Reduced-Order Model for Efficiently Generating Subsonic Aerodynamic Database of a Missile

by

Aniruddha Sinha

Associate Professor

Department of Aerospace Engineering

Indian Institute of Technology Bombay

May 15, 2020

Aniruddha Sinha

Contents

1 Executive Summary

A project was proposed to the Defence Research and Development Laboratory (DRDL), Hyderabad, to develop a fast method for generating the aerodynamic database for subsonic flow over a missile configuration. This report documents the successful completion of the same.

At present computer simulations are performed using ANSYS Fluent software for a number of carefully chosen operating conditions covering a range of Mach number, angle of attack and roll angles of the missile. The proposed approach leverages such an existing database to train a reduced-order model (ROM), that can subsequently be executed very efficiently to predict the aerodynamic forces and moments for any other operating condition within the original range, and somewhat beyond it. The ROM implemented herein uses proper orthogonal decomposition (POD) to learn the essential features of the empirical database, and subsequently uses the governing equations in an optimization paradigm to incorporate the flow physics.

The typical evaluation time of the ROM is less than half a minute, which is negligible compared to the CFD simulation. The predictive capability of the ROM is quite encouraging – it is able to estimate the aerodynamic forces and moments on the missile within 2% for most cases, and within 10% at worst. These percentages are relative to the maximum values of the respective coefficients encountered in the learning database. In comparison, it is shown here that straightforward linear interpolation using the POD modes (but without the optimization step) results in errors that are an order of magnitude higher.

The ROM is encoded in a suite of Python programs developed as part of this project. The programs provide an interface with the ANSYS Fluent data structure, thereby incorporating several flexible features. The programs also calculate the POD modes, and perform the optimization of the solution so as to approximately satisfy the governing equations. Finally, the aerodynamic coefficients are calculated from the flow solutions. Note that the approach delivers the approximate flow field in the entire CFD flow domain, and not just the surface-integrated forces and moments.

The algorithms and programs developed in the project are being transferred to DRDL scientists.

2 Introduction

The design of a missile involves consideration of several variants of the geometry. A fundamental requirement of this process is a detailed aerodynamic database for each of these design choices, listing the forces and moments acting on the missile for a large range of conditions – e.g. various combinations of Mach numbers, angles of attack, and roll angles. Creation of such aerodynamic databases poses a significant cost to organizations involved in the development of novel configurations. With the availability of increasing computing power and improved numerical algorithms, most of this activity has shifted from physical flight tests and wind tunnel experiments to the virtual environment of computational fluid dynamics (CFD). The state-of-the-art CFD invariably involves a large-scale 'high-order model' (HOM), a terminology that distinguishes this model from the more approximate reduced-order model (ROM) outlined subsequently. Depending on the accuracy required, the designer may seek quick Euler (inviscid) solutions or costlier Reynolds-averaged Navier-Stokes (RANS) solutions (that account for molecular and eddy viscosity). In either case, the state-of-the-art HOMs yield *steady* flow solutions as unsteady simulations are neither computationally feasible nor necessary for the engineering answers being sought. Unfortunately, for the complex missile configurations of practical relevance, the computational expense of generating large steady aerodynamics databases is still very high.

Over the years, a variety of methods have been explored for reducing the computational costs of these databases, with minimal attendant degradation in accuracy. All these methods involve empiricism; i.e., the designer must first create an HOM database by sparsely sampling the freestream parameter domain (and/or other geometric parameters like flap deflection angle, aeroelastic deformation condition, etc. for wings, say) for a particular configuration. These flow solutions are termed 'snapshots'. This step requires heavy computational effort – it represents the upfront cost of these empirical approaches. Subsequently, the methods promise to efficiently predict the flow solutions (or their salient properties, like aerodynamic coefficients) using the empirical database of snapshots. Note that fewer HOM snapshots need to be simulated in this approach with the rest being predicted at very low cost, so that the upfront cost incurred is actually less than in the generation of current state-of-the-art HOM database. These methods fall in one of two broad categories – interpolation methods and reduced-order models (ROMs).

