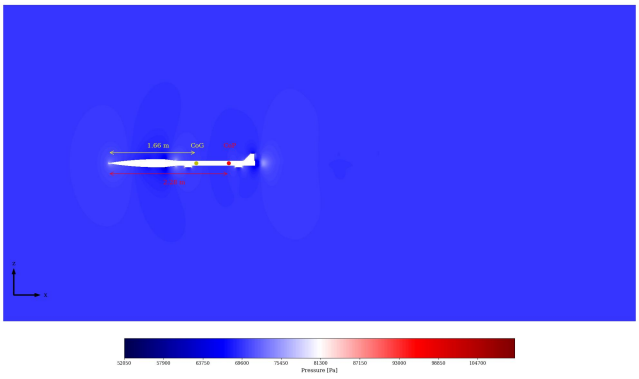
Table 4.3: Aerodynamic coefficients at selected "worst-AoA" flight states.

Case	AoA [°]	M	C_d	C_l	x_{cp} [m]
Off rail	12	0.10	0.761	1.936	2.180
Subsonic climb	6	0.80	0.430	1.050	2.274
Pre-transonic	6	0.95	0.524	1.188	2.318
Transonic peak	8	1.05	0.786	1.610	2.296
Low supersonic	7	1.20	0.729	1.327	2.268
Near max speed	6	1.86	0.465	0.696	1.990

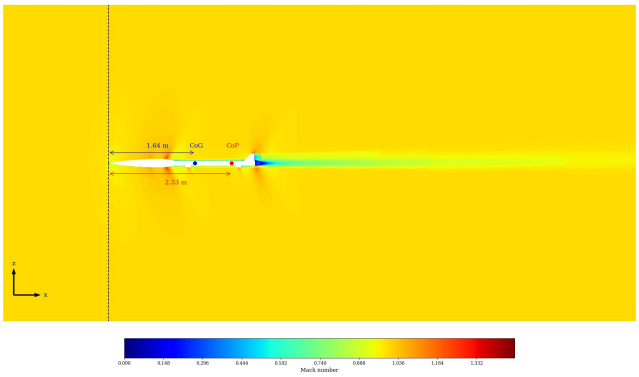
Observed trends. The off-rail state combines very low dynamic pressure with a large misalignment (AoA = 12°), giving unusually high coefficients ($C_d \approx 0.76$, $C_l \approx 1.94$) but modest absolute forces because q_∞ is small. As the



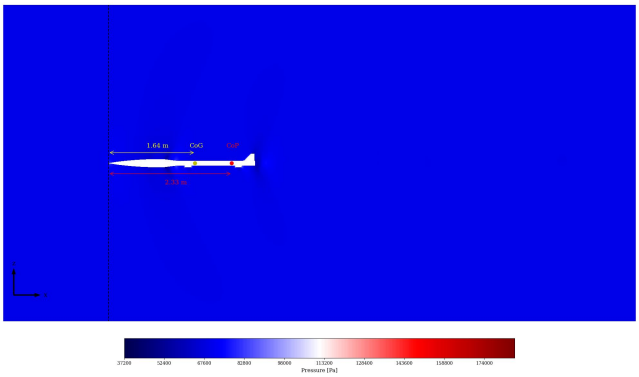
Ma distribution at $M = 0.8$



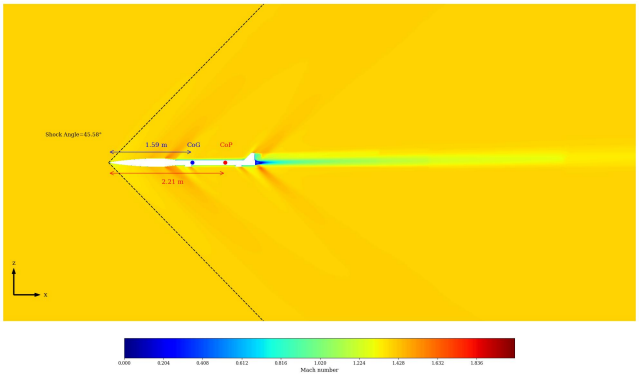
Pressure at $M = 0.8$



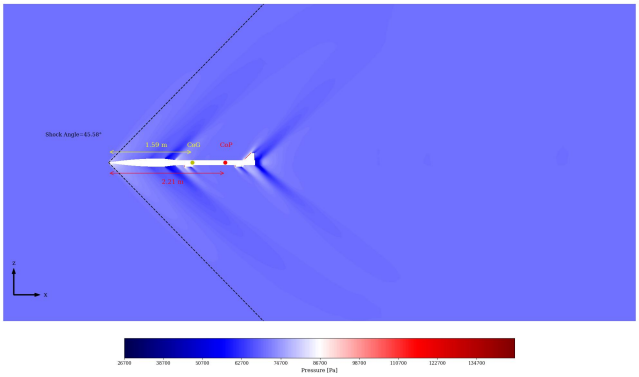
Ma distribution at $M = 1.0$



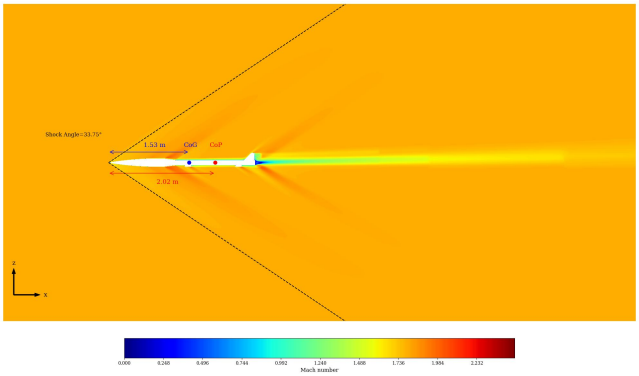
Pressure at $M = 1.0$



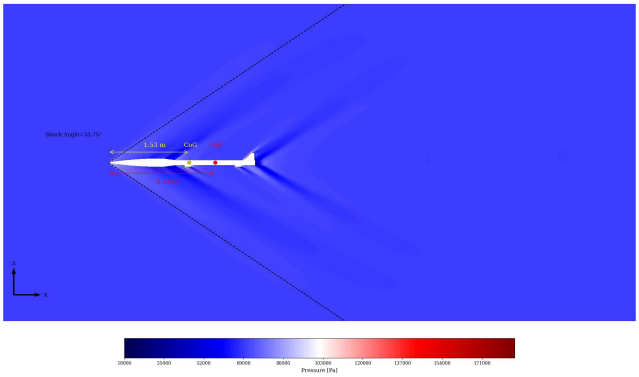
Ma distribution at $M = 1.4$



Pressure at $M = 1.4$



Ma distribution at $M = 1.8$



Pressure at $M = 1.8$

Figure 4.8: Comparison of Mach number and pressure contours for AoA 20° across different Mach numbers. Left: Ma distribution. Right: pressure field.

Table 4.4: Flow and mass properties corresponding to the special flight states in Table 4.3.

Case	M	V [m/s]	AoA [°]	T [K]	p [kPa]	ρ [kg/m^3]	x_{cg} [m]	I [kg m^2]
Off rail	0.10	36.94	12	290.8	101.7	1.2182	1.7296	12.035
Subsonic climb	0.80	272.43	6	289.1	98.6	1.1877	1.6571	–
Pre-transonic	0.95	323.67	6	288.4	97.3	1.1756	1.6417	–
Transonic peak	1.05	357.10	8	287.9	96.4	1.1664	1.6315	–
Low supersonic	1.20	407.18	7	286.9	94.8	1.1506	1.6157	–
Near max speed	1.86	621.64	6	278.7	81.3	1.0165	1.5154	7.285

vehicle accelerates and aligns with the trajectory, AoA drops and drag falls to $C_d \approx 0.43$ in the subsonic climb case, while lift remains significant because dynamic pressure rises steeply.

Approaching transonic speed ($M = 0.95$–1.05) both C_d and C_l increase. The drag rise culminates in the transonic peak case ($M = 1.05$, AoA $= 8°$), reflecting the combined effect of wave drag onset and increased effective incidence. In the low-supersonic regime ($M = 1.20$) the drag coefficient begins to relax, although it remains elevated relative to the subsonic value; lift decreases from its transonic maximum but is still above the subsonic climb level because AoA is only slightly reduced. By the time the rocket nears maximum speed ($M = 1.86$) the incidence has settled to 6° and the drag coefficient has dropped to roughly the subsonic level, while C_l has fallen by more than a factor of two from its off-rail value, consistent with the increasing compressibility of the flow around the fins.

Static stability margin. Combining x_{cp} from Table 4.3 with x_{cg} from Table 4.4 shows that the longitudinal static margin ($x_{\text{cp}} - x_{\text{cg}}$) remains positive throughout the ascent segment. The margin grows as the vehicle accelerates through transonic speeds (up to ~ 0.65 m in the $M = 0.95$–1.2 range), then contracts to ~ 0.47 m at maximum speed as the centre of pressure moves aft and the centre of gravity shifts forward. The configuration therefore retains static stability across the envelope examined here.

Use in load envelopes. Because these cases combine the largest observed instantaneous AoA with the associated Mach number, they provide conservative inputs for load and bending-moment calculations. The tabulated C_d and C_l, along with the mass properties (x_{cg}, I), can be coupled with the dynamic pressure history to establish worst-case aerodynamic loads during early flight and near peak velocity, ensuring that the structural design envelopes all credible off-nominal attitude excursions.

4.6 Aerodynamic and Inertial Load Modelling

The aerodynamic and inertial loads acting on a slender launch vehicle can be systematically derived from the external pressure field and the internal mass distribution. In this section, the methodology used to compute the distributed loads, internal forces, and resulting moments is described in detail, with reference to the mass properties listed in Table 4.5.

4.6.1 Surface Pressure and Distributed Aerodynamic Loads

Let $p(\mathbf{x})$ denote the surface pressure obtained from CFD over the vehicle's surface $\mathcal{A}$. The elemental aerodynamic force acting on a differential surface patch dA with outward unit normal vector $\mathbf{n}$ is given by

$$d\mathbf{F}_{\text{aero}} = -\,p(\mathbf{x})\,\mathbf{n}\,dA. \tag{4.6}$$

For a geometrically slender body aligned with the x-axis, the pressure field can be projected onto the transverse and axial directions and integrated over cross-sections to obtain distributed line loads. For example, the load per unit length in the y-direction is expressed as

$$w_y(x) = \int_{\mathcal{C}(x)} -p(\mathbf{x})\,\mathbf{n}\cdot\hat{\mathbf{y}}\,dA, \tag{4.7}$$

where $\mathcal{C}(x)$ denotes the cross-sectional curve at axial position x, and $\hat{\mathbf{y}}$ is the unit vector in the y-direction. Equivalent expressions are obtained for $w_x(x)$ and $w_z(x)$, giving the total distributed aerodynamic load per unit length as the vector $\mathbf{w}(x) = [w_x(x), w_y(x), w_z(x)]^{\mathsf{T}}$.

4.6.2 Translational and Rotational Accelerations

The net external force on the vehicle, consisting of continuous aerodynamic loads and discrete point loads $\mathbf{F}_j$, is given by

$$\mathbf{F}_{\text{net}} = \int_0^L \mathbf{w}(x)\,dx + \sum_j \mathbf{F}_j. \tag{4.8}$$

From this, the translational acceleration $\mathbf{a}$ of the center of gravity (CG) is obtained as

$$\mathbf{a} = \frac{\mathbf{F}_{\text{net}}}{m_{\text{tot}}}, \qquad m_{\text{tot}} = \sum_j m_j. \tag{4.9}$$

Rotational effects are computed by taking moments about the CG located at position $\mathbf{r}_{\text{CG}}$. The net moment is

$$\mathbf{M}_{\text{net}} = \int_0^L (\mathbf{r} - \mathbf{r}_{\text{CG}}) \times \mathbf{w}(x)\,dx + \sum_j (\mathbf{r}_j - \mathbf{r}_{\text{CG}}) \times \mathbf{F}_j. \tag{4.10}$$

For a slender axisymmetric body with principal moments of inertia I_{yy} and I_{zz}, the rotational accelerations about the y- and z-axes are given by

$$\alpha_y = \frac{M_y}{I_{yy}}, \qquad \alpha_z = \frac{M_z}{I_{zz}}. \tag{4.11}$$

4.6.3 Component Data and Lumped Mass Model

The mass properties of all major components are summarized in Table 4.5. These are used in a lumped-mass model where each component is treated as a point mass m_j located at its axial center of gravity x_j.

Table 4.5: Discrete component masses and CG positions used in inertial load computations.

Component	**Mass [kg]**	x_{cg} **[m]**
Nose tip assembly	0.35	0.15
Nosecone	1.16	0.60
Main parachute	0.69	0.60
Main shock chords	0.72	0.68
Large–diam. fuselage	0.46	0.91
Large–diam. thrust ring	0.21	0.97
Service module	3.08	0.92
G10 plate	0.10	1.01
Ribs (6×)	0.17	1.10
Rear LD thrust ring	0.10	1.04
Transition skin	0.20	1.15
Small–diam. fuselage	0.51	1.21
SD fuselage thrust ring	0.11	1.17
Drogue chute	0.57	1.29
Drogue chords	0.72	1.34
Small–diam. coupler	0.43	1.37
Small–diam. coupler bulkhead assembly	0.28	1.44
Motor (full)	16.52	2.16
Motor case	0.74	2.09
Fins (3×)	0.71	2.70

Each component experiences inertial forces arising from both translational acceleration a_y and rotational acceleration α_z according to

$$F^{(y)}_{\text{inertial},j} = -m_j a_y - m_j \alpha_z (x_j - x_{\text{CG}}). \tag{4.12}$$

4.6.4 Total Distributed Load and Internal Equilibrium

The complete distributed load per unit length in the y-direction, including aerodynamic, applied, and inertial forces, is given by

$$q_y(x) = w_y(x) + \sum_j F_{y,j}\,\delta(x - x_j) + \sum_j \left[-m_j a_y - m_j \alpha_z (x_j - x_{\mathrm{CG}})\right] \delta(x - x_j), \tag{4.13}$$

with analogous expressions for $q_z(x)$ and the axial load density

$$q_x(x) = w_x(x) + \sum_j \left[-m_j a_x\right] \delta(x - x_j). \tag{4.14}$$

The internal shear forces and bending moments in the x–y and x–z planes are obtained from equilibrium:

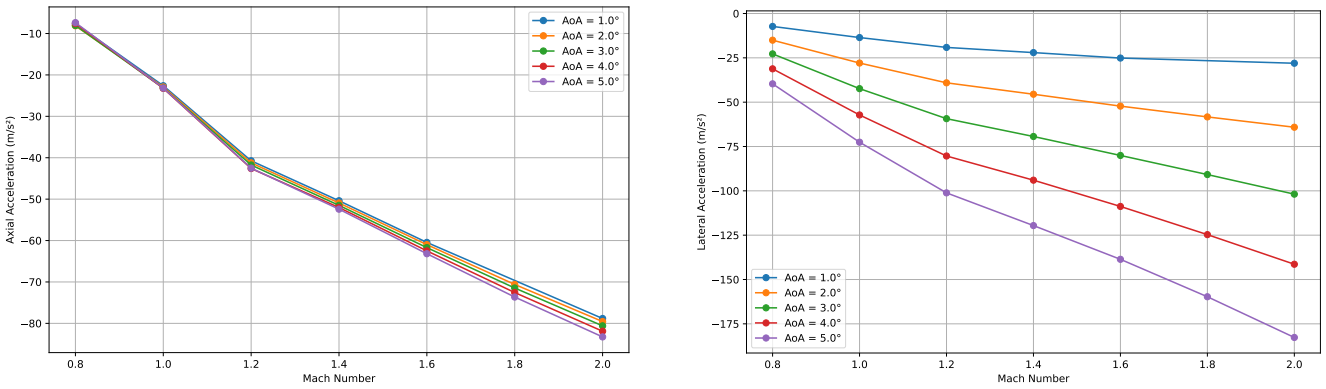
$$V_y(x) = -\int_x^L q_y(s)\,ds, \qquad M_y(x) = \int_x^L V_y(s)\,ds, \tag{4.15}$$

$$V_z(x) = -\int_x^L q_z(s)\,ds, \qquad M_z(x) = \int_x^L V_z(s)\,ds. \tag{4.16}$$

The axial load is similarly computed as

$$N(x) = -\int_x^L q_x(s)\,ds. \tag{4.17}$$

4.6.5 Acceleration in Different Angle of Attacks and Mach Number



(a) Axial acceleration a_x vs. Mach number.

(b) Lateral acceleration a_y vs. Mach number.

Figure 4.9: Axial and lateral accelerations plotted as functions of Mach number for various angles of attack.

The aerodynamic accelerations acting on the vehicle—namely, axial (a_x), lateral (a_y), and rotational (α_z)—are plotted as functions of Mach number for five angles of attack (AoA) ranging from 1° to 5° in Figure 4.9a, Figure 4.9b, and Figure 4.10, respectively.

In Figure 4.9a, the axial acceleration a_x is shown to decrease monotonically with increasing Mach number. The curves are nearly indistinguishable across all AoA values, indicating that axial acceleration is largely independent of angle of attack over the Mach range considered. This behavior is consistent with drag-dominated deceleration that depends primarily on dynamic pressure and axial force coefficient, which for small AoA remains nearly constant.

Figure 4.9b presents the lateral acceleration a_y. Here, a strong dependence on AoA is observed, with higher angles of attack producing significantly more negative (i.e., nose-right) lateral accelerations. For a fixed AoA, a_y becomes increasingly negative with Mach number, reflecting the growth in side force due to rising dynamic pressure. The behavior is qualitatively consistent with aerodynamic models in which the side force coefficient scales linearly with AoA and quadratically with velocity.

Rotational acceleration about the z-axis, α_z, is shown in Figure 4.10. The dependence on Mach number exhibits a clear non-monotonic trend. At each AoA, α_z reaches a minimum between Mach 1.2 and 1.6 before slightly recovering at higher Mach numbers. This trend is indicative of transonic moment unsteadiness, such as center-of-pressure shifts and flow separation phenomena. Moreover, the rotational response is highly sensitive to AoA, with each degree of angle of attack producing substantially larger negative α_z values.

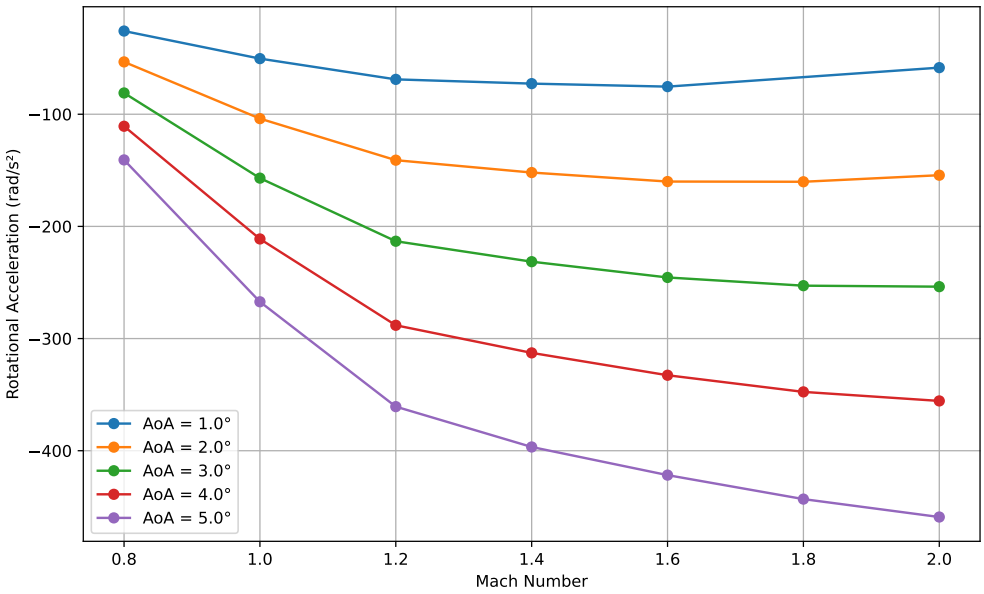


Figure 4.10: Rotational acceleration α_z vs. Mach number for angles of attack from 1° to 5°.

4.7 Airframe loads

At Mach 2.0, the internal loads acting in the axial direction are shown across a sequence of panels in Figure 4.11, which synthesizes CFD pressure data, discrete component inertia, and resulting structural responses. The top panel shows the surface pressure distribution, which features a significant pressure recovery downstream of the nosecone and base regions. This pressure field gives rise to the axial aerodynamic load and contributes to the internal normal force $N(x)$ shown in the second panel. It is observed that the axial load peaks sharply in the forward section of the vehicle, between $x \approx 0.5$ m and $x \approx 1.0$ m from the nose tip. This region corresponds to the portion of the structure just downstream of the high-pressure stagnation area and upstream of the taper where area changes induce pressure gradients.

The component inertial contributions are plotted in the third panel. Translational inertial forces, proportional to $-m_j a_x$, are negative and spread throughout the fuselage, while rotational inertial loads due to α_z are positive and appear concentrated aft of the center of gravity. These opposing contributions are superposed at the same axial stations, resulting in a net normal force distribution that spans the length of the vehicle.

In the fourth panel, the aerodynamic normal force distribution exhibits localized compressive regions, especially near the aft fuselage and base, where strong shock structures and base drag are present. The shear force diagram in the fifth panel reveals that shear loads attain their maximum values approximately in the same region as the axial load peak, i.e., around $x = 0.9$ m to 1.1 m, before reversing direction near the motor casing.

Finally, the bending moment diagram in the bottom panel shows that the moment peaks near the mid-fuselage at approximately $x \approx 1.4$ m, which is downstream of the axial load extremum. This spatial offset is expected, as bending moment results from the cumulative integration of the shear distribution. Overall, at Mach 2.0, the maximum axial-direction loads—both internal force and bending moment—occur in the forward to mid-fuselage region, emphasizing it as the structurally critical zone under transonic–supersonic aerodynamic and inertial conditions.

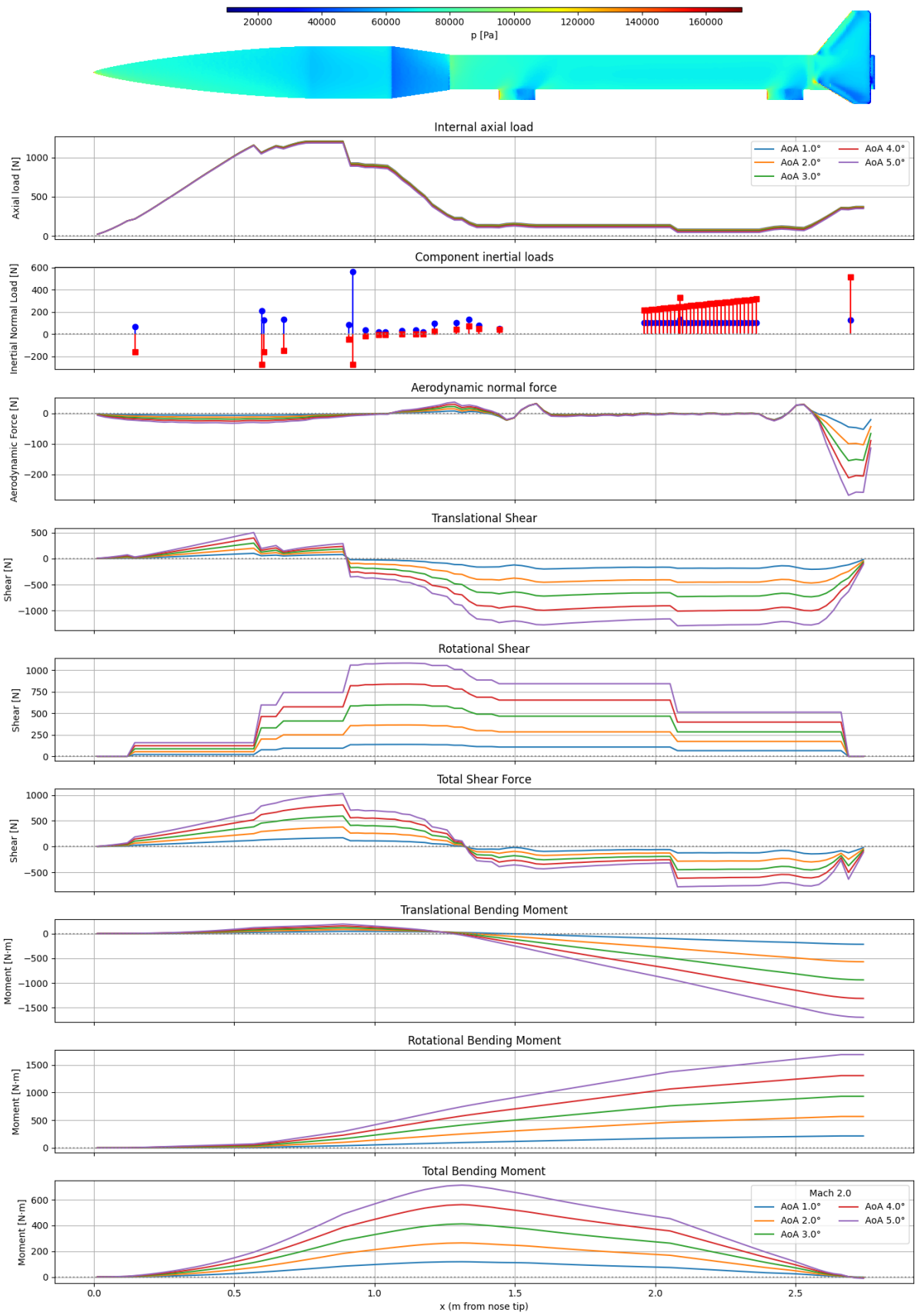


Figure 4.11: Axial-direction load breakdown at Mach 2.0 across multiple AoA values. From top to bottom: CFD surface pressure field, internal axial load, component inertial contributions, aerodynamic normal force, shear force diagram, and bending moment diagram.

Chapter 5

References

5.1 General CFD and Simulation Resources

5.1.1 Video Tutorials

- The k-omega Turbulence Model: https://youtu.be/26QaCK6wDp8
- y+ for Laminar Flow: https://youtu.be/yfYr72Gc4S4
- The k - omega SST Turbulence Model: https://youtu.be/myv-ityFnS4
- What is the difference between Upwind, Linear Upwind and Central Differencing?: https://youtu.be/JVE0fNkc540

5.2 cfMesh - Meshing Tool

5.2.1 cfMesh Video Resources

- cfMesh Tutorial Video 1: https://youtu.be/-V-Rd3-m0s0
- cfMesh Tutorial Video 2: https://youtu.be/5ne5ZUoObGk

5.2.2 cfMesh Documentation and Software

- cfMesh Official Website: cfMesh - Automated Mesh Generation
- cfMesh User Guide v1.1: CF-MESH+ User Guide

5.3 High Performance Computing - SLURM

5.3.1 Compute Canada Resources

- A Complete Guide for using Compute Canada for Deep Learning: Comprehensive Tutorial on Compute Canada Usage

5.3.2 Speed HPC Cluster Documentation

- Speed: The GCS ENCS Cluster - Complete User Manual: Speed HPC Cluster User Guide
- Speed HPC GitHub Repository: Speed Manual PDF

5.4 Linux and System Administration

5.4.1 Linux Basics Video Tutorials

- Linux Fundamentals Video 1: https://youtu.be/gd7BXuUQ91w
- Linux Fundamentals Video 2: https://youtu.be/s3ii48qYBxA

5.5 OpenFOAM Resources

5.5.1 Educational Channels and Training

- OpenFOAM József Nagy YouTube Channel: Comprehensive OpenFOAM Video Tutorials
- Wolf Dynamics - Multiphysics simulations, optimization, and data analytics: Professional OpenFOAM Training and Consulting Services

5.5.2 Official Documentation and Tutorials

- OpenFOAM v12 User Guide - Tutorials: Official OpenFOAM Tutorial Documentation
- CFD with OpenSource Software - Chalmers University: Advanced OpenFOAM Course Materials and Proceedings

Appendix A

Linux Command Line Basics

This chapter provides essential Linux command line knowledge required for OpenFOAM simulation workflows and HPC cluster operations. Understanding these fundamental commands is crucial for effective file management, navigation, and system interaction in computational fluid dynamics environments.