Interpolation methods, as the name suggests, involve interpolation of empirical snapshots on the basis of their freestream conditions to determine the flow solution for a new freestream condition that was not evaluated in the original database. These methods are usually fast, and can deliver results of acceptable accuracy if the flow features are approximately similar between the new condition and those evaluated for the original database. A large variety of interpolation methods are reported in the literature. They may be classified based on the interpolated quantity. The simplest are those that directly interpolate the integrated aerodynamic quantities – viz. coefficients of lift, drag etc. Of intermediate complexity and accuracy are approaches that interpolate the surface flow variables (pressure and shear stress). The other end of the spectrum is occupied by methods that attempt to predict the flow solution over the entire flow domain for new parameter sets of interest. Of particular interest in the latter approach is the use of proper orthogonal decomposition (POD) to obtain an approximately complete basis of flow features prior to interpolation applied to the POD weighting coefficients[1] (more about POD later).

In discussing the earlier reduced-order modelling approaches for flows, we make a distinction between unsteady and steady ROMs. For problems dominated by time-varying flow fields, unsteady ROMs are required to approximately determine the essential flow features in a time-resolved manner – examples may be found in Refs. 2–6. On the other hand, steady ROMs find utility in problems where knowledge of the steady (usually time-averaged) component of the flow field is sufficient for

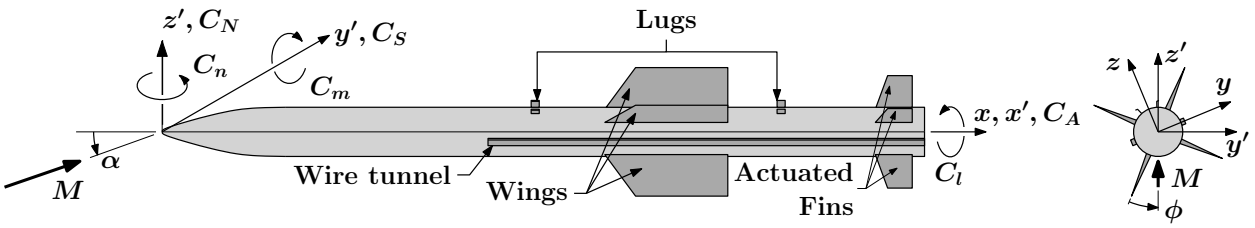


Figure 1: Side and rear views of the configuration of a typical missile to which the ROM method is applied. The total angle of attack coordinate system is $x' - y' - z'$, with the $x' - z'$ plane containing the relative wind vector. The body-fixed nominal coordinate system is $x - y - z$; the two match in the nominal roll orientation, i.e. $\phi = 45°$. The x'-axis always coincides with the x-axis. Also depicted is the sign convention of the aerodynamic force and moment coefficients; these refer to the total angle of attack reference frame.

engineering purposes, as in the present case. Thus, steady ROM approaches, as reported in Refs. 7–11, are most relevant for the problem at hand.

The steady ROM development starts from snapshots of the steady flow field for different values of a set of parameters of the problem, just like in the case of interpolation. The next step is the eduction of a set of basis functions (or modes) that represents the dominant flow features observed across all snapshots. In the literature, this is consistently pursued with proper orthogonal decomposition (POD). POD is a statistical procedure that delivers an orthogonal transformation to convert a set of observations (snapshots) of possibly correlated variables (flow fields) into a set of linearly uncorrelated variables (basis functions/modes)[12,13]. The POD modes afford a low-order representation of the steady flow fields. Being an empirical method, the POD modes derived are entirely dependent on the richness of the snapshots – i.e., the snapshots should cover the range of flow fields that are expected to be encountered by the ROM.

The final step is the prediction of the steady flow solution for a particular parameter set that was not computed in the snapshot-creation step. In case of ROMs, this new flow solution is derived as a linear combination of the POD modes that optimally satisfies the steady governing equations of the flow along with appropriate boundary conditions[7–10]. The steady ROM method is more robust and versatile than the interpolation method since it is guided by the governing equations, although it is more computationally expensive for the same reason. The ROM approach has been shown to be useful for predicting the steady aerodynamics of an aircraft wing/fin in both 2D[7–9] and 3D[10] problems.