A.1 File and Directory Operations

A.1.1 Directory Navigation and Listing

Basic Directory Operations

```
$ pwd                                                    # Print working directory
/home/username/OpenFOAM/simulations

$ ls                                                     # List directory contents
0  0.orig  constant  logs  system

$ ls -l                                                  # Long format listing
drwxr-xr-x 2 username group 4096 Dec 15 10:30 0
drwxr-xr-x 2 username group 4096 Dec 15 10:25 0.orig
drwxr-xr-x 3 username group 4096 Dec 15 10:28 constant

$ ls -ltrh                                               # Long, time-sorted, human-readable
drwxr-xr-x 2 username group 4.0K Dec 15 10:25 0.orig
drwxr-xr-x 3 username group 4.0K Dec 15 10:28 constant
drwxr-xr-x 2 username group 4.0K Dec 15 10:30 0
-rw-r--r-- 1 username group 1.2M Dec 15 10:35 submit.log

$ ls -a                                                  # Include hidden files
.  ..  .bashrc  .gitignore  0  0.orig  constant

$ cd /path/to/directory                                  # Change directory (absolute path)
$ cd ../parent                                           # Navigate to parent directory
$ cd ~                                                   # Change to home directory
$ cd -                                                   # Return to previous directory
```

Listing A.1: Directory navigation and content listing

The `ls` command variants provide different levels of detail: `-l` shows permissions and ownership, `-t` sorts by modification time, `-r` reverses order, and `-h` displays file sizes in human-readable format. The `cd` command enables navigation using absolute paths, relative paths, or shortcuts like `~` for home directory.

A.1.2 File and Directory Management

Creating and Removing Directories

```
$ mkdir newproject                                  # Create single directory
$ mkdir -p project/case/mesh                        # Create nested directories
$ mkdir -p logs/{mesh,solver,post}                  # Create multiple directories

$ rmdir emptydirectory                              # Remove empty directory
$ rm -rf directory                                  # Remove directory and contents (
    dangerous!)
$ rm -ri directory                                  # Interactive removal with
    confirmation
```

Listing A.2: Directory creation and removal

File Operations

```
$ touch newfile.txt                                 # Create empty file
$ cp source.txt destination.txt                     # Copy file
$ cp -r sourcedir/ destdir/                         # Copy directory recursively
$ cp *.log backup/                                  # Copy multiple files using
    wildcards

$ mv oldname.txt newname.txt                        # Rename file
$ mv file.txt /path/to/destination/                 # Move file to directory
$ mv *.dat results/                                 # Move multiple files

$ rm file.txt                                       # Remove file
$ rm *.tmp                                          # Remove files with wildcard
$ rm -i important.txt                               # Interactive removal
```

Listing A.3: File manipulation and management

The `mkdir -p` option creates parent directories as needed, while `rm -rf` forcefully removes directories and all contents without confirmation. Always use `rm -i` for important files to prevent accidental deletion.

A.2 File Content and Text Processing

A.2.1 Viewing and Editing File Contents

File Content Display

```
$ cat controlDict                                   # Display entire file content
$ less controlDict                                  # Page through file (press q to
    quit)
$ head -n 20 logfile                                # Show first 20 lines
$ tail -n 50 logfile                                # Show last 50 lines
$ tail -f solver.log                                # Follow file changes in real-time

$ grep "SIMPLE" system/fvSolution                   # Search for text in file
$ grep -r "boundaryField" .                         # Recursive search in directory
$ grep -n "inlet" 0/U                               # Show line numbers with matches
$ grep -i "error" *.log                             # Case-insensitive search
```

Listing A.4: File content viewing and searching

Text Processing and Manipulation

```
$ wc -l logfile                                     # Count lines in file
$ wc -w document.txt                                # Count words in file
$ sort data.txt                                     # Sort lines alphabetically
$ sort -n numbers.txt                               # Sort numerically
$ uniq sorted.txt                                   # Remove duplicate lines
```

```
$ cut -d' ' -f3 data.txt                          # Extract third column (space-
    delimited)
$ awk '{print $2}' residuals.dat                  # Print second column using awk
$ sed 's/old/new/g' file.txt                      # Replace text (stream editor)
```

Listing A.5: Advanced text processing commands

The `tail -f` command is particularly useful for monitoring log files during simulation runs, while `grep` enables efficient searching through configuration files and results.

A.3 System Information and Process Management

A.3.1 System Status and Resource Monitoring

System Information Commands

```
$ top                                    # Display running processes (press q to quit)
$ htop                                   # Enhanced process viewer (if available)
$ ps aux                                 # List all running processes
$ ps aux | grep openfoam                 # Find OpenFOAM processes

$ df -h                                  # Disk space usage (human-readable)
$ du -sh *                               # Directory sizes in current location
$ du -sh . | sort -h                     # Sort directories by size

$ free -h                                # Memory usage information
$ uptime                                 # System load and uptime
$ whoami                                 # Current username
$ id                                     # User and group information
```

Listing A.6: System status and resource monitoring

Process Control

```
$ ./simulation.sh &                       # Run command in background
$ nohup ./longrun.sh &                    # Run immune to hangups
$ jobs                                    # List active jobs
$ fg %1                                   # Bring job 1 to foreground
$ bg %1                                   # Send job 1 to background

$ kill 12345                              # Terminate process by PID
$ kill -9 12345                           # Force kill process
$ killall -9 simpleFoam                  # Kill all processes by name
```

Listing A.7: Process management and job control

A.4 Environment and Configuration

A.4.1 Environment Variables and Shell Configuration

Environment Management

```
$ echo $HOME                              # Display home directory path
$ echo $PATH                              # Show executable search paths
$ echo $USER                              # Current username
$ printenv                                # Display all environment variables
```

```
$ export FOAM_RUN=$HOME/OpenFOAM/simulations       # Set environment variable
$ echo $FOAM_RUN                                    # Verify variable setting
$ unset FOAM_RUN                                    # Remove environment variable

$ source ~/.bashrc                                  # Reload bash configuration
$ source $FOAM_INST_DIR/OpenFOAM-v2406/etc/bashrc # Load OpenFOAM environment
$ which simpleFoam                                  # Locate executable path
$ type ls                                           # Show command type and location
```

Listing A.8: Environment variables and shell configuration

File Permissions and Ownership

```
$ ls -l script.sh                                   # Check file permissions
-rw-r--r-- 1 username group 1234 Dec 15 10:30 script.sh

$ chmod +x script.sh                                # Make file executable
$ chmod 755 script.sh                               # Set specific permissions (rwxr-xr
    -x)
$ chmod u+w file.txt                                # Add write permission for user

$ chown username:group file.txt                     # Change file ownership
$ chgrp newgroup file.txt                          # Change group ownership
```

Listing A.9: File permissions and ownership management

The permission string format is `rwxrwxrwx` representing user, group, and other permissions where `r`=read (4), `w`=write (2), `x`=execute (1).

A.5 Network and Data Transfer

A.5.1 Remote Access and File Transfer

SSH and Remote Operations

```
$ ssh username@hostname                             # Connect to remote server
$ ssh -X username@hostname                          # Enable X11 forwarding
$ ssh -L 11111:localhost:11111 user@cluster.ca     # SSH tunnel for port forwarding

$ scp file.txt user@remote:/path/destination/       # Copy file to remote server
$ scp -r directory/ user@remote:/path/              # Copy directory recursively
$ scp user@remote:/path/file.txt ./                 # Copy from remote to local

$ rsync -avz local/ user@remote:/path/              # Synchronize directories
$ rsync -avz --progress large_file user@remote:/   # Show transfer progress
```

Listing A.10: SSH connections and remote operations

Archive and Compression

```
$ tar -czf results.tar.gz results/                 # Create compressed archive
$ tar -xzf archive.tar.gz                          # Extract compressed archive
$ tar -tzf archive.tar.gz                          # List archive contents

$ gzip large_file                                   # Compress single file
$ gunzip compressed_file.gz                         # Decompress file
$ zip -r project.zip project/                      # Create ZIP archive
$ unzip archive.zip                                 # Extract ZIP archive
```

Listing A.11: File compression and archive operations

A.6 Advanced Command Line Techniques

A.6.1 Command Chaining and Redirection

Input/Output Redirection

```
$ command > output.txt                                  # Redirect stdout to file
$ command >> logfile.txt                               # Append stdout to file
$ command 2> errors.txt                                # Redirect stderr to file
$ command > output.txt 2>&1                            # Redirect both stdout and stderr

$ command < input.txt                                   # Use file as input
$ command1 | command2                                   # Pipe output to next command
$ grep "error" *.log | wc -l                           # Count error occurrences
$ ls -la | grep "^d"                                    # List only directories
```

Listing A.12: Command chaining and I/O redirection

Command History and Shortcuts

```
$ history                                               # Show command history
$ !!                                                    # Repeat last command
$ !123                                                  # Repeat command number 123
$ !grep                                                 # Repeat last grep command

# Keyboard shortcuts:
# Ctrl+C  - Interrupt/cancel current command
# Ctrl+Z  - Suspend current process
# Ctrl+D  - EOF signal / logout
# Ctrl+A  - Move cursor to beginning of line
# Ctrl+E  - Move cursor to end of line
# Ctrl+R  - Search command history
# Tab     - Auto-complete commands and filenames
```

Listing A.13: Command history and keyboard shortcuts

These fundamental Linux commands form the foundation for effective OpenFOAM simulation management, enabling efficient file operations, process control, and system monitoring essential for computational fluid dynamics workflows.

Appendix B

SLURM Workload Manager and Job Submission

SLURM (Simple Linux Utility for Resource Management) is a highly scalable cluster management and job scheduling system used by most high-performance computing (HPC) facilities. This chapter covers essential SLURM commands and job submission techniques for OpenFOAM simulations and computational workloads.

B.1 SLURM Architecture and Concepts

SLURM manages computational resources through a hierarchical structure of clusters, nodes, and partitions. Jobs are submitted to queues where they wait for resource allocation based on priority, resource requirements, and availability. Understanding this architecture is crucial for effective HPC utilization.

Key SLURM Components:

- **Nodes**: Individual computers in the cluster
- **Partitions**: Logical groupings of nodes with similar characteristics
- **Jobs**: User-submitted computational tasks
- **Job Steps**: Individual processes within a job
- **Allocations**: Reserved resources for job execution

B.2 Basic SLURM Commands

B.2.1 Cluster Information and Status

System Information Commands

```
$ sinfo                                          # Display node and partition info
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
compute*     up   infinite     64   idle node[001-064]
gpu          up   infinite      8   idle gpu[001-008]
bigmem       up   infinite      4   idle bigmem[001-004]

$ sinfo -Nel                                     # Detailed node information
$ sinfo -p compute                               # Information for specific
    partition
$ sinfo -t idle                                  # Show only idle nodes

$ squeue                                         # Display job queue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
12345   compute     CFD1 username  R    1:23:45      4 node[010-013]
```

```
12346   compute     CFD2 username PD       0:00      8 (Resources)

$ squeue -u $USER                                   # Show only your jobs
$ squeue -p compute                                 # Show jobs in specific partition
$ squeue -t RUNNING                                 # Show only running jobs
```

Listing B.1: SLURM cluster information and status

The `sinfo` command provides cluster status information, showing partition availability, time limits, and node states. The `squeue` command displays the job queue with job states: `PD` (pending), `R` (running), `CG` (completing), `CD` (completed), `F` (failed).

B.2.2 Job Information and History

Job Status and Accounting

```
$ scontrol show job 12345                           # Detailed job information
JobId=12345 JobName=OpenFOAM_sim
   UserId=username(1001) GroupId=research(100)
   Priority=4294901757 Nice=0 Account=def-account
   JobState=RUNNING Reason=None Dependency=(null)
   RunTime=01:23:45 TimeLimit=20:00:00
   StartTime=2024-01-15T10:30:00
   NodeList=node[010-013]

$ sacct                                             # Show job accounting information
$ sacct -j 12345                                    # Accounting for specific job
$ sacct --starttime=2024-01-01                      # Jobs since specific date
$ sacct --format=JobID,JobName,State,ExitCode       # Custom output format

$ scancel 12345                                     # Cancel specific job
$ scancel -u $USER                                  # Cancel all your jobs
$ scancel -n job_name                               # Cancel jobs by name
```

Listing B.2: Job monitoring and accounting information

B.3 Job Submission with sbatch

B.3.1 Basic Job Submission

Simple Job Submission

```
$ sbatch simple_job.sh                              # Submit job script
Submitted batch job 12347

$ sbatch --job-name=MySimulation job.sh             # Submit with custom name
$ sbatch --time=10:00:00 --mem=32G job.sh           # Override resource requests
$ sbatch --partition=gpu --gres=gpu:1 gpu_job.sh    # Submit to GPU partition
```

Listing B.3: Basic sbatch job submission

Interactive Job Allocation

```
$ salloc --ntasks=4 --time=2:00:00                  # Allocate interactive resources
salloc: Granted job allocation 12348
salloc: Waiting for resource configuration
salloc: Nodes node010 are ready for job
```

```
$ srun --pty bash                                    # Get interactive shell on
    allocated node
$ srun -n 4 mpirun my_mpi_program                   # Run MPI program interactively
$ exit                                               # Release allocation
```

Listing B.4: Interactive resource allocation

B.3.2 SLURM Directive Options

Resource Specification Directives

```
#!/bin/bash
#SBATCH --job-name=OpenFOAM_simulation               # Job name
#SBATCH --account=def-research                       # Billing account
#SBATCH --time=24:00:00                              # Maximum runtime (D-HH:MM:SS)
#SBATCH --nodes=2                                    # Number of nodes
#SBATCH --ntasks=64                                  # Total number of tasks
#SBATCH --ntasks-per-node=32                         # Tasks per node
#SBATCH --cpus-per-task=1                            # CPU cores per task
#SBATCH --mem=128G                                   # Total memory per node
#SBATCH --mem-per-cpu=2G                             # Memory per CPU core

#SBATCH --partition=compute                          # Partition/queue name
#SBATCH --qos=high                                   # Quality of service
#SBATCH --constraint="intel&ib"                      # Node constraints
#SBATCH --exclusive                                  # Exclusive node access

#SBATCH --output=job_%j.out                          # Standard output file (%j = job ID
    )
#SBATCH --error=job_%j.err                           # Standard error file
#SBATCH --mail-type=ALL                              # Email notifications
#SBATCH --mail-user=user@institution.edu            # Email address

#SBATCH --dependency=afterok:12345                   # Job dependencies
#SBATCH --array=1-10                                 # Job array specification
```

Listing B.5: Common SLURM job directives

The `%j` variable in file names is replaced with the job ID, ensuring unique output files for each job. Time formats can be specified as minutes (M), hours:minutes (H:MM), or days-hours:minutes (D-HH:MM).

B.4 Advanced Job Management

B.4.1 Job Arrays and Dependencies

Job Array Submission

```
#!/bin/bash
#SBATCH --job-name=parameter_study
#SBATCH --array=1-20                                 # Run 20 array tasks
#SBATCH --ntasks=1
#SBATCH --time=4:00:00
#SBATCH --output=array_%A_%a.out                     # %A = array job ID, %a = task ID

# Use array task ID as parameter
MACH_NUMBER=$(echo "1.0 + $SLURM_ARRAY_TASK_ID * 0.1" | bc)
echo "Running simulation with Mach number: $MACH_NUMBER"

# Modify simulation parameters based on array index
```

```
sed -i "s/Ma.*/Ma $MACH_NUMBER;/" constant/thermophysicalProperties
```

Listing B.6: Job arrays for parameter studies

Job Dependencies

```
$ MESH_JOB=$(sbatch --parsable mesh_generation.sh)  # Submit mesh job, get job ID
$ sbatch --dependency=afterok:$MESH_JOB solver.sh   # Submit solver after mesh
    completes
$ sbatch --dependency=afternotok:12345 cleanup.sh   # Run if job 12345 fails
$ sbatch --dependency=afterany:12345:12346 post.sh  # Run after either job completes
```

Listing B.7: Job dependency management

B.4.2 Resource Optimization

Memory and CPU Optimization

```
#!/bin/bash
#SBATCH --job-name=optimized_job
#SBATCH --nodes=1
#SBATCH --ntasks=32                                 # Match decomposition
#SBATCH --mem=0                                     # Use all available memory
#SBATCH --time=12:00:00

# Calculate optimal processor decomposition
NPROCS=$SLURM_NTASKS
NX=$((NPROCS / 4))
NY=2
NZ=2

# Update decomposition dictionary
foamDictionary system/decomposeParDict -entry "numberOfSubdomains" -set "$NPROCS"
foamDictionary system/decomposeParDict -entry "n" -set "($NX $NY $NZ)"
```

Listing B.8: Resource optimization strategies

B.5 Monitoring and Troubleshooting

B.5.1 Job Monitoring

Real-time Job Monitoring

```
$ sstat -j 12345                                    # Real-time job statistics
$ sstat -j 12345 --format=JobID,MaxRSS,AveCPU      # Custom format statistics

$ ssh node010                                       # Connect to compute node
$ top -u $USER                                      # Monitor processes on node
$ nvidia-smi                                        # GPU utilization (if applicable)

$ tail -f slurm-12345.out                          # Follow job output in real-time
$ watch "squeue -u $USER"                          # Monitor job queue continuously
```

Listing B.9: Job monitoring and resource usage

Post-Job Analysis

```
$ sacct -j 12345 --format=JobID,State,ExitCode,MaxRSS,Elapsed
```

```
       JobID      State ExitCode     MaxRSS    Elapsed
------------ ---------- -------- ---------- ----------
12345               COMPLETED      0:0      15.2G   02:45:30
12345.batch     COMPLETED      0:0      15.2G   02:45:30

$ seff 12345                                          # Job efficiency summary
Job ID: 12345
Cluster: mycluster
User/Group: username/research
State: COMPLETED (exit code 0)
Cores: 32
CPU Utilized: 78:23:45
CPU Efficiency: 89.45% of 87:36:00 core-walltime
Memory Utilized: 15.2 GB
Memory Efficiency: 23.75% of 64.0 GB
```

Listing B.10: Job completion analysis

B.5.2 Common Issues and Solutions

Troubleshooting Job Failures

```
# Check job details for failure reason
$ scontrol show job 12345

# Common failure reasons and solutions:
# DUE_TO_TIME_LIMIT - Job exceeded time limit
$ sbatch --time=48:00:00 job.sh                       # Increase time limit

# OUT_OF_MEMORY - Job exceeded memory allocation
$ sbatch --mem=256G job.sh                            # Increase memory

# NODE_FAIL - Hardware failure
$ sbatch --exclude=node010 job.sh                     # Exclude failing node

# PENDING with reason (QOSMaxJobsPerUserLimit)
$ squeue -u $USER                                     # Check current job count
$ scancel 12340                                       # Cancel unnecessary jobs

# Check system-wide issues
$ sinfo -R                                            # Show node reservations
$ sdiag                                               # SLURM scheduler diagnostics
```

Listing B.11: Common job failure diagnostics

B.6 Best Practices and Optimization

B.6.1 Efficient Resource Utilization

Resource Request Guidelines

```
# Start with conservative estimates and scale up
#SBATCH --time=4:00:00                               # Start with shorter times
#SBATCH --mem=32G                                    # Conservative memory estimate

# Use checkpoint/restart for long jobs
#SBATCH --signal=SIGUSR1@90                          # Signal 90 seconds before time
    limit
```

```
# Optimize for queue priority
$ sprio -u $USER                                    # Check job priority factors
$ sshare -u $USER                                   # Check fair share usage

# Test scaling with smaller jobs first
#SBATCH --ntasks=8                                 # Test with fewer processors
#SBATCH --ntasks=16                                # Scale up gradually
#SBATCH --ntasks=32
```

Listing B.12: Resource optimization best practices

Job Script Template for OpenFOAM

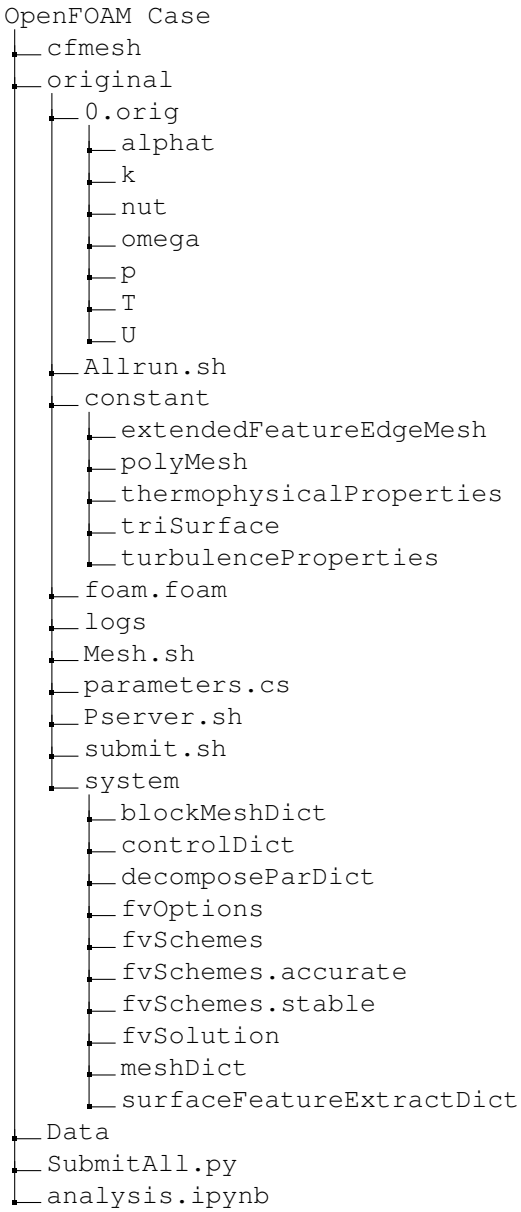
```
#!/bin/bash
#SBATCH --job-name=OpenFOAM_CFD
#SBATCH --account=def-research
#SBATCH --time=12:00:00
#SBATCH --nodes=2
#SBATCH --ntasks=64
#SBATCH --mem=128G
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --mail-user=user@institution.edu
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err

# Load required modules
module purge
module load openfoam/v2406

# Set up environment
cd $SLURM_SUBMIT_DIR
export OMP_NUM_THREADS=1

# Log job information
echo "Job started at: $(date)"
echo "Running on nodes: $SLURM_JOB_NODELIST"
echo "Number of processors: $SLURM_NTASKS"

# Run simulation
./Allrun.sh

# Job completion
echo "Job completed at: $(date)"
echo "Job statistics:"
sstat -j $SLURM_JOB_ID --format=JobID,MaxRSS,AveCPU || true
```

Listing B.13: Complete OpenFOAM job script template

This comprehensive SLURM guide provides the essential knowledge for effective job submission and resource management on HPC systems, enabling efficient execution of OpenFOAM simulations and other computational workloads.

Appendix C

OpenFOAM case structure



C.1 Top-level overview

```
OpenFOAM Case
├── cfmesh
├── original
├── Data
├── SubmitAll.py
└── analysis.ipynb
```

- `cfmesh` : Local install of **cfMesh** utilities.
- `original`: The base OpenFOAM case. All parameterized runs are copied from here.
- `Data` : Sweep workspace. Each parameter set becomes a subfolder copied from `original` and submitted to the cluster. Post-processing outputs should also land under each run here.
- `SubmitAll.py` : a Python driver that clones the `original` case into `Data/`*`runName`*, applies the overrides, prepares `submit.sh`, and submits the job to the cluster.
- `analysis.ipynb` : Jupyter notebook with utilities to read generated CFD results and produce plots and tables.

C.2 Building cfMesh tools

`cfMesh` is an open-source meshing library implemented within the OpenFOAM framework; we build it from source after initializing the OpenFOAM environment. See the official page for details and documentation: cfMesh (open source).

```
$ cd cfmesh
$ ls -la
drwxr-xr-x  8 USER USER 4.0K May  9 00:23 testingInterfaces
drwxr-xr-x  3 USER USER 4.0K May  9 00:23 python
drwxr-xr-x  6 USER USER 4.0K May  9 00:23 executables
-rw-r--r--  1 USER USER 2.2K May  9 00:23 README
-rw-r--r--  1 USER USER 1.8K May  9 00:23 buildDefs.include
-rwxr-xr-x  1 USER USER  143 May  9 00:23 Allwmake
-rwxr-xr-x  1 USER USER  146 May  9 00:23 Allwclean
drwxr-xr-x 24 USER USER 4.0K May  9 00:23 utilities
-rw-r--r--  1 USER USER   35 May  9 00:23 versionInfo.H
drwxr-xr-x  8 USER USER 4.0K May  9 00:31 meshLibrary
```

Listing C.1: Examining CFmesh directory structure

Before building CFmesh, source the OpenFOAM environment and then execute the build script:

```
$ source <path_to_openfoam>/etc/bashrc
$ ./Allwmake
```

Listing C.2: Building CFmesh

The `Allwmake` script will compile all the CFmesh libraries and utilities. Ensure that your OpenFOAM environment is properly configured before running the build process.

C.3 Original Case

The original case directory contains all necessary files and subdirectories for the OpenFOAM simulation workflow. This section provides an overview of the case structure and describes the function of each component within the computational fluid dynamics simulation framework.

`original`

```
├── 0.orig
├── Allrun.sh
├── constant
├── foam.foam
├── logs
├── Mesh.sh
├── parameters.cs
├── Pserver.sh
├── submit.sh
└── system
```

The case directory is organized into several key components, each serving a specific purpose in the simulation workflow:

- **0.orig**: Contains the original initial and boundary condition files for all field variables (velocity, pressure, temperature, etc.). This directory serves as a template that can be copied to create the actual `0` directory before running the simulation.
- **Allrun.sh**: Master script that orchestrates the complete simulation workflow. This shell script executes the sequence of all utilities and solvers in the correct order, from preprocessing through mesh generation to the final solver execution.
- **constant**: Standard OpenFOAM directory containing time-invariant data such as physical properties, turbulence model parameters, and mesh-related files. This directory holds constant coefficients and material properties required throughout the simulation.
- **foam.foam**: ParaView reader file that allows direct opening of the OpenFOAM case in ParaView for post-processing and visualization. This empty file serves as a marker for ParaView to recognize the directory as an OpenFOAM case.
- **logs**: Directory designated for storing output logs and runtime information from various OpenFOAM utilities and solvers. This facilitates debugging and monitoring of the simulation progress.
- **Mesh.sh**: Specialized script containing all meshing processes including Cartesian mesh generation, surface feature extraction, and other mesh-related utilities. This script handles the complete mesh generation workflow from geometry processing to final mesh refinement.
- **parameters.cs**: Configuration file containing simulation parameters and settings. This file centralizes the control parameters that govern the simulation behavior, allowing for easy modification without editing multiple configuration files.
- **Pserver.sh**: ParaView server script that enables remote visualization of results stored on a computing cluster. This script facilitates the setup of a ParaView server connection, allowing users to visualize large datasets remotely without transferring files locally.
- **submit.sh**: Cluster job submission script containing the necessary commands and resource specifications for submitting the simulation to a high-performance computing cluster. This script handles job scheduling, resource allocation, and execution environment setup.
- **system**: Standard OpenFOAM directory containing simulation control parameters, discretization schemes, solution algorithms, and function objects. This directory houses the core configuration files that control how OpenFOAM solves the governing equations.

C.3.1 Initial and Boundary Conditions (0.orig)

The `0.orig` directory contains the template files for initial and boundary conditions of all field variables. These files are later copied to the `0` directory before simulation execution. The directory structure is organized as follows:

```
0.orig
├── alphat
├── k
└── nut
```

```
├── omega
├── p
├── T
└── U
```

All OpenFOAM field files follow a standardized format with a common header structure that identifies the file type, location, and physical dimensions. Each file contains the following key sections:

- **FoamFile header**: Standard OpenFOAM file identifier
- **Dimensions**: Physical dimensions in SI base units $[MLT\Theta NIJ]$
- **Internal field**: Initial values throughout the computational domain
- **Boundary field**: Boundary conditions for each patch

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  11
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;    // Scalar field variable
    location    "0";               // Time directory location
    object      alphat;            // Field name
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [M L T \Theta N I J];    // for example [1 -1 -1 0 0 0 0] means [kg m^-1 s^-1]

internalField <value>;

boundaryField <Directory>;
```

Listing C.3: alphat file structure

Turbulent Thermal Diffusivity (alphat)

The `alphat` file defines the turbulent thermal diffusivity field, which represents the enhanced heat transfer due to turbulent mixing.

```
dimensions      [1 -1 -1 0 0 0 0];    // [kg m^-1 s^-1]

internalField   uniform 0;             // Zero initial turbulent diffusivity

boundaryField
{
    "(sides|outlet|inlet)"             // Far-field boundaries
    {
        type            calculated;     // Automatically calculated
        value           uniform 0;
    }
    rocket                             // Wall boundary
    {
        type            compressible::alphatWallFunction;
        Prt             0.85;          // Turbulent Prandtl number
        value           uniform 0;
    }
}
```

Listing C.4: alphat file structure

Turbulent Kinetic Energy (k)

The turbulent kinetic energy field represents the kinetic energy per unit mass associated with turbulent velocity fluctuations: $k = \frac{1}{2}\overline{u_i' u_i'}$.

```
#include "../parameters.cs"        // Include parameter definitions

dimensions      [0 2 -2 0 0 0 0]; // [m^2 s^-2]

k 0.01;                           // Turbulent kinetic energy value

internalField   uniform $k;       // Uniform initial field

boundaryField
{
    "(sides|outlet|inlet)"
    {
        type            inletOutlet;    // Inlet/outlet boundary
        inletValue      uniform $k;     // Value for inflow
        value           uniform $k;
    }
    rocket
    {
        type            kqRWallFunction;  // Wall function for k
        value           uniform $k;
    }
}
```

Listing C.5: k file structure

Turbulent Viscosity (nut)

The turbulent kinematic viscosity ν_t represents the enhanced momentum diffusion due to turbulent mixing, calculated as $\nu_t = C_\mu \frac{k^2}{\omega}$.

```
#include "../parameters.cs"

dimensions      [0 2 -1 0 0 0 0];     // [m^2 s^-1]

nut 0.0;                              // Initial turbulent viscosity

internalField   uniform $nut;

boundaryField
{
    "(sides|outlet|inlet)"
    {
        type            calculated;   // Automatically calculated
        value           uniform $nut;
    }
    rocket
    {
        type            nutkWallFunction;
        Cmu             0.09;         // Turbulence model constant
        kappa           0.41;         // von Krmn constant
        E               9.8;          // Wall roughness parameter
        value           uniform $nut;
    }
}
```

Listing C.6: nut file structure

Specific Turbulent Dissipation Rate (omega)

The specific dissipation rate ω represents the rate of dissipation of turbulent kinetic energy per unit turbulent kinetic energy: $\omega = \frac{\epsilon}{k}$.

```
dimensions      [0 0 -1 0 0 0 0];    // [s^-1]

#include "../parameters.cs"

omega 10;                            // Specific dissipation rate

internalField   uniform $omega;

boundaryField
{
    "(sides|outlet|inlet)"
    {
        type                    inletOutlet;
        inletValue              uniform $omega;
        value                   uniform $omega;
    }
    rocket
    {
        type            omegaWallFunction;
        Cmu             0.09;        // Turbulence model constant
        kappa           0.41;        // von Krmn constant
        E               9.8;         // Wall roughness parameter
        value           uniform $omega;
    }
}
```

Listing C.7: omega file structure

Pressure (p)

The pressure field represents the thermodynamic pressure in the fluid domain.

```
#include "../parameters.cs"

dimensions      [1 -1 -2 0 0 0 0];   // [kg m^-1 s^-2] = [Pa]

internalField   uniform $P;           // Reference pressure

boundaryField
{
    "(sides|outlet|inlet)"
    {
        type            freestreamPressure;
        freestreamValue $internalField;  // Freestream pressure
        inletValue      $internalField;
        value           $internalField;
    }
    rocket
    {
        type            zeroGradient;    // No pressure gradient at wall
    }
}
```

Listing C.8: p file structure

Temperature (T)

The temperature field represents the absolute temperature in the fluid domain.

```
#include "../parameters.cs"
```

```
dimensions      [0 0 0 1 0 0 0];      // [K]

internalField   uniform $T;           // Reference temperature

boundaryField
{
    "(sides|outlet|inlet)"
    {
        type            inletOutlet;
        gamma           1.4;          // Specific heat ratio
        inletValue      uniform $T;
        value           uniform $T;
    }
    rocket
    {
        type        zeroGradient;     // Adiabatic wall condition
    }
}
```

Listing C.9: T file structure

Velocity (U)

The velocity field represents the fluid velocity vector with components calculated based on angle of attack and flow conditions.

```
#include "../parameters.cs"  // retriving $AoA, $R,  $T, $gamma from parameters set

dimensions      [0 1 -1 0 0 0 0];   // [m s^-1]

// Angle of attack conversion to radians
alpha   #calc "$AoA * 3.14159265358979323846 / 180.0";

// Speed of sound calculation: a = sqrt(gamma * R * T)
a       #calc "sqrt($gamma * $R * $T)";

// Flow velocity: U = Ma * a
U       #calc "$Ma * $a";

// Velocity components
Ux      #calc "$U * cos($alpha * 1.0)";  // x-component
Uy      #calc "$U * sin($alpha * 1.0)";  // y-component
Uz      0;                               // z-component

internalField   uniform ($Ux $Uy $Uz);

boundaryField
{
    "(sides|outlet|inlet)"
    {
        type             inletOutlet;
        inletValue       $internalField;  // Freestream velocity
        value           $internalField;
    }
    rocket
    {
        type            noSlip;          // No-slip wall condition
    }
}
```

Listing C.10: U file structure with calculations

The velocity field utilizes OpenFOAM's `#calc` functionality to compute velocity components based on:

$$\alpha = \text{AoA} \times \frac{\pi}{180} \quad \text{(angle of attack in radians)} \tag{C.1}$$

$$a = \sqrt{\gamma RT} \quad \text{(speed of sound)} \tag{C.2}$$

$$U = \text{Ma} \times a \quad \text{(flow velocity)} \tag{C.3}$$

$$U_x = U\cos(\alpha) \quad \text{(x-velocity component)} \tag{C.4}$$

$$U_y = U\sin(\alpha) \quad \text{(y-velocity component)} \tag{C.5}$$

C.3.2 Constant Directory

The `constant` directory contains time-invariant data including mesh information, physical properties, and turbulence model settings. This directory houses all the fundamental parameters that remain unchanged throughout the simulation.

```
constant
├── extendedFeatureEdgeMesh
│   ├── rocket.extendedFeatureEdgeMesh.gz
│   └── rocketPractice.extendedFeatureEdgeMesh.gz
├── polyMesh
│   ├── boundary
│   ├── faces.gz
│   ├── neighbour.gz
│   ├── owner.gz
│   └── points.gz
├── thermophysicalProperties
├── triSurface
│   ├── combined.fms
│   ├── combined.stl
│   ├── patches.stl
│   ├── rocket.eMesh
│   └── rocket.stl
└── turbulenceProperties
```

Mesh Components

polyMesh Directory The `polyMesh` directory contains the computational mesh generated by OpenFOAM mesh utilities such as `blockMesh`, `snappyHexMesh`, or `cfMesh`. This polyhedral mesh consists of:

- **points.gz**: Coordinates of all mesh vertices in compressed format
- **faces.gz**: Face connectivity information defining mesh faces
- **owner.gz**: Cell ownership data for internal faces
- **neighbour.gz**: Neighboring cell information for internal faces
- **boundary**: Boundary patch definitions and face ranges

triSurface Directory The `triSurface` directory contains surface geometry files used during mesh generation:

- **STL files**: Stereolithography files containing triangulated surface geometry
- **eMesh files**: Edge mesh files for feature edge extraction and refinement
- **FMS files**: cfMesh native surface format with additional metadata for mesh generation

extendedFeatureEdgeMesh Contains compressed feature edge mesh files used for mesh refinement around sharp edges and geometric features. These files are generated by `surfaceFeatureExtract` utility.

Turbulence Properties

The turbulence model configuration defines the approach for modeling turbulent flows in the simulation.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      turbulenceProperties;
}

simulationType      RAS;            // Reynolds-Averaged Simulation

RAS
{
    RASModel        kOmegaSST;       // Shear Stress Transport model
    turbulence      on;              // Enable turbulence modeling
    printCoeffs     on;              // Print model coefficients
}
```

Listing C.11: Turbulence Properties Configuration

The configuration specifies:

- **simulationType**: Reynolds-Averaged Simulation (RAS) approach
- **RASModel**: k-ω SST turbulence model combining the benefits of k-ω near walls and k-ϵ in the freestream
- **turbulence**: Enables turbulent viscosity calculations
- **printCoeffs**: Outputs model coefficients for verification

The k-ω SST model solves transport equations for turbulent kinetic energy (k) and specific dissipation rate (ω):

$$\frac{\partial k}{\partial t} + \nabla \cdot (\mathbf{U}k) = P_k - \beta^* k\omega + \nabla \cdot [(\nu + \sigma_k \nu_t)\nabla k] \tag{C.6}$$

$$\frac{\partial \omega}{\partial t} + \nabla \cdot (\mathbf{U}\omega) = \gamma \frac{P_k}{\nu_t} - \beta\omega^2 + \nabla \cdot [(\nu + \sigma_\omega \nu_t)\nabla \omega] + 2(1 - F_1)\sigma_{\omega 2} \frac{\nabla k \cdot \nabla \omega}{\omega} \tag{C.7}$$

Thermophysical Properties

The thermophysical properties define the fluid characteristics and equation of state for compressible flow simulations.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      thermophysicalProperties;
}

thermoType
{
    type            hePsiThermo;      // Enthalpy-based compressible
    mixture         pureMixture;      // Single component fluid
    transport       const;            // Constant transport properties
    thermo          hConst;           // Constant specific heat
    equationOfState perfectGas;       // Ideal gas law
    specie          specie;           // Single species
    energy          sensibleEnthalpy; // Energy formulation
}

mixture
{
    specie
    {
        molWeight   28.9;             // Molecular weight [kg/kmol]
```

```
    }
    thermodynamics
    {
        Cp          1005;               // Specific heat [J/kg/K]
        Hf          0;                  // Heat of formation [J/kg]
    }
    transport
    {
        mu          1.82e-05;           // Dynamic viscosity [kg/m/s]
        As          1.4792e-06;         // Sutherland coefficient
        Ts          116;                // Sutherland temperature [K]
        Pr          0.7;                // Prandtl number
    }
}
```

Listing C.12: Thermophysical Properties Configuration

The thermophysical model configuration includes:

- **Perfect gas equation of state**: $p = \rho RT$ where $R = \frac{R_u}{M}$
- **Sutherland viscosity model**: $\mu = \mu_0 \frac{T_0+S}{T+S} \left(\frac{T}{T_0}\right)^{3/2}$
- **Air properties**: Standard atmospheric air with molecular weight 28.9 kg/kmol
- **Transport properties**: Constant Prandtl number of 0.7 for air

Boundary Mesh Definition

The boundary file defines the mesh patches and their connectivity information.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       polyBoundaryMesh;    // Boundary mesh definition
    location    "constant/polyMesh";
    object      boundary;
}

3                                    // Number of boundary patches
(
    inlet                            // Inflow boundary
    {
        type            patch;       // Generic boundary patch
        nFaces          1600;        // Number of boundary faces
        startFace       357120;      // Starting face index
    }
    outlet                           // Outflow boundary
    {
        type            patch;
        nFaces          1600;
        startFace       358720;
    }
    sides                            // Far-field boundaries
    {
        type            patch;
        nFaces          12160;       // Larger patch for domain sides
        startFace       360320;
    }
)
```

Listing C.13: Boundary Patch Definitions

The boundary definition specifies three main patches:

- **inlet**: Inflow boundary with 1600 faces for uniform inlet conditions
- **outlet**: Outflow boundary with 1600 faces for pressure outlet

- **sides**: Far-field boundaries with 12160 faces representing domain sides

Note that the rocket geometry appears to be handled as a wall boundary condition in the field files, but is not explicitly listed in this boundary file, suggesting it may be defined elsewhere or handled through mesh manipulation utilities.

C.3.3 System Directory

The `system` directory contains simulation control files, numerical schemes, solver settings, and mesh generation parameters. These files define how OpenFOAM solves the governing equations and controls the simulation workflow.

```
system
├── blockMeshDict
├── controlDict
├── decomposeParDict
├── fvOptions
├── fvSchemes
├── fvSchemes.accurate
├── fvSchemes.stable
├── fvSolution
├── meshDict
└── surfaceFeatureExtractDict
```

Control Dictionary (controlDict)

The control dictionary manages simulation execution, time stepping, and function object operations. This file is divided into several sections for clarity.

Basic Simulation Control

```
FoamFile
{
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}

#include "../parameters.cs"

application     rhoSimpleFoam;          // Compressible steady-state solver

startFrom       latestTime;             // Continue from latest time
startTime       0;                      // Initial time value
stopAt          endTime;                // Stop condition
endTime         8000;                   // Maximum iterations
deltaT          1;                      // Time step size
maxDeltaT       1;                      // Maximum time step

writeControl    timeStep;               // Write control method
writeInterval   $WriteInterval;         // Write frequency
purgeWrite      10;                     // Keep only 10 time directories
writeFormat     ascii;                  // Output format
writePrecision  10;                     // Numerical precision
writeCompression on;                    // Compress output files

runTimeModifiable true;                 // Allow runtime modifications
adjustTimeStep  true;                   // Enable adaptive time stepping
maxCo           0.5;                    // Maximum Courant number
```

Listing C.14: Basic simulation control parameters

Flow Parameter Calculations

```
// Angle of attack conversion and trigonometric functions
alpha       #calc "$AoA * 3.14159265358979323846 / 180.0";
sinAlphaN   #calc "-sin($alpha * 1.0)";
cosAlphaN   #calc "-cos($alpha * 1.0)";
sinAlpha    #calc "sin($alpha * 1.0)";
cosAlpha    #calc "cos($alpha * 1.0)";

// Flow velocity calculations
a           #calc "sqrt($gamma * $R * $T)";     // Speed of sound
U           #calc "$Ma * $a";                   // Flow velocity
```

Listing C.15: Flow parameter calculations using #calc

Function Objects - Forces

```
functions
{
    forceCoeffs
    {
        type            forceCoeffs;
        libs            ("libforces.so");
        writeControl    timeStep;
        writeInterval   1;

        patches         (rocket);           // Wall patch for force calculation

        log             true;               // Enable logging
        rhoInf          1;                  // Reference density
        CofR            (0 0 0);            // Center of rotation
        liftDir         ($sinAlphaN $cosAlpha 0);   // Lift direction
        dragDir         ($cosAlpha $sinAlpha 0);    // Drag direction
        pitchAxis       (0 0 1);            // Pitching moment axis
        magUInf         $U;                 // Reference velocity magnitude
        lRef            3;                  // Reference length
        Aref            0.0188545853;       // Reference area
    }

    forces
    {
        type forces;                         // Function object for force calculation
        libs ("libforces.so"); // Library required for force calculations
        patches ("rocket");                  // Name of the boundary patch

        // Field names
        p               p;
        U               U;
        rho             rho;

        // Reference pressure [Pa]
        pRef            $P;
        log true;                            // Log results to a file

        // Optional: Specify center of rotation for moment calculation
        CofR (0 0 0);                        // Replace with the desired center of rotation
    }
```

Listing C.16: Force coefficient calculation function

Function Objects - Additional Monitoring

```
    // Field minimum and maximum values
    minMax
    {
        enabled         true;
        fields          (Ma U p rho T k omega);
        libs            ("libfieldFunctionObjects.so");
        log             true;
        type            fieldMinMax;
        write           true;
    }

    // Wall y+ calculation
    yPlus1
    {
        type            yPlus;
        libs            ("libfieldFunctionObjects.so");
        writePrecision  8;
        writeToFile     true;
        useUserTime     true;
        region          region0;
        enabled         true;
        log             true;
        timeStart       0;
        timeEnd         $endTime;
        executeControl  timeStep;
        executeInterval 1;
        writeControl    timeStep;
        writeInterval   $WriteInterval;
    }
```

Listing C.17: Field monitoring and analysis functions

Function Objects - Post-Processing

```
    // Centerplane Y slice
    sliceCentreY
    {
        type            surfaces;
        libs            ("libsampling.so");
        writeControl    $writeControl;
        writeInterval   $writeInterval;
        surfaceFormat   vtk;
        interpolationScheme cellPointFace;

        fields          (U T p Ma rho k omega);

        surfaces
        (
            planeSlice
            {
                type            cuttingPlane;
                planeType       pointAndNormal;
                pointAndNormalDict
                {
                    point       (0 0 0);
                    normal      (0 1 0);        // Y-normal plane
                }
                interpolate     true;
                mergeTol        1e-8;
            }
        );
    }
}
```

Listing C.18: Surface sampling and data extraction

Block Mesh Dictionary (blockMeshDict)
The block mesh dictionary defines the background mesh used as input for cfMesh generation.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}

#include "../parameters.cs"

scale 1.0;                              // Mesh scaling factor

// Cell count calculations
numCellsX #calc "round(($xMax - $xMin) / $maxCellSize)";
numCellsY #calc "round(($yMax - $yMin) / $maxCellSize)";
numCellsZ #calc "round(($zMax - $zMin) / $maxCellSize)";

vertices
(
    ($xMin $yMin $zMin)                 // Vertex 0: Bottom-left-front
    ($xMax $yMin $zMin)                 // Vertex 1: Bottom-right-front
    ($xMax $yMax $zMin)                 // Vertex 2: Top-right-front
    ($xMin $yMax $zMin)                 // Vertex 3: Top-left-front
    ($xMin $yMin $zMax)                 // Vertex 4: Bottom-left-back
    ($xMax $yMin $zMax)                 // Vertex 5: Bottom-right-back
    ($xMax $yMax $zMax)                 // Vertex 6: Top-right-back
    ($xMin $yMax $zMax)                 // Vertex 7: Top-left-back
);

blocks
(
    hex (0 1 2 3 4 5 6 7) ($numCellsX $numCellsY $numCellsZ)
        simpleGrading (1 1 1)           // Uniform cell distribution
);

boundary
(
    inlet   { type patch; faces ((0 4 7 3)); }      // Inflow boundary
    outlet  { type patch; faces ((2 6 5 1)); }      // Outflow boundary
    sides   { type patch; faces ((3 7 6 2) (1 5 4 0) (0 3 2 1) (4 5 6 7)); }
);
```

Listing C.19: Block mesh definition

Parallel Decomposition (decomposeParDict)

```
FoamFile
{
    version         2;
    format          ascii;
    class           dictionary;
    object          decomposeParDict;
}

numberOfSubdomains 64;                  // Total number of processor cores

method          hierarchical;           // Decomposition method
n               (16 2 2);               // Subdivision in x, y, z directions
```

Listing C.20: Domain decomposition for parallel processing

This configuration divides the computational domain into 64 subdomains arranged in a $16\times2\times2$ grid, optimizing for the expected mesh anisotropy with higher resolution in the streamwise direction.

Field Value Options (fvOptions)

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvOptions;
}

temperatureLimits
{
    type                limitTemperature;
    limitTemperatureCoeffs
    {
        selectionMode   all;            // Apply to entire domain
        min             100;            // Minimum temperature [K]
        max             1000;           // Maximum temperature [K]
    }
}

limitVelocity
{
    type            limitVelocity;
    selectionMode   all;                // Apply to entire domain
    max             1000;               // Maximum velocity magnitude [m/s]
}
```

Listing C.21: Field limiting and constraints

cfMesh Dictionary (meshDict)

The mesh dictionary controls the cfMesh generation process with surface refinement and boundary layer settings.

```
FoamFile
{
    version   2.0;
    format    ascii;
    class     dictionary;
    location  "system";
    object    meshDict;
}

#include "../parameters.cs"

surfaceFile "constant/triSurface/combined.fms";     // Input surface file

maxCellSize $maxCellSize;                           // Base cell size
symmetryPlaneLayerTopology      0;                  // Disable symmetry layers
meshMultipleDomainsAndBaffles   0;                  // Single domain mesh
nCellsBetweenLevels             4;                  // Refinement transition
```

Listing C.22: cfMesh basic configuration

Surface and Edge Refinement

```
edgeMeshRefinement
{
    edges
    {
        edgeFile "constant/triSurface/rocket.eMesh";
        additionalRefinementLevels    9;            // Edge refinement levels
    }
}

localRefinement
{
    rocket
```

```
    {
        additionalRefinementLevels    7;            // Surface refinement
        refinementThickness          0.1;           // Refinement zone thickness
    }
}
```

Listing C.23: Edge and surface refinement settings

Volumetric Refinement Boxes

```
objectRefinements
{
    box0  // Outermost refinement box
    {
        type box;
        additionalRefinementLevels    1;
        centre                       (5.5 0 0);
        lengthX                      16;
        lengthY                      5;
        lengthZ                      5;
    }

    box1  // Intermediate refinement
    {
        type box;
        additionalRefinementLevels    2;
        centre                       (4.5 0 0);
        lengthX                      13;
        lengthY                      4;
        lengthZ                      4;
    }

    box4  // Finest refinement near rocket
    {
        type box;
        additionalRefinementLevels    5;
        centre                       (2.4 0 0);
        lengthX                      5.4;
        lengthY                      1;
        lengthZ                      1;
    }
}
```

Listing C.24: Progressive volumetric refinement zones

Surface Feature Extraction (surfaceFeatureExtractDict)

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      surfaceFeatureExtractDict;
}

rocket.stl
{
    extractionMethod        extractFromSurface;    // Extract from STL surface
    includedAngle           178;                   // Feature angle threshold

    subsetFeatures
    {
        nonManifoldEdges    yes;                   // Include non-manifold edges
        openEdges           yes;                   // Include open edges
    }

    writeObj                no;                    // Disable OBJ output
    writeVTK                no;                    // Disable VTK output
}
```

Listing C.25: Feature edge extraction configuration

Finite Volume Schemes (fvSchemes)

The numerical schemes dictionary defines discretization methods for different terms in the governing equations.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSchemes;
}

fluxScheme          Tadmor;                          // Riemann solver

ddtSchemes
{
    default         Euler;                           // Time derivative scheme
}

gradSchemes
{
    default         cellLimited Gauss linear 1;    // Gradient calculation
    grad(p)         cellLimited Gauss linear 1;    // Pressure gradient
    limitedGradFace faceLimited Gauss linear 1;    // Face-limited gradient
}
```

Listing C.26: Temporal and gradient schemes

Convection Schemes

```
divSchemes
{
    default         none;

    // Momentum equation
    div(phi,U)      bounded Gauss linearUpwindV limitedGradFace;

    // Energy equations
    div(phi,e)      Gauss linearUpwind limitedGrad;
    div(phi,h)      bounded Gauss linearUpwind limitedGrad;
    div(phi,K)      bounded Gauss linearUpwind limitedGrad;

    // Pressure terms
    div(phid,p)     Gauss upwind;

    // Turbulence equations
    div(phi,k)      bounded Gauss linearUpwind limitedGrad;
    div(phi,omega)  bounded Gauss linearUpwind limitedGrad;

    // Viscous stress tensor
    div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
}
```

Listing C.27: Convection discretization schemes

Diffusion and Interpolation Schemes

```
laplacianSchemes
{
    default         Gauss linear limited 0.5;      // Laplacian operator
}

interpolationSchemes
{
    default         linear;                         // Cell-to-face interpolation
}

snGradSchemes
{
    default         limited 0.5;                   // Surface normal gradient
}

wallDist
{
    method          meshWave;                       // Wall distance calculation
}
```

Listing C.28: Diffusion and interpolation methods

Finite Volume Solution (fvSolution)

The solution dictionary defines solver settings and solution algorithms.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      fvSolution;
}

solvers
{
    "rho.*"                                         // Density equation
    {
        solver          diagonal;                   // Direct diagonal solver
    }

    "(p|pCorr).*"                                   // Pressure correction
    {
        solver          PBiCGStab;                  // Bi-conjugate gradient
        preconditioner  DILU;                       // Diagonal incomplete LU
        tolerance       1e-10;                      // Convergence tolerance
        relTol          0.001;                      // Relative tolerance
        maxIter         100;                        // Maximum iterations
        minIter         5;                          // Minimum iterations
    }

    "(U|k|omega).*"                                 // Velocity and turbulence
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-10;
        relTol          0.01;
        minIter         5;
        maxIter         20;
    }
}
```

Listing C.29: Linear solver configurations

Solution Algorithms(fvSolution)

```
SIMPLE
{
    nCorrectors                3;                  // Pressure corrector steps
    nNonOrthogonalCorrectors   2;                  // Non-orthogonal corrections

    consistent                 yes;                // Consistent SIMPLE
    transonic                  yes;                // Transonic corrections

    pMinFactor                 0.1;                // Minimum pressure factor
    pMaxFactor                 3.0;                // Maximum pressure factor
}

relaxationFactors
{
    fields
    {
        p                      0.5;                // Pressure under-relaxation
        U                      0.5;                // Velocity under-relaxation
    }
    equations
    {
        U                      0.4;                // Momentum equation
        "(h|e)"                0.5;                // Energy equation
        "(k|omega)"            0.5;                // Turbulence equations
    }
}
```

Listing C.30: SIMPLE algorithm settings

C.3.4 Mesh Generation Script (mesh.sh)

The mesh generation process is automated through a shell script that handles environment setup, geometry processing, and mesh creation using OpenFOAM's cartesianMesh utility.

Environment Setup and Initialization The environment setup section implements a robust initialization procedure that handles different OpenFOAM installation scenarios:

Directory Navigation and Safety

```
#!/bin/bash

# Change to script directory and exit if fails
cd "$(dirname "$0")" || exit 1
echo "Running in: $(pwd)"
```

Listing C.31: Script initialization and directory navigation

The script begins with a critical safety mechanism that ensures execution in the correct directory. The `$(dirname "$0")` expression extracts the directory path from the script's location, allowing the script to run in its intended directory regardless of where it's called from. The logical OR operator `|| exit 1` provides immediate termination if the directory change fails, preventing execution in an incorrect location that could lead to file system corruption or missing dependencies.

Environment Detection and Loading

```
# Load OpenFOAM v2406 Environment
echo "Loading OpenFOAM v2406 environment..."
if module avail openfoam/v2406 &>/dev/null; then
    # HPC cluster environment with module system
    module purge                                  # Clear existing modules
    source ~/v2406/OpenFOAM-v2406/etc/bashrc      # User installation
    source ~/OpenFOAM/OpenFOAM-v2406/etc/bashrc   # System installation
    numProcs=$SLURM_NTASKS                        # Get processor count from SLURM
    echo "OpenFOAM v2406 loaded via module with $numProcs processors."
else
```

```
    # Local installation fallback
    source /opt/openfoam2406/etc/bashrc            # Standard system path
    numProcs=4                                      # Default processor count
    echo "OpenFOAM v2406 loaded from /opt with $numProcs processors."
fi
```

Listing C.32: Adaptive OpenFOAM environment loading

The environment loading implements a dual-path strategy that automatically detects the computational environment type. The conditional check `module avail openfoam/v2406 &>/dev/null` determines if a module system is available, with output redirection preventing screen clutter during the detection process. For HPC cluster environments, the script first purges any existing modules to prevent version conflicts, then sources multiple bashrc files to ensure complete environment setup. The processor count is extracted from the SLURM scheduler variable, enabling automatic scaling for parallel execution. When no module system is detected, the script defaults to a standard local installation path and uses a conservative four-processor configuration suitable for desktop workstations.

Case Directory Preparation

```
echo "Cleaning case..."
rm -rf 0 processor* logs                           # Remove previous results
cp -r 0.orig 0                                     # Restore initial conditions
foamCleanTutorials                                  # OpenFOAM cleanup utility

echo "Creating logs directory..."
mkdir -p logs                                      # Create log directory

echo "Creating foam.foam file for ParaView compatibility..."
touch foam.foam                                    # ParaView recognition file
```

Listing C.33: Case directory cleanup and initialization

The cleanup and initialization phase establishes a clean working environment essential for reliable mesh generation. The removal of previous solution directories, processor decomposition data, and log files ensures that no artifacts from previous runs interfere with the current execution. The restoration of initial boundary conditions from the `0.orig` backup directory provides a pristine starting point, while the OpenFOAM `foamCleanTutorials` utility removes case-specific generated files that could cause conflicts. The creation of a logs directory with the `-p` flag prevents errors if the directory already exists, and the empty `foam.foam` file serves as a recognition marker that enables ParaView to properly identify and load the OpenFOAM case for post-processing visualization.

This comprehensive initialization ensures consistent execution across different computational platforms while maintaining case integrity and enabling seamless transition to post-processing workflows.

Background Mesh Generation

```
echo "Running blockMesh and logging to logs/blockMesh.log..."
blockMesh > logs/blockMesh.log 2>&1

surfaceFeatureExtract > logs/surfaceFeatureExtract.log 2>&1
gzip -d constant/triSurface/rocket.eMesh.gz
```

Listing C.34: Background mesh creation and surface feature extraction

The `blockMesh` utility creates a structured background mesh based on the `blockMeshDict` configuration. Surface features are extracted from the geometry files, and compressed mesh files are decompressed for processing.

Geometry Processing Pipeline

```
echo "Extracting boundary patches to STL (patches.stl)..."
foamToSurface -tri -constant ./constant/triSurface/patches.stl

echo "Renaming solid and endsolid entries in rocketV4.stl to 'rocket'..."
sed -i 's/^solid .*/solid rocket/; s/^endsolid .*/endsolid rocket/' \
    ./constant/triSurface/rocket.stl
```

```
echo "Combining rocket.stl and patches.stl into combined.stl..."
cat ./constant/triSurface/rocket.stl ./constant/triSurface/patches.stl > \
    ./constant/triSurface/combined.stl

echo "Converting combined.stl to FMS format (combined.fms)..."
surfaceToFMS ./constant/triSurface/combined.stl
```

Listing C.35: STL file processing and combination

The geometry processing involves several critical steps:

- **Boundary extraction**: The `foamToSurface` utility extracts boundary patches as STL triangular surfaces
- **Surface naming**: The `sed` command standardizes solid names in the STL files for consistent processing
- **Geometry combination**: Multiple STL files are concatenated to create a unified surface description
- **Format conversion**: The combined geometry is converted to FMS format for cartesianMesh compatibility

Cartesian Mesh Generation

```
echo "Running cartesianMesh..."
if ! cartesianMesh > logs/cartesianMesh.log 2>&1 ; then
    echo "cartesianMesh failed - aborting Mesh.sh" >&2
    exit 1
fi
gunzip ./constant/polyMesh/boundary.gz     #extract the .gz zip
foamDictionary ./constant/polyMesh/boundary -entry "entry0/inlet/type" -set "patch"
foamDictionary ./constant/polyMesh/boundary -entry "entry0/outlet/type" -set "patch"
foamDictionary ./constant/polyMesh/boundary -entry "entry0/sides/type" -set "patch"
foamDictionary ./constant/polyMesh/boundary -entry "entry0/rocket/type" -set "wall"

echo "Mesh generation pipeline completed successfully."
```

Listing C.36: Mesh generation and boundary configuration

The final phase executes the cartesian mesh generation with error checking. If `cartesianMesh` fails, the script terminates with an error message. Upon successful completion, the boundary file is decompressed and boundary patch types are configured using `foamDictionary`:

- **inlet, outlet, sides**: Set as `patch` type for flow boundaries
- **rocket**: Set as `wall` type for no-slip solid surfaces

The cartesianMesh utility generates an unstructured mesh by cutting the background Cartesian grid with the provided surface geometry, automatically handling complex geometries while maintaining good mesh quality near walls.

C.3.5 Complete Simulation Pipeline (allrun.sh)

The `allrun.sh` script orchestrates the entire CFD simulation workflow, from mesh preprocessing through solver execution to post-processing. This automation script integrates multiple OpenFOAM utilities in a sequential pipeline designed for parallel execution.

Script Initialization

```
#!/bin/bashW

# Navigate to script directory with error checking
cd "$(dirname "$0")" || exit 1
echo "Running in: $(pwd)"
```

Listing C.37: Script header and initialization

The script follows the same robust initialization pattern as the mesh generation script, ensuring execution in the correct directory context. The header comment provides a high-level overview of the pipeline stages, establishing the workflow sequence from mesh generation through solution execution.

Environment Configuration

```
# Load OpenFOAM v2406 Environment
echo "Loading OpenFOAM v2406 environment..."
if module avail openfoam/v2406 &>/dev/null; then
    module purge                                    # Clear existing modules
    source ~/v2406/OpenFOAM-v2406/etc/bashrc       # Load user environment
    source ~/OpenFOAM/OpenFOAM-v2406/etc/bashrc    # Load system environment
    numProcs=$SLURM_NTASKS                          # HPC processor allocation
    echo "OpenFOAM v2406 loaded via module with $numProcs processors."
else
    source /opt/openfoam2406/etc/bashrc            # Local installation
    numProcs=4                                      # Default local processors
    echo "OpenFOAM v2406 loaded from /opt with $numProcs processors."
fi
```

Listing C.38: OpenFOAM environment loading and processor configuration

The environment setup mirrors the mesh script implementation, providing seamless operation across different computational platforms. The processor count determination is crucial for subsequent parallel decomposition and MPI execution stages.

Case Directory Initialization

```
# Clean Case
echo "Cleaning case..."
rm -rf 0 processor*                                 # Remove previous results
cp -r 0.orig 0                                      # Restore initial conditions

mkdir -p logs                                       # Create logging directory
touch foam.foam                                     # ParaView compatibility file
```

Listing C.39: Case cleanup and preparation

The case preparation ensures a clean starting state by removing previous solution data and processor decomposition directories, then restoring the original boundary and initial conditions.

Mesh Preprocessing

```
# Step 6: Renumber
echo "Renumbering mesh..."
renumberMesh -overwrite > logs/renumberMesh.log 2>&1

# Step 7: Decompose final mesh
echo "Decomposing final mesh..."
subdiv_x=$((numProcs / 4))                          # Calculate x-direction subdivisions
foamDictionary system/decomposeParDict -entry "numberOfSubdomains" -set "$numProcs"
foamDictionary system/decomposeParDict -entry "n" -set "($subdiv_x 2 2)"
decomposePar > logs/decomposePar.log 2>&1
```

Listing C.40: Mesh renumbering and final decomposition

The active preprocessing begins with mesh renumbering to optimize memory access patterns and computational efficiency. The decomposition strategy calculates subdivision ratios based on the total processor count, implementing a structured decomposition pattern that assumes a 4:2:2 ratio scaling with processor availability.

Mesh Quality Assessment

```
# Step 8: checkMesh
echo "Running checkMesh..."
mpirun -np $numProcs checkMesh -parallel > logs/checkMesh.log 2>&1
```

```
mpirun -np $numProcs checkMesh -allGeometry -allTopology -parallel > logs/checkMeshAll.
    log 2>&1

# Step 9: foam.foam for ParaView
echo "Creating foam.foam..."
for d in processor*; do touch "$d/foam.foam"; done
```

Listing C.41: Comprehensive mesh quality checking

Mesh quality assessment occurs through two checkMesh executions: a standard check and a comprehensive analysis including all geometric and topological metrics. The creation of ParaView compatibility files in each processor directory enables visualization of parallel decomposed results.

Solver Configuration and Execution

```
# Copy stable numerical schemes
cp ./system/fvSchemes.stable ./system/fvSchemes

echo "Running rhoSimpleFoam in parallel..."
mpirun -np $numProcs rhoSimpleFoam -parallel > logs/rhoSimpleFoam.log 2>&1
```

Listing C.42: Solver execution with scheme switching

The solver execution includes a crucial scheme switching operation, replacing the default numerical schemes with a more stable configuration stored in `fvSchemes.stable`. This suggests a two-stage approach where initial mesh generation may use different schemes than the final solution process.

Results Processing

```
# Step 11: Reconstruct results
echo "Reconstructing latest time..."
reconstructPar -latestTime > logs/reconstructPar.log 2>&1

# Step 12: Post-process
echo "Sampling rocket patch to VTK..."
postProcess -func rocketVTK -latestTime > logs/rocketVTK.log 2>&1
```

Listing C.43: Solution reconstruction and post-processing

The post-processing phase reconstructs the parallel solution for unified analysis and executes specialized post-processing functions to extract surface data from the rocket geometry in VTK format suitable for advanced visualization and analysis.

This automated pipeline demonstrates sophisticated workflow management, combining robust error handling, platform adaptability, and comprehensive logging to ensure reliable execution across different computational environments.

C.3.6 ParaView Server Script(Pserver.sh)

The `Pserver.sh` script establishes a ParaView server instance on HPC cluster resources for remote visualization of OpenFOAM results.

```
#!/bin/bash

salloc --account=def-tembelym --time=02:00:00 --ntasks=1 --cpus-per-task=1 --mem=12G \
srun --ntasks=1 --cpus-per-task=1 --mem=12G --pty bash -c '
    module purge                                  # Clear existing modules
    module load paraview/5.13                     # Load ParaView visualization
    pvserver --force-offscreen-rendering          # Launch headless server
'
```

Listing C.44: ParaView server allocation and execution

Server Execution Execute the server script on the cluster:

```
# On the cluster terminal
$ ./Pserver.sh

# Server will display connection information:
# Accepting connection(s): nodename:11111
```

Listing C.45: Running ParaView server on cluster

Local Connection Setup From your local machine, establish an SSH tunnel to the cluster node:

```
# On local terminal (replace nodename with actual cluster node)
$ ssh user@servername.ca -L 11111:nodename:11111
```

Listing C.46: SSH tunnel establishment

ParaView Client Connection Open ParaView on your local machine and connect to the server:

```
1. Open ParaView (ensure version matches server: 5.13)
2. File  Connect
3. Add Server: localhost, Port: 11111
4. Connect to established server session
```

Listing C.47: ParaView client connection

This client-server architecture enables processing of large CFD datasets on cluster resources while maintaining local interactivity for visualization and analysis.

C.3.7 Simulation Parameters Configuration(parameters.cs)

The `parameters.cs` file defines the key simulation parameters used by the mesh generation and solver configuration scripts.

```
// **Define Parameters for Mesh Dimensions**
xMin -8;                                          // Domain upstream extent
xMax 30;                                          // Domain downstream extent

yMin -10;                                         // Domain lateral minimum
yMax  10;                                         // Domain lateral maximum

zMin -10;                                         // Domain vertical minimum
zMax  10;                                         // Domain vertical maximum

// **Define Number of Cells**
maxCellSize 0.5;                                  // Maximum cell dimension

P 69681;                                          // Total pressure (Pa)
T 268;                                            // Total temperature (K)

Ma      1.2;                                      // Mach number
gamma   1.4;                                      // Specific heat ratio
R       287.05;                                   // Gas constant (J/kgK)

AoA 0.0;                                          // Angle of attack (degrees)

WriteInterval 100;                                // Solution output frequency
```

Listing C.48: Global simulation parameters

The parameter file establishes a computational domain extending 8 units upstream and 30 units downstream of the geometry, with symmetric lateral and vertical boundaries spanning 20 units each. The supersonic flow conditions are defined by a Mach number of 1.2 at standard atmospheric conditions, representing typical rocket nozzle operating conditions.

C.3.8 HPC Job Submission Script(submit.sh)

The `submit.sh` script manages the complete simulation workflow execution on SLURM-based HPC systems.

```
#!/bin/bash
#SBATCH --job-name=CR25                                  # Job identifier
#SBATCH --mem=68G                                        # Total memory allocation
#SBATCH --nodes=1                                        # Single node execution
#SBATCH --ntasks=64                                      # Parallel task count
#SBATCH --cpus-per-task=1                                # CPU cores per task
#SBATCH --time=20:0:0                                    # Maximum runtime (20 hours)
#SBATCH --mail-user=darasamii@gmail.com                  # Notification email
#SBATCH --mail-type=ALL                                  # Email all job events
#SBATCH --account=def-tembelym                           # Research allocation account
#SBATCH --output=submit.log                              # Job output log file

./Mesh.sh                                                # Execute mesh generation
./Allrun.sh                                              # Execute complete simulation
```

Listing C.49: SLURM job submission configuration

The submission script configures a high-performance computing job with 64 parallel tasks and 68GB memory allocation, suitable for large-scale CFD simulations. The 20-hour time limit accommodates extended mesh generation and iterative solution processes. The script executes the complete workflow by first running the mesh generation script followed by the full simulation pipeline.

The commented alternative mesh script `MeshPractice.sh` suggests different meshing strategies may be available for testing purposes. The email notification system provides job status updates, while the dedicated log file captures all submission-related output for debugging and monitoring purposes.

Job Submission and Monitoring Submit the job to the SLURM scheduler:

```
# Submit the job to SLURM queue
$ sbatch submit.sh

# Monitor job status and queue position
$ squeue -u $USER

# Expected output shows job information:
 JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
 12345    normal     CR25 username  R    1:23:45      1 node001
```

Listing C.50: Job submission and status monitoring

The `sbatch` command submits the job script to the SLURM scheduler and returns a job ID for tracking. The `squeue` command displays current job status where `ST` indicates job state: `PD` (pending), `R` (running), or `CG` (completing). The `TIME` field shows elapsed runtime, while `NODELIST` indicates assigned compute nodes.

Appendix D

Postprocessing scripts

Imports

importing necessary packages for postprocessing the Raw CFD Data

In [120…
```
import numpy as np
import pandas as pd
import re
import os
from pathlib import Path
import glob



from scipy.optimize import root
from itertools import combinations
import pyvista as pv

import matplotlib.pyplot as plt
from matplotlib.patches import FancyArrowPatch
import matplotlib as mpl
import matplotlib.lines as mlines
import matplotlib.tri as mtri
import matplotlib.patches as mpatches
from mpl_toolkits.axes_grid1.inset_locator import inset_axes

from matplotlib.ticker import LogLocator, NullFormatter


%matplotlib inline
```

Updating the defualt configuration of the matplotlib ploting package

In [ ]:
```
mpl.rcParams.update({
    'text.usetex': False,
    'mathtext.fontset': 'cm',  # Computer Modern font
    'font.family': 'serif',
    'axes.labelsize': 12,
    'font.size': 12,
    'legend.fontsize': 10,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10
})
```

Utility Functions

In [3]:
```
def load_dat(path, verbose=False):
    """
    Load a .dat file into a pandas DataFrame, handling comment lines and extracting column names
    from the last commented line.

    Parameters:
    - path: Path to the .dat file.

    Returns:
    - df: pandas DataFrame containing the data.
    """
    # Read the file line by line to handle comments and column names
    with open(path, 'r') as file:
        lines = file.readlines()

    # Extract column names from the last line starting with '#'
    coments = [line for line in lines if line.startswith('#')]
    column_line = [line for line in lines if line.startswith('#')][-1]
    column_names = column_line.strip('#').strip().split()

    # Read the data, skipping all comment lines
    df = pd.read_csv(
        path,
        comment='#',           # Ignores all lines that begin with '#'
        delim_whitespace=True, # Split columns by whitespace
        header=None,           # No header row in the data portion
        names=column_names     # Assign column names from the last '#'
    )

    if verbose:
        for i in coments:
            print(i)
    return df
```

In [4]:
```python
def extract_cell_counts(log_path):
    """
    Extracts cell counts from a checkMesh.log file.

    Returns a dictionary like:
    {
        'hexahedra': 16850459,
        'prisms': 32,
        ...
    }
    """
    cell_counts = {}
    start_marker = "Overall number of cells of each type:"
    pattern = r"(\w[\w ]*?):\s+(\d+)"

    with open(log_path, "r") as f:
        lines = f.readlines()

    capturing = False
    for line in lines:
        if start_marker in line:
            capturing = True
            continue

        if capturing:
            if line.strip() == "":
                break  # end of section
            match = re.match(pattern, line.strip())
            if match:
                cell_type = match.group(1).strip()
                count = int(match.group(2))
                cell_counts[cell_type] = count

    return cell_counts
```

In [5]:
```python
def extract_all_times(log_path):
    """
    Extracts ExecutionTime and ClockTime from a rhoSimpleFoam log file.

    Returns:
        {
            'ExecutionTime': [exec1, exec2, ...],
            'ClockTime': [clock1, clock2, ...]
        }
    """
    times = {
        'ExecutionTime': [],
        'ClockTime': []
    }

    pattern = r"ExecutionTime\s*=\s*([\d.]+)\s*s\s+ClockTime\s*=\s*(\d+)\s*s"

    with open(log_path, "r") as f:
        for line in f:
            match = re.search(pattern, line)
            if match:
                exec_time = float(match.group(1))
                clock_time = int(match.group(2))
                times['ExecutionTime'].append(exec_time)
                times['ClockTime'].append(clock_time)

    return times
```

In [6]:
```python
def summarize_cases(base_dir):
    rows = []

    for case_name in sorted(os.listdir(base_dir)):
        case_path = os.path.join(base_dir, case_name)
        if not os.path.isdir(case_path):
            continue

        # 1. Mesh info
        mesh_log = os.path.join(case_path, 'logs', 'checkMesh.log')
        try:
            cell_counts = extract_cell_counts(mesh_log)
            mesh_total = sum(cell_counts.values())
        except Exception:
            mesh_total = np.nan

        # 2. Solver times
        foam_log = os.path.join(case_path, 'logs', 'rhoSimpleFoam.log')
        try:
            times = extract_all_times(foam_log)
            exec_time = np.diff(times['ExecutionTime'], prepend=0, append=0)[1:-1].mean()
            clock_time = np.diff(times['ClockTime'], prepend=0, append=0)[1:-1].mean()
        except Exception:
```

```
            exec_time = np.nan
            clock_time = np.nan

        # 3. Final Cd from forceCoeffs/latest/coefficient.dat
        coeff_path = os.path.join(case_path, 'postProcessing', 'forceCoeffs')
        Cd = np.nan
        if os.path.exists(coeff_path):
            subdirs = [d for d in os.listdir(coeff_path) if os.path.isdir(os.path.join(coeff_path, d))]
            if subdirs:
                latest = sorted(subdirs, key=lambda x: float(x))[-1]
                dat_file = os.path.join(coeff_path, latest, 'coefficient.dat')
                if os.path.exists(dat_file):
                    try:
                        df = load_dat(dat_file)
                        Cd = df["Cd"].iloc[-1]
                    except Exception:
                        pass

        # 4. Final max y+ from yPlus1/latest/yPlus.dat
        yplus_path = os.path.join(case_path, 'postProcessing', 'yPlus1')
        max_yplus = np.nan
        if os.path.exists(yplus_path):
            subdirs = [d for d in os.listdir(yplus_path) if os.path.isdir(os.path.join(yplus_path, d))]
            if subdirs:
                latest = sorted(subdirs, key=lambda x: float(x))[-1]
                dat_file = os.path.join(yplus_path, latest, 'yPlus.dat')
                if os.path.exists(dat_file):
                    try:
                        df = load_dat(dat_file)
                        max_yplus = df["max"].iloc[-1]
                    except Exception:
                        pass

        # 5. Append row
        rows.append({
            "case_name": case_name,
            "mesh_cells": mesh_total,
            "exec_time": exec_time,
            "clock_time": clock_time,
            "Cd": Cd,
            "max_yplus": max_yplus
        })

    return pd.DataFrame(rows)
```

Test utility tools

In [ ]:
```
extract_cell_counts("Data/CollectedResultsMesh/mesh_0p4/logs/checkMesh.log")
```

Out[ ]:
```
{'hexahedra': 16850459,
 'prisms': 32,
 'wedges': 0,
 'pyramids': 40,
 'tet wedges': 0,
 'tetrahedra': 16,
 'polyhedra': 277564}
```

In [9]:
```
Volume = 38 * 20 * 20  # Volume of the mesh in mm^3

df = summarize_cases("/home/dara/Professional/projects/spaceConcordia/CR25/Data/CollectedResultsMesh/")
df['h'] =  df['mesh_cells'].apply(lambda x: ((Volume)/x)**(1/3) if x > 0 else np.nan)


df.head(10)
```

Out[9]:

	case_name	mesh_cells	exec_time	clock_time	Cd	max_yplus	h
0	mesh_0p3	39141125	47.359798	48.357095	0.399830	49.470680	0.072958
1	mesh_0p35	25982239	40.090506	41.045855	0.406857	53.070195	0.083635
2	mesh_0p4	17128111	30.602828	31.148768	0.415331	56.998662	0.096097
3	mesh_0p45	13081897	16.943099	17.304455	0.402429	68.024968	0.105129
4	mesh_0p5	9100227	10.587603	11.124181	0.444002	182.734021	0.118649
5	mesh_0p55	7394267	8.423145	8.673959	0.409703	72.718618	0.127150

In [23]:
```
df_clan = df.drop([4])
df_clan
```

Out[23]:

	case_name	mesh_cells	exec_time	clock_time	Cd	max_yplus	h
0	mesh_0p3	39141125	47.359798	48.357095	0.399830	49.470680	0.072958
1	mesh_0p35	25982239	40.090506	41.045855	0.406857	53.070195	0.083635
2	mesh_0p4	17128111	30.602828	31.148768	0.415331	56.998662	0.096097
3	mesh_0p45	13081897	16.943099	17.304455	0.402429	68.024968	0.105129
5	mesh_0p55	7394267	8.423145	8.673959	0.409703	72.718618	0.127150

Mesh Analysis

In [18]:

```python
def richardson_triplet(h1, h2, h3, phi1, phi2, phi3, Fs=1.25,verbose=False):
    '''
    h3: coarse
    h2: medium
    h1: fine
    '''
    # Bundle and sort by decreasing h (coarse → fine)
    sorted_data = sorted(zip([h1, h2, h3], [phi1, phi2, phi3]), key=lambda x: -x[0])
    (h3, phi3), (h2, phi2), (h1, phi1) = sorted_data

    # Compute refinement ratios
    r32 = h3 / h2
    r21 = h2 / h1

    # Error differences
    e21 = phi2 - phi1
    e32 = phi3 - phi2

    # Sign factor
    s = np.sign(e32 / e21)


    if e21 == 0 or e32 == 0 or np.sign(e32 / e21) == 0:
        raise RuntimeError("Unstable error ratios: division or log undefined.")

    # System of equations
    def equations(x):
        p, q = x
        try:
            eq1 = q - np.log((r21**p - s) / (r32**p - s))
            eq2 = p - (1 / np.log(r21)) * abs(np.log(np.abs(e32 / e21)) + q)
            if verbose:
                print(f"  ↪ p = {p:.6f}, q = {q:.6f}, eq1 = {eq1:.2e}, eq2 = {eq2:.2e}")
            return [eq1, eq2]
        except Exception as e:
            print(f"  ✗ Exception in equations: {e}")
            raise


    # Initial guess
    x0 = [1.0, 0.1]

    # Solve nonlinear system
    sol = root(equations, x0)

    if not sol.success:
        raise RuntimeError("Failed to converge: " + sol.message)

    p, q = sol.x

    # Extrapolated solution using fine and medium
    phi_ext = phi1 + (phi1 - phi2) / (r21**p - 1)

    # GCI for fine–medium and medium–coarse
    GCI_12 = Fs * abs(phi1 - phi2) / abs(phi1) / abs(r21**p - 1)
    GCI_23 = Fs * abs(phi2 - phi3) / abs(phi2) / abs(r32**p - 1)

    return {
        'p': p,
        'q': q,
        's': s,
        'r21': r21,
        'r32': r32,
        'e21': e21,
        'e32': e32,
        'phi_ext': phi_ext,
        'GCI_12 (%)': GCI_12 * 100,
        'GCI_23 (%)': GCI_23 * 100,
    }
```

In [19]:
```
def analyze_richardson_dataset(h_list, phi_list, tolerance=0.05,verbose=True):
    results = []

    # Iterate over all unique triplets (coarse, medium, fine)
    for i, j, k in combinations(range(len(h_list)), 3):
        # Extract values
        hs = [h_list[i], h_list[j], h_list[k]]
        phis = [phi_list[i], phi_list[j], phi_list[k]]

        try:
            # Run triplet Richardson analysis
            result = richardson_triplet(*hs, *phis,verbose=verbose)
            result['Triplet'] = f"[{i}-{j}-{k}]"
            result['h1'] = hs[0]
            result['h2'] = hs[1]
            result['h3'] = hs[2]
            result['phi1'] = phis[0]
            result['phi2'] = phis[1]
            result['phi3'] = phis[2]
            results.append(result)
        except Exception as e:
            results.append({'Triplet': f"[{i}-{j}-{k}]", 'Error': str(e)})



    # Convert to DataFrame
    df = pd.DataFrame(results)

    # Drop failed rows
    df_valid = df.dropna(subset=['phi_ext']).copy()

    # Optionally average close phi_ext values
    if not df_valid.empty:
        phi_exts = df_valid['phi_ext'].values
        ref = np.median(phi_exts)
        within_tol = np.abs((phi_exts - ref) / ref) < tolerance
        phi_avg = np.mean(phi_exts[within_tol])
        phi_std = np.std(phi_exts[within_tol])
        print(f"\nAverage extrapolated φ (within {tolerance*100:.1f}%): {phi_avg:.6f} ± {phi_std:.6f} from {withi
    else:
        print("No valid triplets found.")

    return df
```

Test Mesh functions

In [21]:
```
h_list = df.h.values
phi_list = df.Cd.values

df1 = analyze_richardson_dataset(h_list, phi_list, tolerance=0.05,verbose=False)
df1.dropna(subset=['phi_ext'], inplace=True)
```

```
Average extrapolated φ (within 5.0%): 0.395681 ± 0.005935 from 9 triplets
```

In [22]:
```
dfFinal = df1[(df1['r21'] > 1.1) & (df1['r32'] > 1.1)]
dfFinal
```

Out[22]:

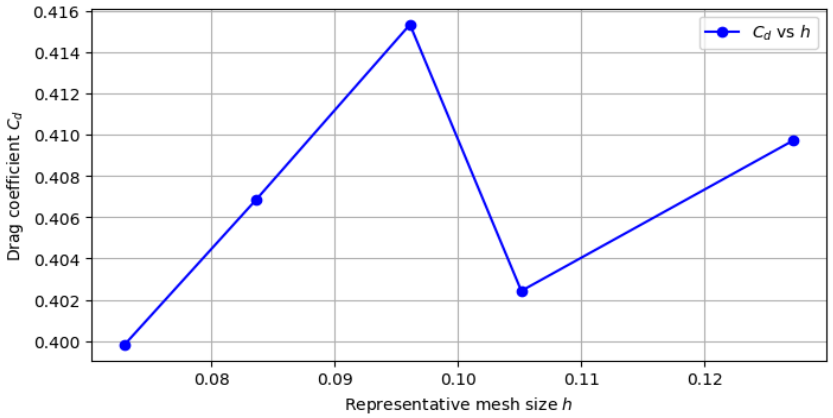
	p	q	s	r21	r32	e21	e32	phi_ext	GCI_12 (%)	GCI_23 (%)	Triplet	h1	h2	
0	1.237256	-0.018245	1.0	1.146354	1.149006	0.007027	0.008474	0.361664	11.931876	13.884660	[0-1-2]	0.072958	0.083635	0.0960
1	7.206109	-0.522418	-1.0	1.146354	1.257000	0.007027	-0.004428	0.395637	1.310871	0.324060	[0-1-3]	0.072958	0.083635	0.1051
2	2.839794	-1.277233	1.0	1.146354	1.418648	0.007027	0.037145	0.385001	4.636054	6.714842	[0-1-4]	0.072958	0.083635	0.1186
5	3.702939	0.405061	1.0	1.317168	1.234674	0.015500	0.028672	0.391090	2.732457	7.295572	[0-2-4]	0.072958	0.096097	0.1186
6	3.723023	-0.012417	-1.0	1.317168	1.323137	0.015500	-0.005627	0.391165	2.708952	0.922369	[0-2-5]	0.072958	0.096097	0.1271
7	23.415816	5.781836	1.0	1.440968	1.128599	0.002599	0.041573	0.399830	0.000157	0.807455	[0-3-4]	0.072958	0.105129	0.1186
8	6.667542	1.406593	1.0	1.440968	1.209461	0.002599	0.007274	0.399581	0.077947	0.884742	[0-3-5]	0.072958	0.105129	0.1271
11	4.466897	-0.598504	1.0	1.149006	1.234674	0.008474	0.028672	0.397001	3.028096	5.516596	[1-2-4]	0.083635	0.096097	0.1186
13	17.739944	1.818054	-1.0	1.257000	1.128599	-0.004428	0.041573	0.406935	0.023934	1.709988	[1-3-4]	0.083635	0.105129	0.1186
14	2.425594	0.058383	-1.0	1.257000	1.209461	-0.004428	0.007274	0.412828	1.834349	3.854875	[1-3-5]	0.083635	0.105129	0.1271

In [25]:
```
meanCd = dfFinal['phi_ext'].mean()
df_clan['error'] = np.abs(df_clan['Cd'] - meanCd) / meanCd * 100
print(meanCd)
```

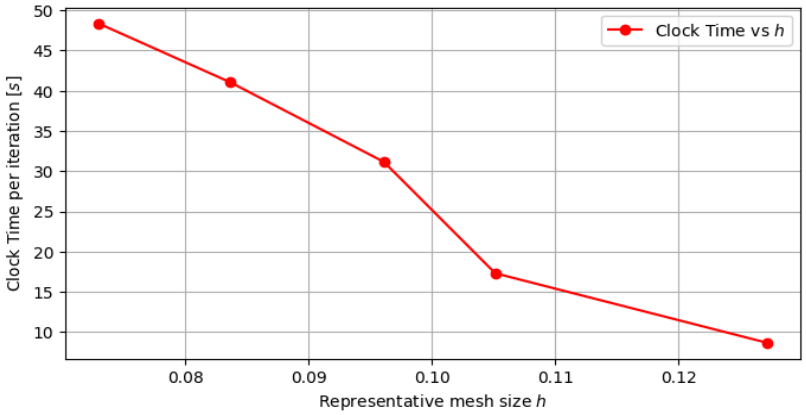
```
0.3940732261409317
```

Ploting the mesh Independency graphs

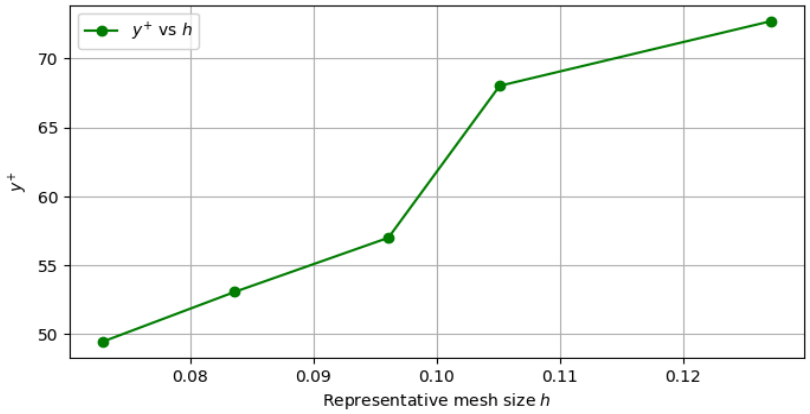
In [27]:
```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(df_clan['h'], df_clan['Cd'], 'b-o', label=r'$C_d$ vs $h$')

ax.set_xlabel(r'Representative mesh size $h$')
ax.set_ylabel(r'Drag coefficient $C_d$')
ax.grid(True)
ax.legend()

plt.savefig("./results/CdMesh.pdf")
```



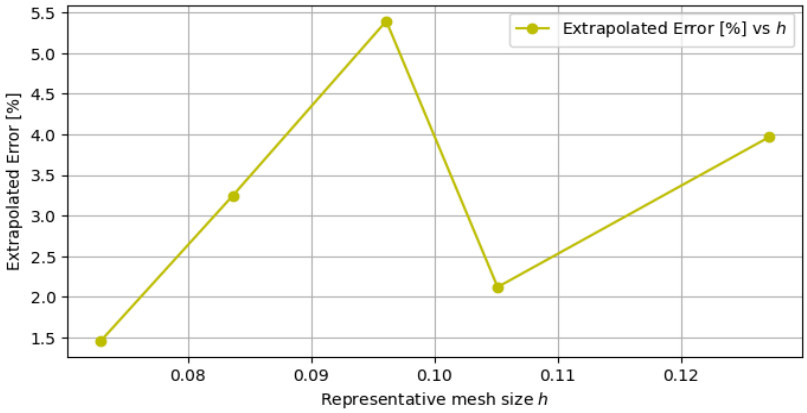
In [28]:
```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(df_clan['h'], df_clan['clock_time'], 'r-o', label=r'Clock Time vs $h$')

ax.set_xlabel(r'Representative mesh size $h$')
ax.set_ylabel(r'Clock Time per iteration [$s$]')
ax.grid(True)
ax.legend()

plt.savefig("./results/ClockTimeMesh.pdf")
```



In [29]:
```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(df_clan['h'], df_clan['max_yplus'], 'g-o', label=r'$y^{+}$ vs $h$')

ax.set_xlabel(r'Representative mesh size $h$')
ax.set_ylabel(r'$y^{+}$')
ax.grid(True)
ax.legend()

plt.savefig("./results/yPlusMesh.pdf")
```



In [30]:
```
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(df_clan['h'], df_clan['error'], 'y-o', label=r'Extrapolated Error [%] vs $h$')

ax.set_xlabel(r'Representative mesh size $h$')
ax.set_ylabel(r'Extrapolated Error [%]')
ax.grid(True)
ax.legend()

plt.savefig("./results/errorMesh.pdf")
```



Mach Angle Contours

In [123…
```
def compute_center_of_pressure(mesh, pressure_field='p'):
    pressure = mesh[pressure_field]
    cell_centers = mesh.cell_centers()
    cell_areas = mesh.compute_cell_sizes()['Area']
    mesh_n = mesh.compute_normals(cell_normals=True, point_normals=False)
    normals = mesh_n['Normals']
    if pressure.size == mesh.n_points:
        mesh_copy = mesh.copy()
        mesh_copy[pressure_field] = pressure
        cell_data = mesh_copy.cell_data_to_point_data().point_data_to_cell_data()
        pressure_cells = cell_data[pressure_field]
    else:
        pressure_cells = pressure
    ref = np.array([mesh.bounds[0], 0, 0])
    forces = pressure_cells[:,None] * cell_areas[:,None] * (-normals)
    total_force = forces.sum(axis=0)
    centers = cell_centers.points
    moments = np.cross(centers - ref, forces)
    total_moment = moments.sum(axis=0)
    cp_x = ref[0] + total_moment[2]/total_force[1] if abs(total_force[1]) > 1e-10 else ref[0]
    cp_y = ref[1] - total_moment[2]/total_force[0] if abs(total_force[0]) > 1e-10 else ref[1]
    return np.array([cp_x, cp_y, ref[2]])
```

In [124…
```
def plot_mach_contour_with_cp_cg(sliceZ, rocket, cmap='jet', Ma=1.2, cg=1.8):
    Cop = compute_center_of_pressure(rocket, pressure_field='p')
    cp_x, cp_y = Cop[0], Cop[1]
```

```
    tri_mesh = sliceZ.triangulate()
    faces = tri_mesh.faces.reshape(-1, 4)[:, 1:]

    pts = tri_mesh.points
    x, y = pts[:, 0], pts[:, 1]
    var = tri_mesh['Ma']

    triang = mtri.Triangulation(x, y, faces)
    fig, ax = plt.subplots(figsize=(20, 14), facecolor='white')
    cont = ax.tricontourf(triang, var, cmap=cmap, levels=400)

    theta = np.degrees(np.arcsin(1 / Ma))
    L = 15
    x_off, y_off = L * np.cos(np.radians(theta)), L * np.sin(np.radians(theta))

    ax.plot([0, x_off], [0, y_off], '--k')
    ax.plot([0, x_off], [0, -y_off], '--k')
    if Ma > 1:
        ax.text(-1.2, 0.5, f"Shock Angle={theta:.2f}°", color='black')

    ax.plot(cp_x, 0.0, 'ro', markersize=8)
    ax.text(cp_x - 0.1, 0.3, 'CoP', color='red', fontsize=14)
    arrow = FancyArrowPatch((0.0, -0.2), (cp_x, -0.2), arrowstyle='<->', color='red',
                            linewidth=1, mutation_scale=20)
    ax.add_patch(arrow)
    ax.text(cp_x / 2 - 0.1, -0.4, f'{cp_x:.2f} m', color='red', fontsize=14)

    ax.plot(cg, 0.0, 'bo', markersize=8)
    ax.text(cg - 0.1, 0.3, 'CoG', color='blue', fontsize=14)
    arrow = FancyArrowPatch((0.0, 0.2), (cg, 0.2), arrowstyle='<->', color='blue',
                            linewidth=1, mutation_scale=20)
    ax.add_patch(arrow)
    ax.text(cg / 2 - 0.1, 0.3, f'{cg:.2f} m', color='blue', fontsize=14)

  # Add origin anchor (X and Z directions)
    origin_x, origin_y = -1.8, -2.5
    dx, dz = 0.5, 0.5

    # X direction arrow
    ax.annotate('', xy=(origin_x + dx, origin_y), xytext=(origin_x, origin_y),
                arrowprops=dict(facecolor='black', shrink=0.0, width=2, headwidth=8))
    ax.text(origin_x + dx + 0.1, origin_y, 'x', fontsize=14, ha='center', va='center')

    # Z direction arrow
    ax.annotate('', xy=(origin_x, origin_y + dz), xytext=(origin_x, origin_y),
                arrowprops=dict(facecolor='black', shrink=0.0, width=2, headwidth=8))
    ax.text(origin_x, origin_y + dz + 0.1, 'z', fontsize=14, ha='center', va='center')


    ax.set_xlim(-2, 10)
    ax.set_ylim(-3, 3)
    ax.set_aspect('equal')
    ax.axis('off')
    cb = fig.colorbar(cont, ax=ax, orientation='horizontal', fraction=0.046, pad=0.04)
    cb.set_label('Mach number')

    plt.tight_layout()
    plt.show()

    return fig
```

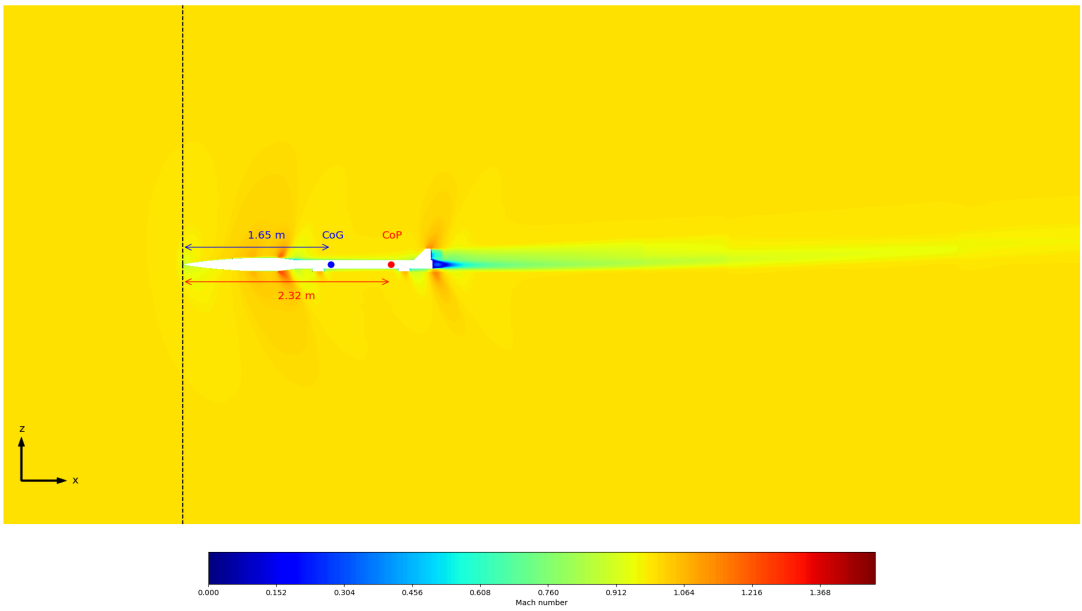
In [125…

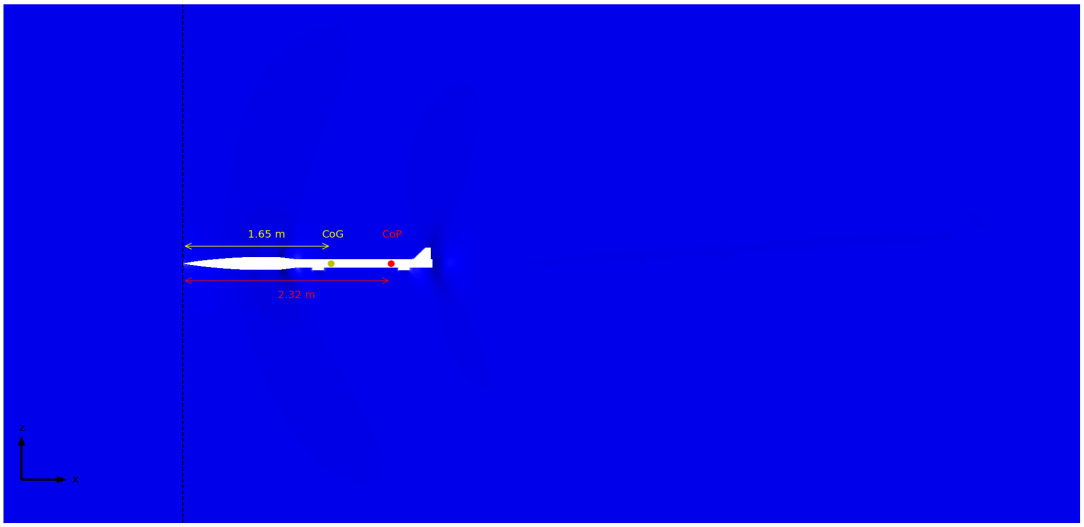
```
def plot_pressure_contour_with_cp_cg(sliceZ, rocket, cmap='seismic', Ma=1.2, cg=1.72):
    Cop = compute_center_of_pressure(rocket, pressure_field='p')
    cp_x, cp_y = Cop[0], Cop[1]

    tri_mesh = sliceZ.triangulate()
    faces = tri_mesh.faces.reshape(-1, 4)[:, 1:]

    pts = tri_mesh.points
    x, y = pts[:, 0], pts[:, 1]
    var = tri_mesh['p']

    triang = mtri.Triangulation(x, y, faces)
    fig, ax = plt.subplots(figsize=(20, 14), facecolor='white')
    cont = ax.tricontourf(triang, var, cmap=cmap, levels=400)

    theta = np.degrees(np.arcsin(1 / Ma))
    L = 15
    x_off, y_off = L * np.cos(np.radians(theta)), L * np.sin(np.radians(theta))
    ax.plot([0, x_off], [0, y_off], '--k')
    ax.plot([0, x_off], [0, -y_off], '--k')
    if Ma > 1:
        ax.text(-1.2, 0.5, f"Shock Angle={theta:.2f}°", color='black')

    ax.plot(cp_x, 0.0, 'ro', markersize=8)
    ax.text(cp_x - 0.1, 0.3, 'CoP', color='red', fontsize=14)
    arrow = FancyArrowPatch((0.0, -0.2), (cp_x, -0.2), arrowstyle='<->',
```

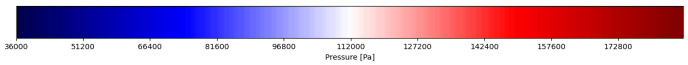
```
                              color='red', linewidth=1, mutation_scale=20)
    ax.add_patch(arrow)
    ax.text(cp_x / 2 - 0.1, -0.4, f'{cp_x:.2f} m', color='red', fontsize=14)

    ax.plot(cg, 0.0, 'yo', markersize=8)
    ax.text(cg - 0.1, 0.3, 'CoG', color='yellow', fontsize=14)
    arrow = FancyArrowPatch((0.0, 0.2), (cg, 0.2), arrowstyle='<->',
                              color='yellow', linewidth=1, mutation_scale=20)
    ax.add_patch(arrow)
    ax.text(cg / 2 - 0.1, 0.3, f'{cg:.2f} m', color='yellow', fontsize=14)


  # Add origin anchor (X and Z directions)
    origin_x, origin_y = -1.8, -2.5
    dx, dz = 0.5, 0.5

    # X direction arrow
    ax.annotate('', xy=(origin_x + dx, origin_y), xytext=(origin_x, origin_y),
                arrowprops=dict(facecolor='black', shrink=0.0, width=2, headwidth=8))
    ax.text(origin_x + dx + 0.1, origin_y, 'x', fontsize=14, ha='center', va='center')

    # Z direction arrow
    ax.annotate('', xy=(origin_x, origin_y + dz), xytext=(origin_x, origin_y),
                arrowprops=dict(facecolor='black', shrink=0.0, width=2, headwidth=8))
    ax.text(origin_x, origin_y + dz + 0.1, 'z', fontsize=14, ha='center', va='center')


    ax.set_xlim(-2, 10)
    ax.set_ylim(-3, 3)
    ax.set_aspect('equal')
    ax.axis('off')
    cb = fig.colorbar(cont, ax=ax, orientation='horizontal', fraction=0.046, pad=0.04)
    cb.set_label('Pressure [Pa]')

    plt.tight_layout()
    plt.show()

    return fig
```

Ploting the contours

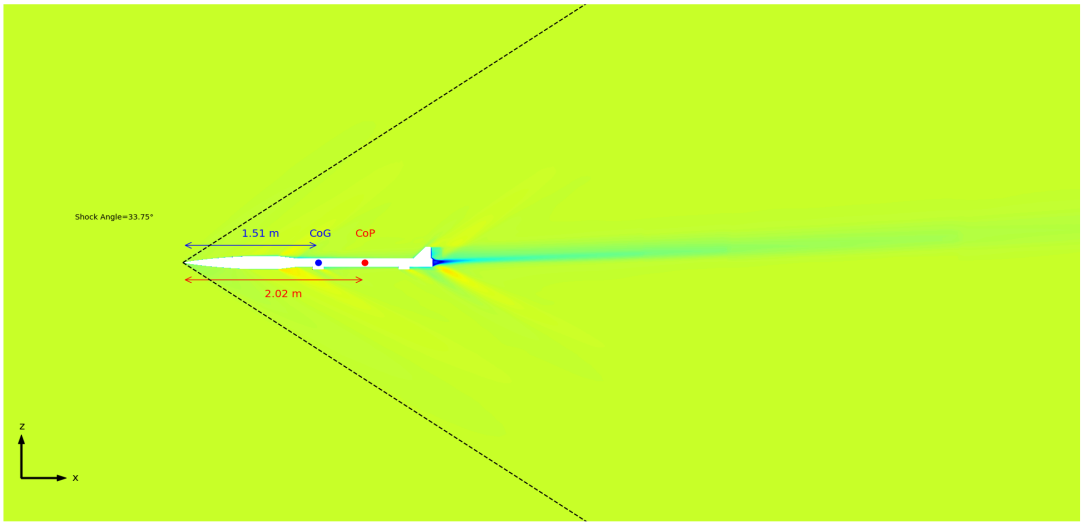
In [35]:
```
case = "AoA_5p0_Ma_1p0"
cg = 1.65

t = os.listdir(f"Data/CollectedResultsMesh_New/{case}/postProcessing/sliceCentreZ/")[0]

sliceZ = pv.read(f"Data/CollectedResultsMesh_New/{case}/postProcessing/sliceCentreZ/{t}/planeSlice.vtp")
rocket = pv.read(f"Data/CollectedResultsMesh_New/{case}/postProcessing/rocketVTK/{t}/rocketWall.vtp")

_ = plot_mach_contour_with_cp_cg(sliceZ, rocket, cmap='jet', Ma=1, cg=cg )

_ = plot_pressure_contour_with_cp_cg(sliceZ, rocket, cmap='seismic', Ma=1, cg=cg)
```

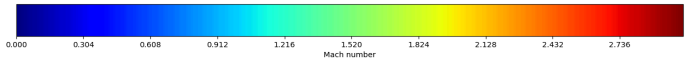


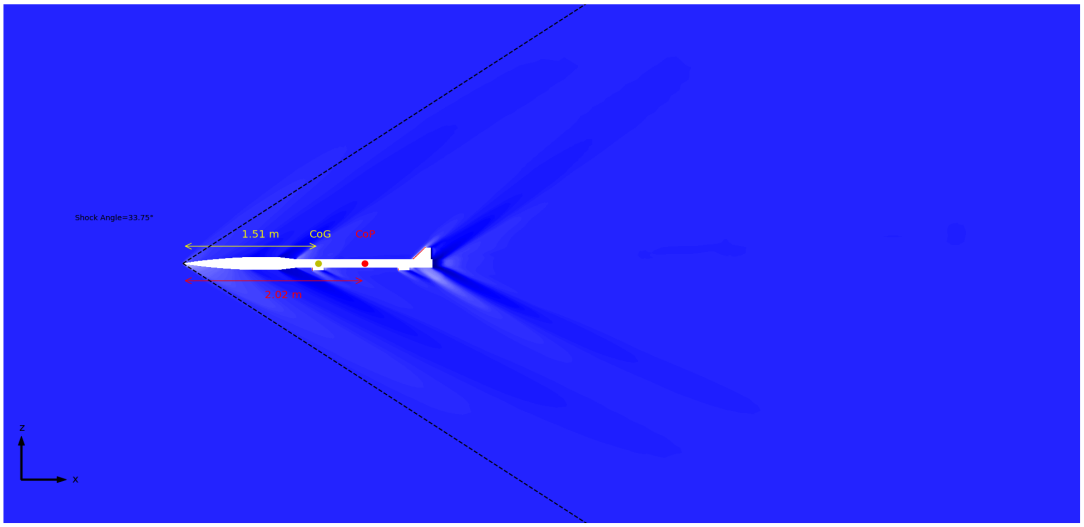


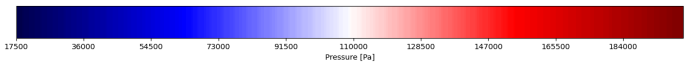


In [36]:
```
case = "AoA_5p0_Ma_1p8"
cg = 1.51

t = os.listdir(f"Data/CollectedResultsMesh_New/{case}/postProcessing/sliceCentreZ/")[0]

sliceZ = pv.read(f"Data/CollectedResultsMesh_New/{case}/postProcessing/sliceCentreZ/{t}/planeSlice.vtp")
rocket = pv.read(f"Data/CollectedResultsMesh_New/{case}/postProcessing/rocketVTK/{t}/rocketWall.vtp")

_ = plot_mach_contour_with_cp_cg(sliceZ, rocket, cmap='jet', Ma=1.8, cg=cg )

_ = plot_pressure_contour_with_cp_cg(sliceZ, rocket, cmap='seismic', Ma=1.8, cg=cg)
```









Center of Pressure Calculations

In [37]:
```python
def compute_center_of_pressure(mesh, pressure_field='p', reference_point=None):
    """
    Compute the center of pressure for a rocket from CFD results.

    Parameters:
    -----------
    mesh : pyvista.PolyData
        The surface mesh with pressure data
    pressure_field : str
        Name of the pressure field in the mesh data
    reference_point : array-like, optional
        Reference point for moment calculation (default: nose of rocket)

    Returns:
    --------
    dict : Dictionary containing center of pressure results
    """

    # Check if pressure data exists
    if pressure_field not in mesh.array_names:
        print(f"Available arrays: {mesh.array_names}")
        raise ValueError(f"Pressure field '{pressure_field}' not found in mesh")

    # Get pressure data and check if it's point or cell data
    pressure = mesh[pressure_field]

    print(f"Pressure data shape: {pressure.shape}")
    print(f"Number of cells: {mesh.n_cells}")
    print(f"Number of points: {mesh.n_points}")

    # Compute cell centers and areas
    cell_centers = mesh.cell_centers()
    cell_areas = mesh.compute_cell_sizes()['Area']

    # Compute surface normals (pointing outward from rocket)
    mesh_with_normals = mesh.compute_normals(cell_normals=True, point_normals=False)
    normals = mesh_with_normals['Normals']

    # Handle pressure data - convert to cell data if it's point data
    if len(pressure) == mesh.n_points:
        print("Converting point data to cell data...")
        # Interpolate point data to cell centers
        mesh_with_pressure = mesh.copy()
        mesh_with_pressure[pressure_field] = pressure
        cell_mesh = mesh_with_pressure.cell_data_to_point_data().point_data_to_cell_data()
        pressure_cells = cell_mesh[pressure_field]
    elif len(pressure) == mesh.n_cells:
        print("Using cell data directly...")
        pressure_cells = pressure
    else:
```

```
        raise ValueError(f"Pressure data size ({len(pressure)}) doesn't match cells ({mesh.n_cells}) or points (

    # Set reference point (typically nose of rocket)
    if reference_point is None:
        # Use the point with minimum x-coordinate (nose)
        reference_point = np.array([mesh.bounds[0], 0, 0])

    # Compute pressure forces on each cell
    # Force = pressure * area * normal (pointing into the fluid)
    pressure_forces = pressure_cells[:, np.newaxis] * cell_areas[:, np.newaxis] * (-normals)

    # Total force components
    total_force = np.sum(pressure_forces, axis=0)

    # Compute moments about reference point
    # Get cell center coordinates
    centers = cell_centers.points

    # Moment arm from reference point to each cell center
    moment_arms = centers - reference_point

    # Compute moments for each cell
    moments = np.cross(moment_arms, pressure_forces)

    # Total moment
    total_moment = np.sum(moments, axis=0)

    # Compute center of pressure
    # For a rocket, we're primarily interested in the longitudinal position (x-direction)
    # CP_x = reference_x + M_z / F_y (for pitching moment)
    # CP_y = reference_y - M_z / F_x (for yawing moment)

    results = {
        'total_force': total_force,
        'total_moment': total_moment,
        'reference_point': reference_point,
        #'pressure_forces': pressure_forces,
        #'cell_centers': centers,
        #'cell_areas': cell_areas,
        #'pressure': pressure_cells
    }

    # Calculate center of pressure coordinates
    if abs(total_force[1]) > 1e-10:  # F_y component
        cp_x = reference_point[0] + total_moment[2] / total_force[1]
    else:
        cp_x = reference_point[0]

    if abs(total_force[0]) > 1e-10:  # F_x component
        cp_y = reference_point[1] - total_moment[2] / total_force[0]
    else:
        cp_y = reference_point[1]

    results['center_of_pressure'] = np.array([cp_x, cp_y, reference_point[2]])

    return results
```

In [40]:

```
def parse_case_name(case_name):
    """
    Parse case name to extract AoA and Mach number
    Example: AoA_0p0_Ma_0p8 -> AoA=0.0, Ma=0.8
    """
    # Extract AoA
    aoa_match = re.search(r'AoA_([0-9]+p[0-9]+)', case_name)
    if aoa_match:
        aoa_str = aoa_match.group(1).replace('p', '.')
        aoa = float(aoa_str)
    else:
        aoa = None

    # Extract Mach number
    ma_match = re.search(r'Ma_([0-9]+p[0-9]+)', case_name)
    if ma_match:
        ma_str = ma_match.group(1).replace('p', '.')
        ma = float(ma_str)
    else:
        ma = None

    return aoa, ma
```

In [41]:

```
def find_latest_time_directory(base_path):
    """
    Find the directory with the highest numerical name (latest time)
    """
    time_dirs = []
    for item in os.listdir(base_path):
        item_path = os.path.join(base_path, item)
```

```
        if os.path.isdir(item_path):
            try:
                time_val = float(item)
                time_dirs.append((time_val, item))
            except ValueError:
                continue

    if time_dirs:
        # Return the directory name with the highest time value
        latest_time = max(time_dirs, key=lambda x: x[0])
        return latest_time[1], latest_time[0]
    else:
        return None, None
```

In [42]:
```
def load_mesh_from_case(case_path):
    """
    Load the rocket mesh from the case directory
    """
    vtk_path = os.path.join(case_path, "postProcessing", "rocketVTK")

    if not os.path.exists(vtk_path):
        print(f"VTK path not found: {vtk_path}")
        return None, None

    # Find latest time directory
    time_dir, time_value = find_latest_time_directory(vtk_path)
    if time_dir is None:
        print(f"No time directories found in: {vtk_path}")
        return None, None

    # Load the mesh
    mesh_file = os.path.join(vtk_path, time_dir, "rocketWall.vtp")
    if not os.path.exists(mesh_file):
        print(f"Mesh file not found: {mesh_file}")
        return None, None

    try:
        mesh = pv.read(mesh_file)
        return mesh, time_value
    except Exception as e:
        print(f"Error loading mesh from {mesh_file}: {e}")
        return None, None
```

In [50]:
```
def compute_center_of_pressure_simple(mesh, pressure_field='p', reference_point=None):
    """
    Simplified center of pressure calculation from mesh
    """
    if pressure_field not in mesh.array_names:
        print(f"Pressure field '{pressure_field}' not found. Available: {mesh.array_names}")
        return None

    # Get pressure data
    pressure = mesh[pressure_field]

    # Compute cell centers and areas
    cell_centers = mesh.cell_centers()
    cell_areas = mesh.compute_cell_sizes()['Area']

    # Compute surface normals
    mesh_with_normals = mesh.compute_normals(cell_normals=True, point_normals=False)
    normals = mesh_with_normals['Normals']

    # Handle pressure data conversion if needed
    if len(pressure) == mesh.n_points:
        # Convert point data to cell data
        mesh_temp = mesh.copy()
        mesh_temp[pressure_field] = pressure
        cell_mesh = mesh_temp.cell_data_to_point_data().point_data_to_cell_data()
        pressure_cells = cell_mesh[pressure_field]
    elif len(pressure) == mesh.n_cells:
        pressure_cells = pressure
    else:
        print(f"Pressure data size mismatch: {len(pressure)} vs cells:{mesh.n_cells} points:{mesh.n_points}")
        return None

    # Set reference point
    if reference_point is None:
        reference_point = np.array([mesh.bounds[0], 0, 0])  # Nose of rocket

    # Compute pressure forces
    pressure_forces = pressure_cells[:, np.newaxis] * cell_areas[:, np.newaxis] * (normals)

    # Total force
    total_force = np.sum(pressure_forces, axis=0)

    # Compute moments about reference point
    centers = cell_centers.points
```

```
    moment_arms = centers - reference_point
    moments = np.cross(moment_arms, pressure_forces)
    total_moment = np.sum(moments, axis=0)

    # Calculate center of pressure using component method
    cp = np.zeros(3)

    # X-component (longitudinal position)
    if abs(total_force[1]) > 1e-10:  # F_y component
        cp[0] = reference_point[0] + total_moment[2] / total_force[1]
    else:
        cp[0] = reference_point[0]

    # Y-component (vertical position)
    if abs(total_force[0]) > 1e-10:  # F_x component
        cp[1] = reference_point[1] - total_moment[2] / total_force[0]
    else:
        cp[1] = reference_point[1]

    # Z-component (lateral position)
    if abs(total_force[0]) > 1e-10:  # F_x component
        cp[2] = reference_point[2] + total_moment[1] / total_force[0]
    else:
        cp[2] = reference_point[2]

    return {
        'center_of_pressure': cp,
        'total_force': total_force,
        'total_moment': total_moment,
        'reference_point': reference_point
    }
```

In [51]:

```
def load_force_coefficients(case_path):
    """
    Load force coefficients from coefficient.dat file using the provided load_dat function

    Parameters:
    -----------
    case_path : str
        Path to the case directory
    load_dat : function
        Your existing function to load .dat files
    """
    coeffs_path = os.path.join(case_path, "postProcessing", "forceCoeffs",)

    if not os.path.exists(coeffs_path):
        print(f"Force coefficients path not found: {coeffs_path}")
        return None

    # Find latest time directory
    time_dir, _ = find_latest_time_directory(coeffs_path)
    if time_dir is None:
        print(f"No time directories found in: {coeffs_path}")
        return None

    coeffs_file = os.path.join(coeffs_path, f"{time_dir}", "coefficient.dat")
    if not os.path.exists(coeffs_file):
        print(f"Coefficient file not found: {coeffs_file}")
        return None

    try:
        # Use your load_dat function
        df = load_dat(coeffs_file)

        if df is None or df.empty:
            print(f"load_dat returned empty DataFrame for {coeffs_file}")
            return None

        # Get the final row (converged values)
        final_row = df.iloc[-1]

        # Extract coefficients - adjust column names as needed
        # Common names: Cd, Cl, CmPitch, Cs, CmRoll, CmYaw
        coeffs = {}

        # Try different possible column names
        cd_cols = ['Cd', 'cd', 'CD', 'drag']
        cl_cols = ['Cl', 'cl', 'CL', 'lift']
        cm_cols = ['CmPitch', 'Cm', 'cm', 'CMpitch', 'moment']

        for col in cd_cols:
            if col in df.columns:
                coeffs['Cd'] = final_row[col]
                break
        else:
            coeffs['Cd'] = None
```

```
        for col in cl_cols:
            if col in df.columns:
                coeffs['Cl'] = final_row[col]
                break
        else:
            coeffs['Cl'] = None

        for col in cm_cols:
            if col in df.columns:
                coeffs['CmPitch'] = final_row[col]
                break
        else:
            coeffs['CmPitch'] = None

        # Also return the time and all available columns for debugging
        coeffs['time'] = final_row.get('Time', final_row.iloc[0])
        coeffs['available_columns'] = df.columns.tolist()

        return coeffs

    except Exception as e:
        print(f"Error loading coefficients from {coeffs_file}: {e}")
        return None
```

In [52]:

```
def analyze_all_cases(base_directory):
    """
    Analyze all cases in the base directory
    """
    results = []

    # Find all case directories
    case_pattern = os.path.join(base_directory, "AoA_*_Ma_*")
    case_dirs = glob.glob(case_pattern)

    if not case_dirs:
        print(f"No case directories found matching pattern: {case_pattern}")
        return pd.DataFrame()

    print(f"Found {len(case_dirs)} cases to analyze...")

    for case_dir in sorted(case_dirs):
        case_name = os.path.basename(case_dir)
        print(f"\nAnalyzing case: {case_name}")

        # Parse AoA and Mach number
        aoa, ma = parse_case_name(case_name)

        # Load mesh and get iteration number
        mesh, iterations = load_mesh_from_case(case_dir)
        if mesh is None:
            print(f"Skipping {case_name} - mesh loading failed")
            continue

        # Compute center of pressure
        cop_results = compute_center_of_pressure_simple(mesh)
        if cop_results is None:
            print(f"Skipping {case_name} - CoP calculation failed")
            continue

        # Load force coefficients
        coeffs = load_force_coefficients(case_dir)
        if coeffs is None:
            print(f"Warning: Could not load coefficients for {case_name}")
            coeffs = {'Cd': None, 'Cl': None, 'CmPitch': None}

        # Compile results
        result = {
            'AoA': aoa,
            'Ma': ma,
            'iterations': iterations,
            'CopX': cop_results['center_of_pressure'][0],
            'CopY': cop_results['center_of_pressure'][1],
            'CopZ': cop_results['center_of_pressure'][2],
            'Force_X': cop_results['total_force'][0],
            'Force_Y': cop_results['total_force'][1],
            'Force_Z': cop_results['total_force'][2],
            'Moment_X': cop_results['total_moment'][0],
            'Moment_Y': cop_results['total_moment'][1],
            'Moment_Z': cop_results['total_moment'][2],
            'Cd': coeffs['Cd'],
            'Cl': coeffs['Cl'],
            'CmPitch': coeffs['CmPitch'],
            'case_name': case_name
        }

        results.append(result)
```

```
        # Print summary for this case
        print(f"  AoA: {aoa}, Ma: {ma}, Iterations: {iterations}")
        print(f"  CoP: [{result['CopX']:.6f}, {result['CopY']:.6f}, {result['CopZ']:.6f}]")
        print(f"  Coeffs: Cd={coeffs['Cd']}, Cl={coeffs['Cl']}, Cm={coeffs['CmPitch']}")

    # Convert to DataFrame
    df = pd.DataFrame(results)

    # Sort by AoA and Mach number
    if not df.empty:
        df = df.sort_values(['AoA', 'Ma']).reset_index(drop=True)

    return df
```

Testing the CoP calc functions

In [53]:
```
mesh = pv.read("Data/CollectedResults/AoA_0p0_Ma_1p2/postProcessing/rocketVTK/6200/rocketWall.vtp")
mesh
```

Out[53]:

Header

PolyData	Information
N Cells	172662
N Points	177262
N Strips	0
X Bounds	4.930e-11, 2.782e+00
Y Bounds	-8.762e-02, 1.718e-01
Z Bounds	-1.498e-01, 1.498e-01
N Arrays	6

Data Arrays

Name	Field	Type	N Comp	Min	Max
Ma	Points	float32	1	6.676e-03	1.017e+00
T	Points	float32	1	2.429e+02	3.403e+02
p	Points	float32	1	8.583e+03	1.435e+05
rho	Points	float32	1	1.025e-01	1.533e+00
U	Points	float32	3	-2.067e+02	3.078e+02
TimeValue	Fields	float32	1	6.200e+03	6.200e+03

In [54]:
```
results = compute_center_of_pressure(mesh)
results
```

```
Pressure data shape: (177262,)
Number of cells: 172662
Number of points: 177262
Converting point data to cell data...
```

Out[54]:
```
{'total_force': pyvista_ndarray([-439.18264859,    4.28798407,    1.0886823 ]),
 'total_moment': array([-0.28730781, -2.92672062, 10.42615181]),
 'reference_point': array([4.92959319e-11, 0.00000000e+00, 0.00000000e+00]),
 'center_of_pressure': array([2.43148101, 0.0237399 , 0.        ])}
```

In [55]:
```
coeffs = load_force_coefficients("Data/CollectedResults/AoA_3p0_Ma_1p6")
coeffs
```

Out[55]:
```
{'Cd': np.float64(0.37405347747),
 'Cl': np.float64(0.28636442028),
 'CmPitch': np.float64(-0.20749544517),
 'time': np.float64(5054.0),
 'available_columns': ['Time',
  'Cd',
  'Cd(f)',
  'Cd(r)',
  'Cl',
  'Cl(f)',
  'Cl(r)',
  'CmPitch',
  'CmRoll',
  'CmYaw',
  'Cs',
  'Cs(f)',
  'Cs(r)']}
```

In [ ]:
```
the_df = analyze_all_cases("Data/CollectedResultsMesh_New")
the_dfSpecial = analyze_all_cases("Data/CollectedResultsSpecials")
```

In [127…
```
the_df.head()
```

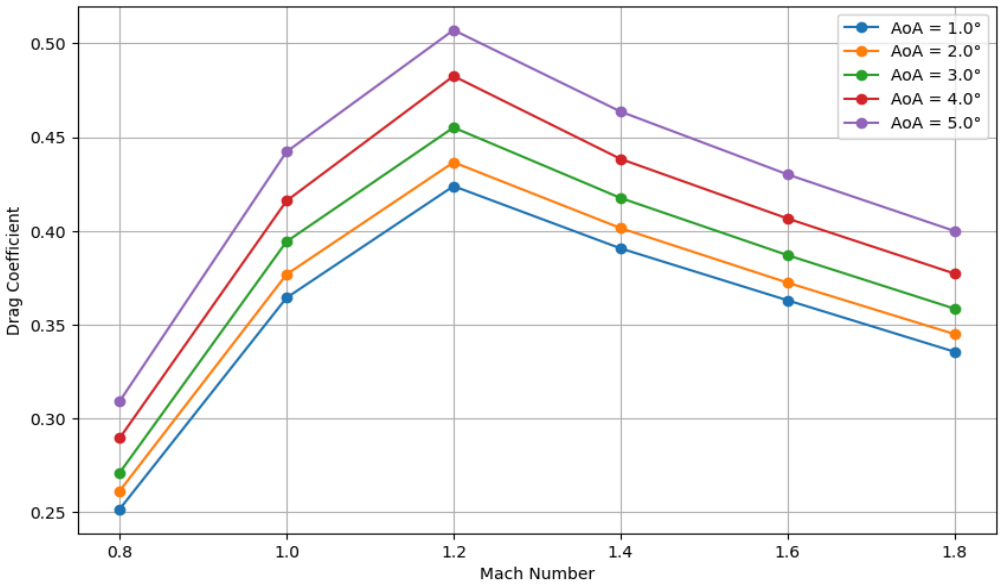
Out[127…

	df	df_components	edges	dx	Fy_bin	Fy_point	Fy_total	
0	x_mid_m Fx_N Fy_N Fz_...	name m...	[0.0003049805762232963 4, 0.026800267771071355,...	0.026495	[-6.362681913392432, -12.00105000655714, -15.4...	[0.0, 0.0, 0.0, 0.0, 0.0, -93.01035899782347, ...	[-6.362681913392432, -12.00105000655714, -15.4...	[-2 -3
1	x_mid_m Fx_N Fy_N Fz_N...	name m...	[0.0003049805858866758, 0.026800267780734736, ...	0.026495	[-1.1881993595594427, -2.2101281237661974, -3....	[0.0, 0.0, 0.0, 0.0, 0.0, -36.084138841465276,...	[-1.1881993595594427, -2.2101281237661974, -3....	[-18 -29
2	x_mid_m Fx_N Fy_N Fz_...	name m...	[0.0003049806051917212, 0.02680026780003978, 0...	0.026495	[-1.733972074090311, -3.0037449994632217, -3.9...	[0.0, 0.0, 0.0, 0.0, 0.0, -66.08088894120708, ...	[-1.733972074090311, -3.0037449994632217, -3.9...	[-17 -27
3	x_mid_m Fx_N Fy_N Fz_N...	name m...	[0.0003049806051917212, 0.02680026780003978, 0...	0.026495	[-0.4453728201587215, -0.748796009043716, -1.0...	[0.0, 0.0, 0.0, 0.0, 0.0, -12.994177252160153,...	[-0.4453728201587215, -0.748796009043716, -1.0...	[-15 -2
4	x_mid_m Fx_N Fy_N Fz_...	name m...	[0.0003049805762232963 4, 0.026800267771071355,...	0.026495	[-1.8665751818486127, -3.4049962928404915, -4....	[0.0, 0.0, 0.0, 0.0, 0.0, -54.94356977181285, ...	[-1.8665751818486127, -3.4049962928404915, -4....	[-18 -29

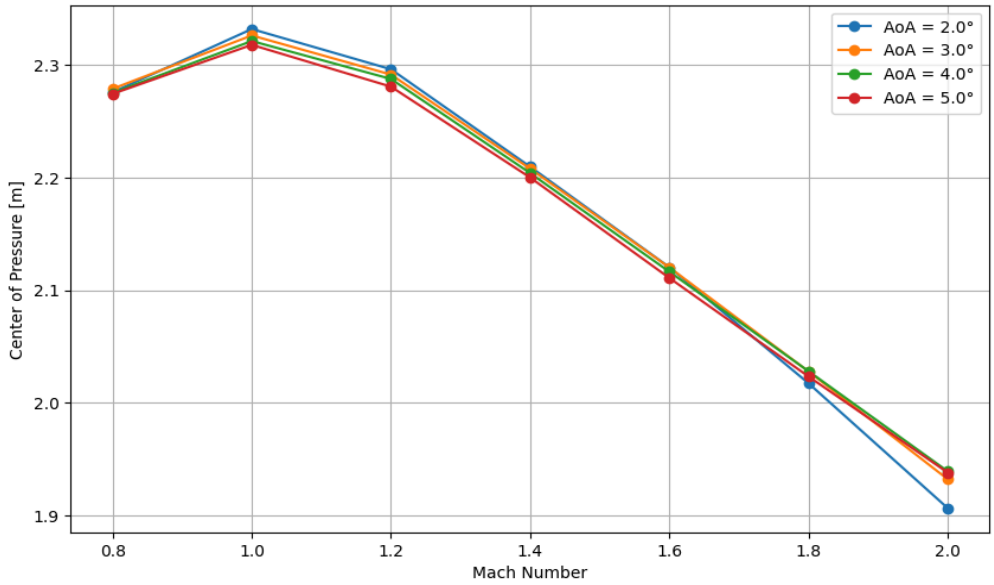
5 rows × 26 columns

Ploting the Coefficients

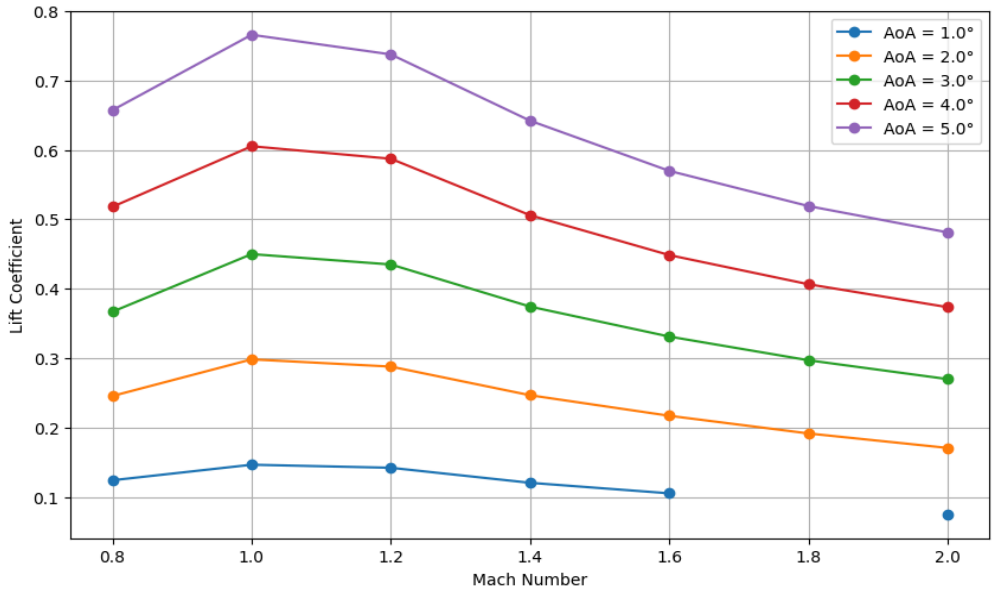
In [99]:
```
y_column = 'Cd'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['AoA'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['AoA'] == aoa].sort_values('Ma')
    plt.plot(data['Ma'][:-1], data[y_column][:-1], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel('Drag Coefficient')
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)

plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



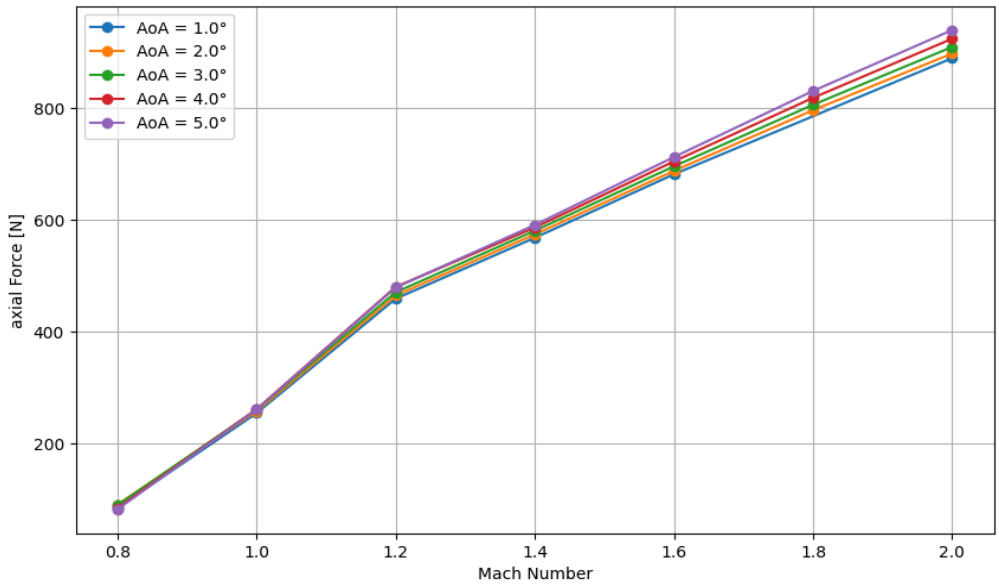
In [100…
```
y_column = 'CopX'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['AoA'].unique())[1:]

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['AoA'] == aoa].sort_values('Ma')
    plt.plot(data['Ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("Center of Pressure [m]")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)
plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



In [102…
```
y_column = 'Cl'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['AoA'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['AoA'] == aoa].sort_values('Ma')
    plt.plot(data['Ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("Lift Coefficient")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)


plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```

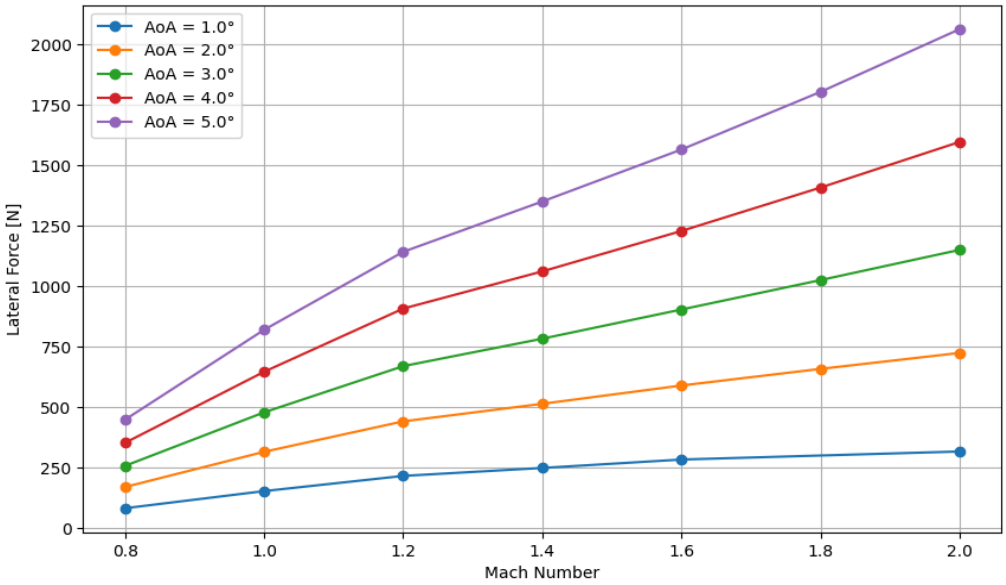


In [61]:
```
y_column = 'Force_X'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['AoA'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['AoA'] == aoa].sort_values('Ma')
    plt.plot(data['Ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("axial Force [N]")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)

plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



In [62]:
```
y_column = 'Force_Y'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))
```

```
# Get unique AoA values
aoa_values = sorted(the_df['AoA'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['AoA'] == aoa].sort_values('Ma')
    plt.plot(data['Ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("Lateral Force [N]")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)

plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



Residuals

In [128…

```
base_dir = "Data/CollectedResults"

cases = os.listdir(base_dir)

solverInfo = []
for case in cases:
    match = re.match(r"AoA_(\d+p\d+)_Ma_(\d+p\d+)", case)
    if match:
        solver = load_dat(os.path.join(base_dir, case, "postProcessing", "solverInfo", "0", "solverInfo.dat"))
        solver['case'] = case
        solverInfo.append(solver)


solver_df = pd.concat(solverInfo, ignore_index=True)
solver_df.head()
```

Out[128…

	Time	U_solver	Ux_initial	Ux_final	Ux_iters	Uy_initial	Uy_final	Uy_iters	Uz_initial	Uz_final	…	omega_initial	omega_fina
0	1	DILUPBiCGStab	0.999797	1.079700e-10	5	1.000000	3.168901e-10	5	1.000000	3.973690e-10	…	6.416371e-06	4.730041e-1
1	2	DILUPBiCGStab	0.001155	6.651379e-15	5	0.114956	8.888956e-13	5	0.210009	1.192484e-11	…	7.660933e-07	8.682887e-1
2	3	DILUPBiCGStab	0.000215	1.556813e-14	5	0.053715	3.685329e-13	5	0.114670	7.893272e-11	…	8.086241e-07	7.336947e-1
3	4	DILUPBiCGStab	0.000177	5.262708e-15	5	0.028945	2.053355e-13	5	0.090735	1.052943e-12	…	7.734756e-07	7.971181e-1
4	5	DILUPBiCGStab	0.000160	2.504231e-15	5	0.023868	1.492081e-13	5	0.094075	6.857327e-13	…	6.576966e-07	7.229833e-1

5 rows × 28 columns

In [66]:

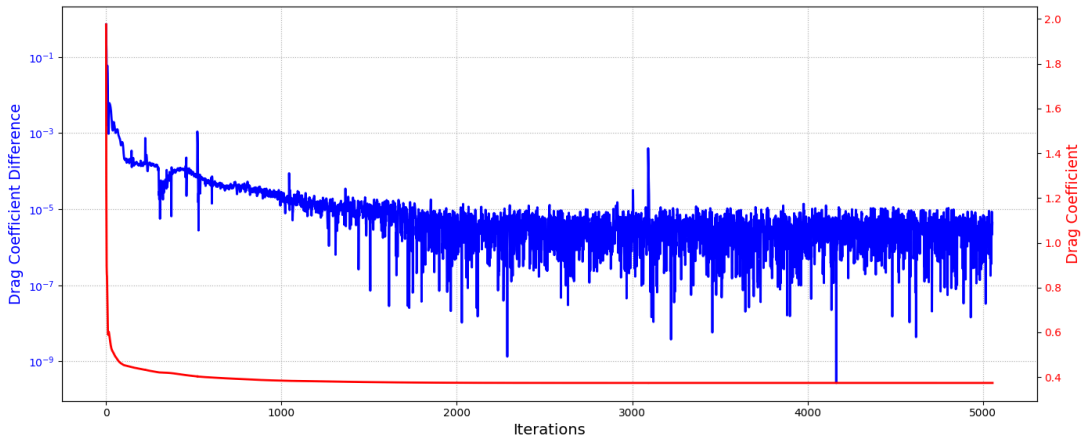
```
plt.figure(figsize=(14, 6))

components = ['Ux', 'Uy', 'Uz', 'k', 'omega', 'p']
colors = ['blue', 'orange', 'green', 'red', 'purple', 'brown']


dict_labels = {
"Ux": "$U_x$",
"Uy": "$U_y$",
"Uz": "$U_z$",
"k": "$k$",
"omega": "$\\omega$",
"p": "$p$"
}

for i, component in enumerate(components):
    grouped = solver_df.groupby('Time')[f'{component}_initial']
    time = grouped.mean().index[:6000]
    mean = grouped.mean().values[:6000]
    min_vals = grouped.min().values[:6000]
    max_vals = grouped.max().values[:6000]

    # Fill between min and max
    plt.fill_between(
        time, min_vals, max_vals,
        color=colors[i], alpha=0.1
    )

    # Plot mean as solid line
    plt.plot(
        time, mean,
        label=f'{dict_labels[component]}',
        color=colors[i], linewidth=2
    )

# Log scale
# Log scale on y-axis
plt.yscale('log')

# Set major and minor ticks explicitly
plt.gca().yaxis.set_major_locator(LogLocator(base=10.0, subs=(1.0,), numticks=10))
plt.gca().yaxis.set_minor_locator(LogLocator(base=10.0, subs=np.arange(2, 10)*0.1, numticks=100))
plt.gca().yaxis.set_minor_formatter(NullFormatter())  # hide minor tick labels

# Grid: dotted for both
plt.grid(True, which='both', linestyle=':', linewidth=0.8)


plt.xlabel('Iterations', fontsize=14)
plt.ylabel('Residuals', fontsize=14)
plt.legend(fontsize=12, loc='upper right', frameon=True, shadow=True)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()


plt.savefig("./results/solver_residuals.pdf", bbox_inches='tight')
```

In [130…
```
base_dir = "Data/CollectedResults"

cases = os.listdir(base_dir)

coeff = []
for case in cases:
    match = re.match(r"AoA_(\d+p\d+)_Ma_(\d+p\d+)", case)
    if match:
        data = load_dat(os.path.join(base_dir, case, "postProcessing", "forceCoeffs", "0", "coefficient.dat"))
        data['case'] = case
        coeff.append(data)


coeff_df = pd.concat(coeff, ignore_index=True)
coeff_df.head()
```

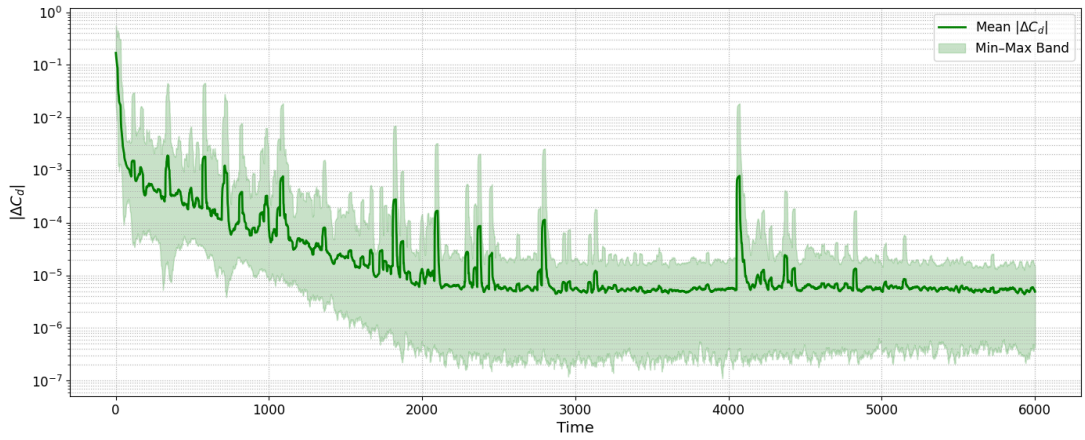
Out[130…

	Time	Cd	Cd(f)	Cd(r)	Cl	Cl(f)	Cl(r)	CmPitch	CmRoll	CmYaw	Cs	Cs(f)	Cs(r)	
0	1	2.341862	1.170740	1.171122	-0.032063	0.012420	-0.044483	0.028452	-0.000191	0.005347	0.006613	0.008654	-0.002040	AoA_(
1	2	2.070011	1.034793	1.035218	0.018077	-0.002084	0.020161	-0.011123	-0.000213	0.006016	0.007241	0.009636	-0.002395	AoA_(
2	3	1.747696	0.873757	0.873938	0.040892	-0.008501	0.049392	-0.028947	-0.000091	0.006253	0.007754	0.010129	-0.002376	AoA_(
3	4	1.392240	0.696126	0.696114	0.040001	-0.007066	0.047067	-0.027066	0.000006	0.005540	0.007344	0.009212	-0.001869	AoA_(
4	5	1.140421	0.570282	0.570139	0.020338	0.000228	0.020110	-0.009941	0.000072	0.003465	0.005179	0.006055	-0.000876	AoA_(

In [68]:
```
fig, ax1 = plt.subplots(figsize=(14, 6))

# Primary Y-axis: |ΔCd|
color1 = 'blue'
cd_diff = np.abs(np.diff(data['Cd']))
ax1.plot(data['Time'][:-1], cd_diff, color=color1, label='|ΔCd|', linewidth=2)
ax1.set_xlabel('Iterations', fontsize=14)
ax1.set_ylabel('Drag Coefficient Difference', color=color1, fontsize=14)
ax1.set_yscale('log')
ax1.tick_params(axis='y', labelcolor=color1)
ax1.grid(True, which='both', linestyle=':', linewidth=0.8)

# Secondary Y-axis: Cd
ax2 = ax1.twinx()
color2 = 'red'
ax2.plot(data['Time'], data['Cd'], color=color2, label='Cd', linewidth=2)
ax2.set_ylabel('Drag Coefficient', color=color2, fontsize=14)
ax2.tick_params(axis='y', labelcolor=color2)

# Title and layout
#plt.title('Cd and |ΔCd| vs Time', fontsize=16, fontweight='bold')
fig.tight_layout()

plt.savefig("./results/Cd_and_deltaCd_vs_Time.pdf", bbox_inches='tight')
```



In [69]:
```
cd_df_sorted = coeff_df.sort_values(by=['case', 'Time'])
cd_df_sorted['Cd_diff'] = cd_df_sorted.groupby('case')['Cd'].diff().abs()

# Step 2: Group by Time across all cases
grouped = cd_df_sorted.groupby('Time')['Cd_diff']
time = grouped.mean().index[:6000]
cd_diff_mean = grouped.mean()
cd_diff_min = grouped.min()
cd_diff_max = grouped.max()

# Step 3: Apply smoothing
window = 20  # number of points in the moving window
mean_smooth = cd_diff_mean.rolling(window, center=True, min_periods=1).mean()[:6000]
min_smooth = cd_diff_min.rolling(window, center=True, min_periods=1).mean()[:6000]
max_smooth = cd_diff_max.rolling(window, center=True, min_periods=1).mean()[:6000]

# Step 4: Plot
plt.figure(figsize=(14, 6))
plt.plot(time, mean_smooth, color='green', linewidth=2, label=r'Mean $|\Delta C_d|$')
plt.fill_between(time, min_smooth, max_smooth, color='green', alpha=0.2, label='Min–Max Band')

plt.yscale('log')
plt.xlabel('Time', fontsize=14)
plt.ylabel(r'$|\Delta C_d|$', fontsize=14)
plt.grid(True, which='both', linestyle=':', linewidth=0.8)
plt.legend(fontsize=12, loc='upper right')
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.tight_layout()
plt.savefig("./results/mean_Cd_diff_vs_Time.pdf", bbox_inches='tight')
```



In [ ]:
```
base_dir = "Data/CollectedResults"

cases = os.listdir(base_dir)

yp = []
for case in cases:
    match = re.match(r"AoA_(\d+p\d+)_Ma_(\d+p\d+)", case)
    if match:
```

```
        data = load_dat(os.path.join(base_dir, case, "postProcessing", "yPlus1", "0", "yPlus.dat"))
        data['case'] = case
        yp.append(data)


yp_df = pd.concat(yp, ignore_index=True)
yp_mean = yp_df.groupby('Time')['max'].mean().reset_index()
yp_min = yp_df.groupby('Time')['max'].min().reset_index()
yp_max = yp_df.groupby('Time')['max'].max().reset_index()

plt.figure(figsize=(14, 6))
plt.plot(yp_mean['Time'], yp_mean['max'], label='Mean y+', color='blue', linewidth=2)
plt.fill_between(yp_mean['Time'], yp_min['max'], yp_max['max'], color='blue', alpha=0.2, label='Min–Max Band')

plt.xlabel('Time', fontsize=14)
plt.ylabel('y+', fontsize=14)
```

Validation

In [131…

```
ras_df = pd.read_csv("Data/CDRocekt.csv")
ras_df.head()
```

Out[131…

	Mach	Alpha	CD	CD Power-Off	CD Power-On	CA Power-Off	CA Power-On	CL	CN	CN Potential	CN Viscous	CNalpha (0 to 4 deg) (per rad)	CP	CP (0 to 4 deg)	Reynolds Number
0	0.01	0	0.437251	0.437251	0.431133	0.437251	0.431133	0.0	0.0	0.0	0.0	8.503829	89.637836	85.50716	646814.4
1	0.02	0	0.391705	0.391705	0.385587	0.391705	0.385587	0.0	0.0	0.0	0.0	8.503829	89.637836	85.50716	1291031.0
2	0.03	0	0.368988	0.368988	0.362870	0.368988	0.362870	0.0	0.0	0.0	0.0	8.503829	89.637836	85.50716	1932664.0
3	0.04	0	0.354372	0.354372	0.348254	0.354372	0.348254	0.0	0.0	0.0	0.0	8.503829	89.637836	85.50716	2571728.0
4	0.05	0	0.343816	0.343816	0.337698	0.343816	0.337698	0.0	0.0	0.0	0.0	8.503829	89.637836	85.50716	3208236.0

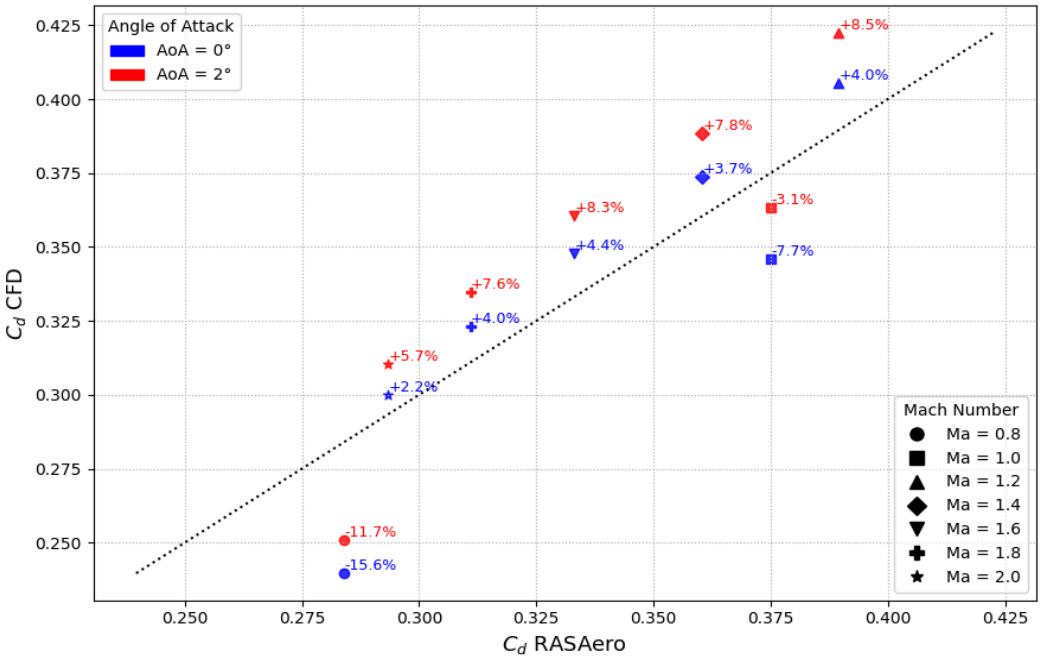
In [ ]:

```
the_df = analyze_all_cases("Data/CollectedResults")
the_df1 = the_df[the_df['Ma'] != 0.8].copy()
```

In [85]:

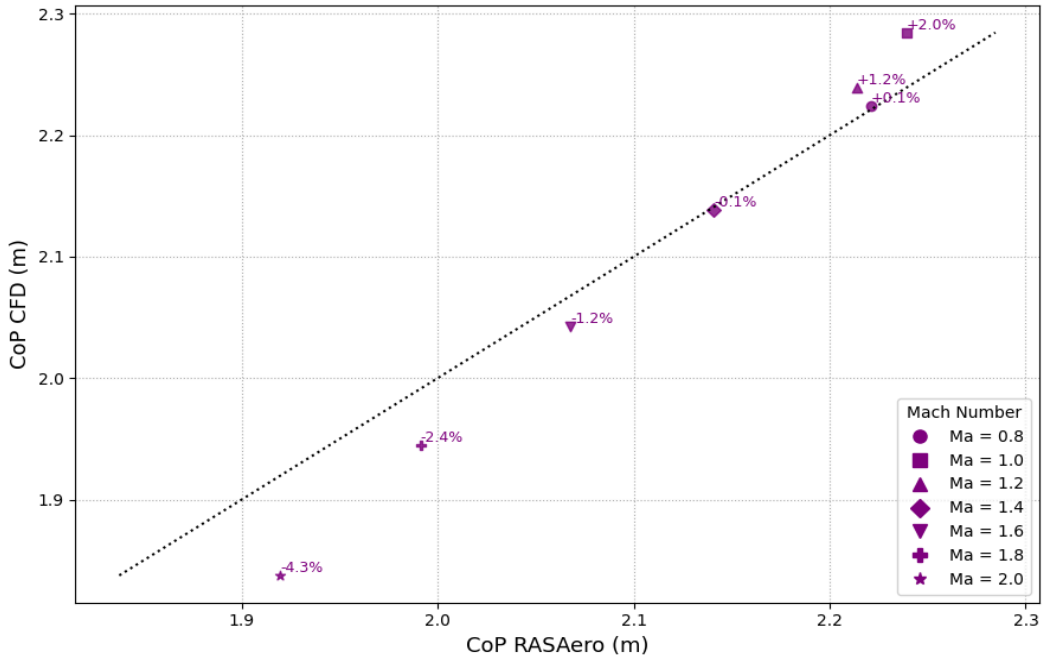
```
cd_ras = []
cd_cfd = []
aoa_list = []
mach_list = []

for row in the_df.itertuples():
    aoa = row.AoA
    ma = row.Ma

    match = ras_df[(ras_df['Alpha'] == aoa) & (ras_df['Mach'] == ma)]

    if not match.empty:
        cd_ras.append(match['CD'].values[0])
        cd_cfd.append(row.Cd)
        aoa_list.append(aoa)
        mach_list.append(ma)

# Build DataFrame
compare_df = pd.DataFrame({
    'Cd_ras': cd_ras,
    'Cd_cfd': cd_cfd,
    'AoA': aoa_list,
    'Ma': mach_list
})

# Compute % error
compare_df['Cd_pct_error'] = 100 * (compare_df['Cd_cfd'] - compare_df['Cd_ras']) / compare_df['Cd_ras']

# === Your plotting code here ===
# Same as before for plotting the points
# We'll just improve the legend now:

plt.figure(figsize=(9, 6))

# Color map for AoA
colors = {0: 'blue', 2: 'red'}

# Marker map for Mach number
unique_mach = sorted(compare_df['Ma'].unique())
markers = ['o', 's', '^', 'D', 'v', 'P', '*']
mach_marker_map = {ma: markers[i % len(markers)] for i, ma in enumerate(unique_mach)}
```

```
# Plot points
for aoa in [0, 2]:
    subset = compare_df[compare_df['AoA'] == aoa]
    for ma in subset['Ma'].unique():
        subsub = subset[subset['Ma'] == ma]
        plt.scatter(
            subsub['Cd_ras'], subsub['Cd_cfd'],
            color=colors[aoa],
            marker=mach_marker_map[ma],
            alpha=0.8
        )
        for _, row in subsub.iterrows():
            label = f"{row['Cd_pct_error']:+.1f}%"
            plt.text(row['Cd_ras'], row['Cd_cfd'], label, fontsize=9, color=colors[aoa], ha='left', va='bottom')

# Diagonal reference line
min_val = min(compare_df['Cd_ras'].min(), compare_df['Cd_cfd'].min())
max_val = max(compare_df['Cd_ras'].max(), compare_df['Cd_cfd'].max())
plt.plot([min_val, max_val], [min_val, max_val], linestyle=':', color='black')

plt.xlabel('$C_d$ RASAero', fontsize=13)
plt.ylabel('$C_d$ CFD', fontsize=13)
#plt.title('$Cd$ Comparison: CFD vs RASAero', fontsize=14, fontweight='bold')
plt.grid(True, linestyle=':', linewidth=0.8)

# === Clean custom legends ===

# Color legend for AoA
color_handles = [
    mpatches.Patch(color='blue', label='AoA = 0°'),
    mpatches.Patch(color='red', label='AoA = 2°')
]

# Marker legend for Mach numbers
marker_handles = [
    mlines.Line2D([], [], color='black', marker=mach_marker_map[ma], linestyle='None', markersize=8, label=f'Ma =
    for ma in unique_mach
]

# Combine and place
legend1 = plt.legend(handles=color_handles, title='Angle of Attack', loc='upper left', frameon=True)
legend2 = plt.legend(handles=marker_handles, title='Mach Number', loc='lower right', frameon=True)
plt.gca().add_artist(legend1)  # keep both legends

plt.tight_layout()

plt.savefig("./results/Cd_comparison_CFD_vs_RASAero.pdf", bbox_inches='tight')
```



In [86]:
```
cp_ras = []
cp_cfd = []
mach_list = []
```

```
aoa = 2.0  # Only AoA 2.0
for row in the_df[the_df['AoA'] == aoa].itertuples():
    ma = row.Ma
    match = ras_df[(ras_df['Alpha'] == aoa) & (ras_df['Mach'] == ma)]
    if not match.empty:
        cp_ras.append(match['CP'].values[0] * 0.0254)  # Convert inches to meters
        cp_cfd.append(row.CopX)
        mach_list.append(ma)

# Build dataframe
compare_cp = pd.DataFrame({
    'Cp_ras': cp_ras,
    'Cp_cfd': cp_cfd,
    'Ma': mach_list
})
compare_cp['Cp_pct_error'] = 100 * (compare_cp['Cp_cfd'] - compare_cp['Cp_ras']) / compare_cp['Cp_ras']

# Plot
plt.figure(figsize=(9, 6))
markers = ['o', 's', '^', 'D', 'v', 'P', '*']
unique_mach = sorted(compare_cp['Ma'].unique())
mach_marker_map = {ma: markers[i % len(markers)] for i, ma in enumerate(unique_mach)}

# Plot scatter points and annotations
for ma in unique_mach:
    sub = compare_cp[compare_cp['Ma'] == ma]
    plt.scatter(
        sub['Cp_ras'], sub['Cp_cfd'],
        color='purple',
        marker=mach_marker_map[ma],
        label=f'Ma = {ma}',
        alpha=0.8
    )
    for _, row in sub.iterrows():
        label = f"{row['Cp_pct_error']:+.1f}%"
        plt.text(row['Cp_ras'], row['Cp_cfd'], label, fontsize=9, color='purple', ha='left', va='bottom')

# Reference line
min_val = min(compare_cp['Cp_ras'].min(), compare_cp['Cp_cfd'].min())
max_val = max(compare_cp['Cp_ras'].max(), compare_cp['Cp_cfd'].max())
plt.plot([min_val, max_val], [min_val, max_val], linestyle=':', color='black')

plt.xlabel('CoP RASAero (m)', fontsize=13)
plt.ylabel('CoP CFD (m)', fontsize=13)
#plt.title('Center of Pressure Comparison (AoA = 2°)', fontsize=14, fontweight='bold')
plt.grid(True, linestyle=':', linewidth=0.8)

# Legend: Mach only
marker_handles = [
    mlines.Line2D([], [], color='purple', marker=mach_marker_map[ma], linestyle='None', markersize=8, label=f'Ma
    for ma in unique_mach
]
plt.legend(handles=marker_handles, title='Mach Number', loc='lower right', frameon=True)

plt.tight_layout()

plt.savefig("./results/Cp_comparison_CFD_vs_RASAero.pdf", bbox_inches='tight')
```



Force Analysis

In [103…

```
COMPONENTS = [
    {"name": "Nose tip assembly",                    "mass_g": 353.99,      "cg_mm": 146.06},
    {"name": "Nosecone",                             "mass_g": 1159.0,      "cg_mm": 598.67},
    {"name": "Main Parachute",                       "mass_g": 691.0,       "cg_mm": 605.0},
    {"name": "Main Shock chords",                    "mass_g": 720.0,       "cg_mm": 677.0},
    {"name": "Large-diam. Fuselage",                 "mass_g": 459.82,      "cg_mm": 909.54},
    {"name": "Large-diam. thrust ring",              "mass_g": 207.10,      "cg_mm": 967.99},
    {"name": "Service Module",                       "mass_g": 3079.0,      "cg_mm": 923.0},
    {"name": "G10 plate",                            "mass_g": 95.47,       "cg_mm": 1013.841},
    {"name": "Ribs (6×)",                            "mass_g": 6*28.53,     "cg_mm": 1097.39},
    {"name": "Rear LD thrust ring",                  "mass_g": 97.48,       "cg_mm": 1038.67},
    {"name": "Transition skin",                      "mass_g": 200.03,      "cg_mm": 1146.69},
    {"name": "Small-diam. fuselage",                 "mass_g": 514.393,     "cg_mm": 1213.54},
    {"name": "SD fuselage thrust ring",              "mass_g": 105.72,      "cg_mm": 1173.23},
    {"name": "Drogue chute",                         "mass_g": 569.0,       "cg_mm": 1291.0},
    {"name": "Drogue chords",                        "mass_g": 720.0,       "cg_mm": 1336.0},
    {"name": "Small-diam. coupler",                  "mass_g": 425.07,      "cg_mm": 1371.35},
    {"name": "Small-diam. coupler bulkhead assembly", "mass_g": 283.89,     "cg_mm": 1444.68},
    {"name": "Motor (full)",                         "mass_g": 16525.0,     "cg_mm": 2160.0},
    {"name": "Motorcase",                            "mass_g": 740.11,      "cg_mm": 2087.74},
    {"name": "Fins (3×)",                            "mass_g": 3*235.11,    "cg_mm": 2695.61},
]

df_components1 = pd.DataFrame(COMPONENTS)

df_components1["mass_kg"] = df_components1["mass_g"] / 1000.0
df_components1["cg_m"] = df_components1["cg_mm"] / 1000.0
```

In [104…

```
def distribute_component(
    df: pd.DataFrame,
    component_name: str,
    half_length: float,
    n_div: int,
    *,
    mass_col: str = "mass_kg",
    cg_col: str   = "cg_m",
    name_col: str = "name",
) -> pd.DataFrame:
    """
    Replace one component row with `n_div` equal-mass slices spread over
    [cg - L, cg + L].

    Parameters
    ----------
    df : DataFrame
        Must contain at least columns `name_col`, `mass_col`, `cg_col`.
    component_name : str
```

```
        Exact string in `df[name_col]` to replace.
    half_length : float
        L in metres.  The new slice CGs are linearly spaced from
        (cg - L) to (cg + L).
    n_div : int
        Number of slices ( ≥ 2 ).
    mass_col, cg_col, name_col : str
        Column names for mass, CG position and component label.

    Returns
    -------
    DataFrame
        New frame with the original row removed and the slices appended.
    """
    if n_div < 2:
        raise ValueError("n_div must be ≥ 2")

    # 1) locate the row
    mask = df[name_col] == component_name
    if not mask.any():
        raise KeyError(f"{component_name!r} not found in {name_col} column")

    row = df.loc[mask].iloc[0]        # original component
    df_out = df.loc[~mask].copy()     # all other rows

    total_mass = row[mass_col]
    cg0        = row[cg_col]

    # 2) new masses & CGs
    masses   = np.full(n_div, total_mass / n_div)
    cg_new   = np.linspace(cg0 - half_length, cg0 + half_length, n_div)

    # 3) build new rows
    add_rows = pd.DataFrame({
        name_col : [f"{component_name} seg{i+1}" for i in range(n_div)],
        mass_col : masses,
        cg_col   : cg_new,
    })

    # 4) return combined frame (re-index optional)
    return pd.concat([df_out, add_rows], ignore_index=True)
```

In [107…

```
def compute_rocket_bending(rocket_tri: pv.PolyData,
                           df_components: pd.DataFrame,
                           n_slices: int = 105):
    """
    Wraps the user-supplied analysis in a single function.

    Parameters
    ----------
    rocket_tri     : pyvista.PolyData
        Triangulated surface mesh of the rocket wall (.vtp → .extract_surface().triangulate()).
    df_components  : pd.DataFrame
        Component table with columns ['name', 'mass_g', 'cg_mm'] at minimum.
    n_slices       : int, optional
        Number of axial slices for the aerodynamic binning (default 105).

    Returns
    -------
    dict
        All data frames, arrays and scalars generated by the script.
    """

    # -------------- original code (verbatim) ------------------------
    rocket_tri = rocket_tri.compute_cell_sizes()
    area       = rocket_tri.cell_data['Area']
    normals    = rocket_tri.cell_normals

    pts        = rocket_tri.points
    x, y, z    = pts[:, 0], pts[:, 1], pts[:, 2]
    faces      = rocket_tri.faces.reshape(-1, 4)[:, 1:]          # noqa: F841

    rocket_with_cell_data = rocket_tri.point_data_to_cell_data()
    pressure   = rocket_with_cell_data['p']

    force_elements = -pressure[:, None] * area[:, None] * normals
    force_x = force_elements[:, 0]
    force_y = force_elements[:, 1]
    force_z = force_elements[:, 2]

    # ----------------------------------------------------------------
    x_min, x_max = x.min(), x.max()
    dx           = (x_max - x_min) / n_slices

    centres      = rocket_tri.cell_centers().points
    x_centres    = centres[:, 0]
```

```
edges        = np.linspace(x_centres.min(), x_centres.max(), n_slices + 1)
idx          = np.digitize(x_centres, edges, right=True) - 1

Fx = np.zeros(n_slices)
Fy = np.zeros(n_slices)
Fz = np.zeros(n_slices)

np.add.at(Fx, idx, force_x)
np.add.at(Fy, idx, force_y)
np.add.at(Fz, idx, force_z)

Fmag = np.sqrt(Fx**2 + Fy**2 + Fz**2)
mid  = (edges[:-1] + edges[1:]) / 2

df = pd.DataFrame({
    "x_mid_m": mid,
    "Fx_N": Fx,
    "Fy_N": Fy,
    "Fz_N": Fz,
    "Fmag_N": Fmag
})

# repeat block kept exactly as in original script ----------------
centres   = rocket_tri.cell_centers().points
x_centres = centres[:, 0]
edges     = np.linspace(x_centres.min(), x_centres.max(), n_slices + 1)
idx       = np.digitize(x_centres, edges, right=True) - 1

Fx = np.zeros(n_slices)
Fy = np.zeros(n_slices)
Fz = np.zeros(n_slices)

np.add.at(Fx, idx, force_x)
np.add.at(Fy, idx, force_y)
np.add.at(Fz, idx, force_z)

Fmag = np.sqrt(Fx**2 + Fy**2 + Fz**2)
mid  = (edges[:-1] + edges[1:]) / 2

df = pd.DataFrame({
    "x_mid_m": mid,
    "Fx_N": Fx,
    "Fy_N": Fy,
    "Fz_N": Fz,
    "Fmag_N": Fmag
})

# ------------------------------------------------------------------
# component processing (original code untouched)
df_components["mass_kg"] = df_components["mass_g"] / 1000.0
df_components["cg_m"]    = df_components["cg_mm"] / 1000.0

total_mass     = df_components["mass_kg"].sum()
total_forceX   = df["Fx_N"].sum()
total_forceY   = df["Fy_N"].sum()
total_forceZ   = df["Fz_N"].sum()

aY, aX, aZ = (total_forceY / total_mass,
              total_forceX / total_mass,
              total_forceZ / total_mass)

cg_x = (df_components["cg_m"] * df_components["mass_kg"]).sum() / total_mass

df['r'] = df["x_mid_m"] - cg_x
r       = df['r']

M_y = -(r * df["Fz_N"]).sum()
M_z =  (r * df["Fy_N"]).sum()

I_yz = (df_components["mass_kg"] *
        (df_components["cg_m"] - cg_x)**2).sum()

alpha_y = M_y / I_yz
alpha_z = M_z / I_yz

df_components["F_inertial_Y_N"]          = -df_components["mass_kg"] * aY
df_components["F_inertial_Y_N_rotation"] = (
    df_components["mass_kg"] * alpha_z * (cg_x - df_components["cg_m"])
)
df_components["F_inertial_Y_total"]      = (
    df_components["F_inertial_Y_N"] + df_components["F_inertial_Y_N_rotation"]
)

# ------------------------------------------------------------------
# distributed aerodynamic & inertial loads  (replace old block)
# ------------------------------------------------------------------
x_bin  = df["x_mid_m"].to_numpy()     # slice centres (length = n_slices)
```

```
    edges  = np.concatenate(([x_bin[0] - dx/2], x_bin + dx/2))  # slice edges
    Fy_bin = df["Fy_N"].to_numpy()       # aerodynamic per slice

    # Separate contributions
    Fy_translational = np.zeros_like(Fy_bin)
    Fy_rotational    = np.zeros_like(Fy_bin)

    for _, comp in df_components.iterrows():
        k = np.digitize(comp["cg_m"], edges, right=True) - 1
        k = np.clip(k, 0, len(Fy_bin) - 1)
        Fy_translational[k] += comp["F_inertial_Y_N"]
        Fy_rotational[k]    += comp["F_inertial_Y_N_rotation"]

    Fy_point = Fy_translational + Fy_rotational
    Fy_total = Fy_bin + Fy_point

    # Compute bending moments from each source
    shear_translational = -np.cumsum(Fy_bin + Fy_translational)
    shear_rotational    = -np.cumsum(Fy_rotational)

    moment_translational = np.cumsum(shear_translational * dx)
    moment_rotational    = np.cumsum(shear_rotational * dx)

    # Total moment = translational + rotational
    shear_y = shear_translational + shear_rotational
    moment_y = moment_translational + moment_rotational

    # ---------------------------------------------------------------------


    # ---------------------------------------------------------------------
    # AXIAL (X) LOADS  –  aerodynamic drag  +  component inertia
    # ---------------------------------------------------------------------
    # aerodynamic bins (already in df)
    Fx_bin = df["Fx_N"].to_numpy()          # length = n_slices

    # component inertial loads  F = –m aX   (no rotation term in X)
    df_components["F_inertial_X_N"] = -df_components["mass_kg"] * aX

    # put each component into its slice
    Fx_point = np.zeros_like(Fx_bin)
    for _, comp in df_components.iterrows():
        k = np.digitize(comp["cg_m"], edges, right=True) - 1
        k = np.clip(k, 0, len(Fx_point) - 1)
        Fx_point[k] += comp["F_inertial_X_N"]

    # total axial force per slice  (positive = thrust,  negative = drag)
    Fx_total = Fx_bin + Fx_point

    # internal axial load N(x)   (compression +, tension –)
    # start from nose tip so integrate *negative* of external forces
    axial_load = -np.cumsum(Fx_total)
    return {
        # raw and intermediate data
        "df": df,
        "df_components": df_components,
        "edges": edges,
        "dx": dx,
        "Fy_bin": Fy_bin,
        "Fy_point": Fy_point,
        "Fy_total": Fy_total,
        "Fx_total": Fx_total,
        "axial_load": axial_load,
        "Fy_translational": Fy_translational,
        "Fy_rotational": Fy_rotational,
        "moment_translational": moment_translational,
        "moment_rotational": moment_rotational,
        "shear_y": shear_y,
        "moment_y": moment_y,
        "Fx": total_forceX,
        "Fy": total_forceY,
        "Fz": total_forceZ,
        "aX": aX,
        "aY": aY,
        "aZ": aZ,
        "alpha_y": alpha_y,
        "alpha_z": alpha_z,
        "cg_x": cg_x,
    }
```

In [132…

```
df_components_distrib = distribute_component(
    df_components1,
    "Motor (full)",
    half_length=0.2,
    n_div=30)
```

In [ ]:
```
path = "Data/CollectedResultsMesh_New/AoA_5p0_Ma_1p4/postProcessing/rocketVTK/7100/rocketWall.vtp"
#path = "Data/CollectedResultsMesh_New/AoA_1p0_Ma_2p0/postProcessing/rocketVTK/6600/rocketWall.vtp"
rocket = pv.read(path)
rocket = rocket.compute_cell_sizes()
rocket_tri = rocket.extract_surface().triangulate()
```

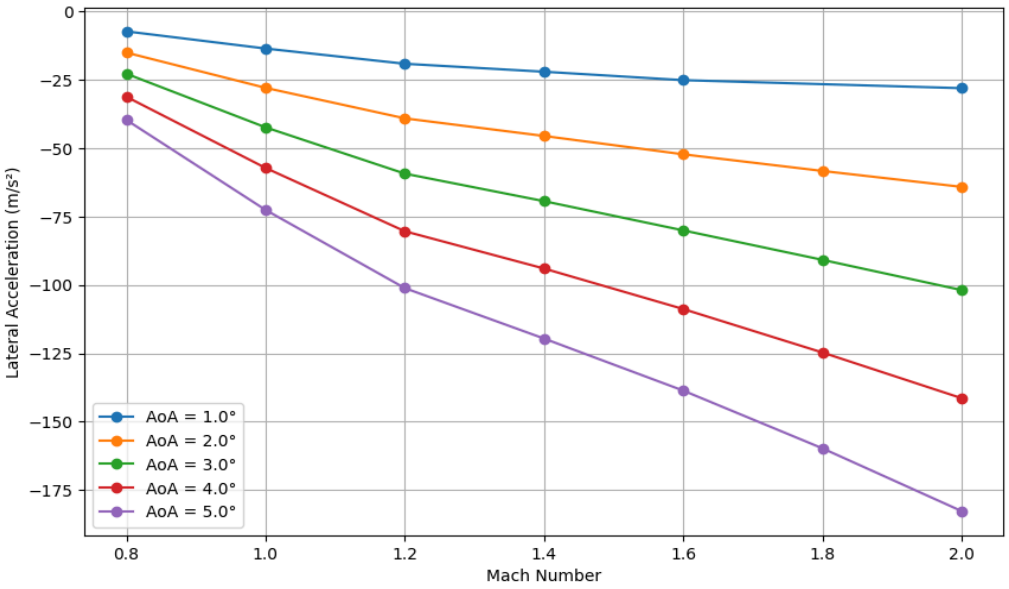
In [133…
```
base_dir = "Data/CollectedResultsMesh_New"

results = []
for case in os.listdir(base_dir):
    if not case.startswith("AoA_"):
        continue

    m = re.fullmatch(r"AoA_([+-]?\d+)p(\d+)_Ma_([+-]?\d+)p(\d+)", case)
    aoa = float(f"{m.group(1)}.{m.group(2)}")   # 2.0
    ma  = float(f"{m.group(3)}.{m.group(4)}")   # 0.8

    t = os.listdir(os.path.join(base_dir, case, "postProcessing", "rocketVTK"))[0]
    path = os.path.join(base_dir, case, "postProcessing", "rocketVTK", t, "rocketWall.vtp")
    rocket = pv.read(path)
    rocket = rocket.compute_cell_sizes()
    rocket_tri = rocket.extract_surface().triangulate()

    result = compute_rocket_bending(rocket_tri, df_components_distrib)
    result["aoa"] = aoa
    result["ma"] = ma

    results.append(result)


the_df = pd.DataFrame(results)
the_df.head()
```

Out[133…

	df	df_components	edges	dx	Fy_bin	Fy_point	Fy_total	
0	x_mid_m Fx_N Fy_N Fz_...	name m...	[0.0003049805762232963 4, 0.026800267771071355,...	0.026495	[-6.362681913392432, -12.00105000655714, -15.4...	[0.0, 0.0, 0.0, 0.0, 0.0, -93.01035899782347, ...	[-6.362681913392432, -12.00105000655714, -15.4...	[-2 -3
1	x_mid_m Fx_N Fy_N Fz_N...	name m...	[0.0003049805858866758, 0.026800267780734736, ...	0.026495	[-1.1881993595594427, -2.2101281237661974, -3....	[0.0, 0.0, 0.0, 0.0, 0.0, -36.084138841465276,...	[-1.1881993595594427, -2.2101281237661974, -3....	[-18 -29
2	x_mid_m Fx_N Fy_N Fz_...	name m...	[0.0003049806051917212, 0.02680026780003978, 0...	0.026495	[-1.733972074090311, -3.0037449994632217, -3.9...	[0.0, 0.0, 0.0, 0.0, 0.0, -66.08088894120708, ...	[-1.733972074090311, -3.0037449994632217, -3.9...	[-17 -27
3	x_mid_m Fx_N Fy_N Fz_N...	name m...	[0.0003049806051917212, 0.02680026780003978, 0...	0.026495	[-0.4453728201587215, -0.748796009043716, -1.0...	[0.0, 0.0, 0.0, 0.0, 0.0, -12.994177252160153,...	[-0.4453728201587215, -0.748796009043716, -1.0...	[-15 -2
4	x_mid_m Fx_N Fy_N Fz_...	name m...	[0.0003049805762232963 4, 0.026800267771071355,...	0.026495	[-1.8665751818486127, -3.4049962928404915, -4....	[0.0, 0.0, 0.0, 0.0, 0.0, -54.94356977181285, ...	[-1.8665751818486127, -3.4049962928404915, -4....	[-18 -29

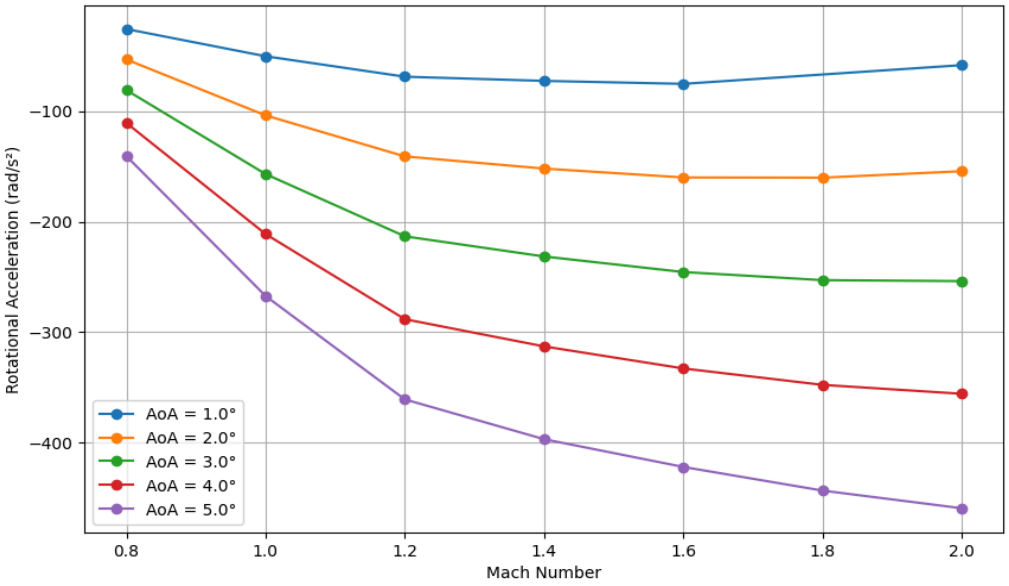
5 rows × 26 columns

In [110…
```
y_column = 'aY'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['aoa'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['aoa'] == aoa].sort_values('ma')
    plt.plot(data['ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("Lateral Acceleration (m/s²)")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)


plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



In [111…

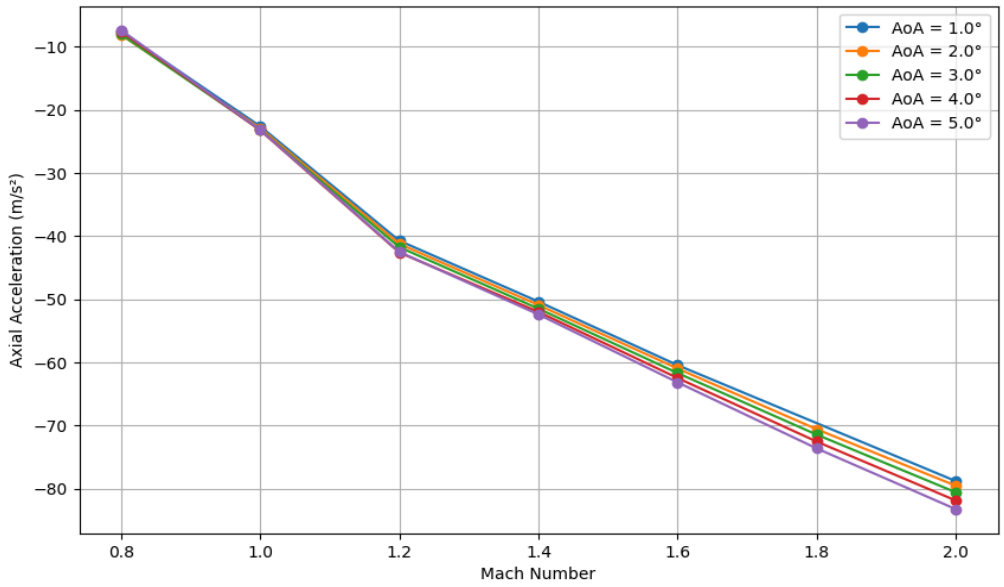
```
y_column = 'alpha_z'  # Change this to the column you want to plot

plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['aoa'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['aoa'] == aoa].sort_values('ma')
    plt.plot(data['ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("Rotational Acceleration (rad/s²)")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)


plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



In [112…

```
y_column = 'aX'  # Change this to the column you want to plot
```

```
plt.figure(figsize=(10, 6))

# Get unique AoA values
aoa_values = sorted(the_df['aoa'].unique())

# Plot each AoA
for aoa in aoa_values:
    data = the_df[the_df['aoa'] == aoa].sort_values('ma')
    plt.plot(data['ma'], data[y_column], 'o-', label=f'AoA = {aoa}°')

plt.xlabel('Mach Number')
plt.ylabel("Axial Acceleration (m/s²)")
#plt.title(f'{y_column} vs Mach Number')
plt.legend()
plt.grid(True)


plt.savefig(f"./results/{y_column}_vs_Ma.pdf", bbox_inches='tight')
```



In [113…

```
COMPONENTS = [
    {"name": "Nose tip assembly",                    "mass_g": 353.99,      "cg_mm": 146.06},
    {"name": "Nosecone",                             "mass_g": 1159.0,      "cg_mm": 598.67},
    {"name": "Main Parachute",                       "mass_g": 691.0,       "cg_mm": 605.0},
    {"name": "Main Shock chords",                    "mass_g": 720.0,       "cg_mm": 677.0},
    {"name": "Large-diam. Fuselage",                 "mass_g": 459.82,      "cg_mm": 909.54},
    {"name": "Large-diam. thrust ring",              "mass_g": 207.10,      "cg_mm": 967.99},
    {"name": "Service Module",                       "mass_g": 3079.0,      "cg_mm": 923.0},
    {"name": "G10 plate",                            "mass_g": 95.47,       "cg_mm": 1013.841},
    {"name": "Ribs (6×)",                            "mass_g": 6*28.53,     "cg_mm": 1097.39},
    {"name": "Rear LD thrust ring",                  "mass_g": 97.48,       "cg_mm": 1038.67},
    {"name": "Transition skin",                      "mass_g": 200.03,      "cg_mm": 1146.69},
    {"name": "Small-diam. fuselage",                 "mass_g": 514.393,     "cg_mm": 1213.54},
    {"name": "SD fuselage thrust ring",              "mass_g": 105.72,      "cg_mm": 1173.23},
    {"name": "Drogue chute",                         "mass_g": 569.0,       "cg_mm": 1291.0},
    {"name": "Drogue chords",                        "mass_g": 720.0,       "cg_mm": 1336.0},
    {"name": "Small-diam. coupler",                  "mass_g": 425.07,      "cg_mm": 1371.35},
    {"name": "Small-diam. coupler bulkhead assembly", "mass_g": 283.89,     "cg_mm": 1444.68},
    {"name": "Motor (full)",                         "mass_g": 16525.0,     "cg_mm": 2160.0},
    {"name": "Motorcase",                            "mass_g": 740.11,      "cg_mm": 2087.74},
    {"name": "Fins (3×)",                            "mass_g": 3*235.11,    "cg_mm": 2695.61},
]

df_components1 = pd.DataFrame(COMPONENTS)

df_components1["mass_kg"] = df_components1["mass_g"] / 1000.0
df_components1["cg_m"] = df_components1["cg_mm"] / 1000.0
```

In [118…

```
pressure = rocket_tri.point_data['p']
x = rocket_tri.points[:, 0]
y = rocket_tri.points[:, 1]
z = rocket_tri.points[:, 2]

faces = rocket_tri.faces.reshape(-1, 4)[:, 1:]          # noqa: F841
```

In [121…

```
ma_target = 2.0
rows_ma = the_df[np.isclose(the_df["ma"], ma_target)]
if rows_ma.empty:
    raise ValueError(f"No runs at Mach {ma_target}")

# first run supplies pressure map & stems
row0    = rows_ma.iloc[0]
df_bin0 = row0["df"]
#df_comp = row0["df_components"]

# ------------------------------- figure ------------------------------
fig, axes = plt.subplots(6, 1, figsize=(14, 18), sharex=True)

# 0) pressure map ---------------------------------------------------------
tc = axes[0].tripcolor(x, y, faces, pressure, shading="flat", cmap="jet")
cax = inset_axes(axes[0], width="60%", height="4%", loc="lower center",
                 bbox_to_anchor=(0, 0.9, 1, 1),
                 bbox_transform=axes[0].transAxes)
fig.colorbar(tc, cax=cax, orientation="horizontal").set_label("p [Pa]")
axes[0].grid(False)
for sp in axes[0].spines.values():
    sp.set_visible(False)
axes[0].tick_params(left=False, bottom=False,
                    labelleft=False, labelbottom=False)

# 1) axial internal load N(x) ---------------------------------------------
colors = plt.rcParams["axes.prop_cycle"].by_key()["color"]
for i, (_, rw) in enumerate(rows_ma.sort_values("aoa").iterrows()):
    col   = colors[i % len(colors)]
    lbl   = f"AoA {rw['aoa']:.1f}°"
    xmid  = rw["df"]["x_mid_m"].values

    # compute axial load if not stored
    if "axial_load" in rw:
        axial = rw["axial_load"]
    else:  # derive on the fly
        Fx_total = rw["Fx_bin"] + rw["Fx_point"]
        axial = -np.cumsum(Fx_total)

    axes[1].plot(xmid, axial, color=col, label=lbl)

axes[1].set_ylabel("Axial load [N]")
axes[1].set_title("Internal axial load")
axes[1].legend(ncol=2)
axes[1].axhline(0, ls=":", color="gray")
axes[1].grid(True)

# 2) stems from first run -------------------------------------------------
aY0, alpha_z0, cg_x0 = row0["aY"], row0["alpha_z"], row0["cg_x"]
trans0 = -df_components_distrib["mass_kg"] * aY0
rot0   =  df_components_distrib["mass_kg"] * alpha_z0 * (cg_x0 - df_components_distrib["cg_m"])

axes[2].stem(df_components_distrib["cg_m"], trans0, basefmt=" ",
             linefmt="b-", markerfmt="bo", label="Trans-inertial")
axes[2].stem(df_components_distrib["cg_m"], rot0,   basefmt=" ",
             linefmt="r-", markerfmt="rs", label="Rot-inertial")
axes[2].set_ylabel("Inertial Normal Load [N]")
axes[2].set_title("Component inertial loads")
axes[2].legend()
axes[2].axhline(0, ls=":", color="gray")
axes[2].grid(True)

# 3-5) Fy_bin, shear, moment for each AoA ---------------------------------
for i, (_, rw) in enumerate(rows_ma.sort_values("aoa").iterrows()):
    col   = colors[i % len(colors)]
    lbl   = f"AoA {rw['aoa']:.1f}°"
    dloc  = rw["df"]
    xmid  = dloc["x_mid_m"].values

    axes[3].plot(xmid, rw["Fy_bin"],  color=col, label=lbl)
    axes[4].plot(xmid, rw["shear_y"], color=col, label=lbl)
    axes[5].plot(xmid, rw["moment_y"], color=col, label=lbl)

axes[3].set_ylabel("Aerodynamic Force [N]")
axes[3].set_title("Aerodynamic normal force")

axes[4].set_ylabel("Shear Force [N]")
axes[4].set_title("Shear force diagram")

axes[5].set_ylabel("Bending Moment [N·m]")
axes[5].set_xlabel("x (m from nose tip)")
axes[5].set_title("Bending moment diagram")

# single legend on panel 3
axes[3].legend(title=f"Mach {ma_target}", ncol=2)
axes[4].legend().set_visible(False)
axes[5].legend().set_visible(False)
```

```
# zero lines on panels 2-5
for idx in (2, 3, 4, 5):
    axes[idx].axhline(0, color="gray", linestyle=":", linewidth=1.2)
    axes[idx].grid(True)


plt.savefig(f"./results/forces_Ma2.pdf", bbox_inches='tight')
```

```
/tmp/ipykernel_102457/2905843558.py:94: UserWarning: This figure includes Axes that are not compatible with tight
_layout, so results might be incorrect.
  fig.tight_layout()
```