Here we report on the further development and application of the POD-ROM approach to the problem of predicting the aerodynamic characteristics of a missile. The scope of the present work is limited to a regime of subsonic missile flight where shocks are absent; modelling shocks in a POD setting is complicated[8], and will be pursued in the future. Even with this constraint, the ROM tool may partially replace expensive HOM calculations in aerodynamic characterization, leading to large savings in computational time and cost. Indeed, we demonstrate how well the ROM can replicate the results of the RANS, at a very small fraction of the computational cost.

3 Missile configuration being studied and its HOM database

To motivate the exposition of the POD-ROM theory to follow, we first describe the air-launched missile configuration being studied; a sketch is shown in fig. 1. The missile has a set of four wings and another nominally coplanar set of four tail fins; the latter are actuated for controlling the

Parameter	Values
Mach number, M	0.6, 0.8
Angle of attack, α (degrees)	0, 2, 5, 10, 15, 20
Roll angle, ϕ (degrees)	0, 22.5, 45

Table 1: Parameter value combinations in 'learning database' of subsonic missile aerodynamics.

trajectory. In the nominal roll orientation, the wings and fins form an 'X' with respect to the plane containing the longitudinal axis and the relative wind vector. A pair of external wire tunnels run along the length of the missile, one on each side. A set of 4 lugs at the top allow for anchoring of the missile underneath an airplane.

Two separate coordinate systems are used to describe the missile. The $x-y-z$ system is fixed to the body, such that y-axis is normal to the relative wind vector in the nominal roll orientation. This will be called the 'body-fixed nominal coordinate system'. In the industry, however, the $x'-y'-z'$ system is prevalent. The x'-axis coincides with the longitudinal axis of the missile, and hence the x-axis; the y'-axis is always oriented so as to be normal to the relative wind vector. This is termed the 'total angle of attack frame'.

The convention for the roll angle ϕ is also specific to the industry. In particular, $\phi = 45^\circ$ is the nominal roll orientation, with the relative wind vector lying in the first quadrant of the $x-z$ plane (and also the coincident $x'-z'$ plane). If the relative wind vector happens to be in the fourth quadrant of the $x-z$ plane, then we say that $\phi = -135^\circ$. On the other hand, ϕ is 0° when the relative wind is in the plane of the bottom left wing/fin with reference to the rear view in fig. 1. In summary, ϕ varies in the range -180° to $+180^\circ$, and the angle of attack α only takes positive values; usually $\alpha \leq 20^\circ$.

Figure 1 also presents the sign convention for the aerodynamic force and moment coefficients. They refer to the total angle of attack coordinate system. The force coefficients are axial C_A, side C_S and normal C_N. The corresponding components of the moment coefficient are rolling C_l, pitching C_m and yawing C_n; the moment arm is measured from the missile nose. In the nominal roll orientation $\phi = 45^\circ$, the coefficients of side force, roll moment and yaw moment all vanish due to the symmetry of the missile. As usual, the characteristic length, area, speed and density for defining the coefficients are the missile diameter D, cross-sectional area $\pi D^2/4$, freestream speed and ambient density, respectively.

Table 1 presents the combinations of flight conditions comprising the learning database for the empirical ROM. There are two subsonic Mach numbers, six angles of attack and three roll angles; note that none of these cases displayed shocks in the flow field. Although all possible combinations should result in 36 snapshots, only 32 of these are unique since the roll angle ceases to matter at $\alpha = 0^\circ$. If the minor protuberances of the two wire tunnels and four lugs were absent, then the missile would have had a four-fold rotational symmetry and a two-fold mirror symmetry. The minor symmetry-breaking features are indeed ignored for the purpose of the aerodynamic database, which explains the limited range of roll angles simulated.

The empirical database was generated entirely by scientists at the Defence Research and Development Laboratory (DRDL), under the Defence Research and Development Organization (DRDO) of India.

The flow domain was a cube of side about 160 times the missile diameter D, centred on the nose. The edges of the cube were aligned with the body-fixed coordinate axes shown in fig. 1. A three-

dimensional hexahedral unstructured grid was generated in HEXPRESS (version 6.0; NUMECA International Company; Brussels, Belgium), with approximately 10 million cells. The mesh around the missile surface had fine enough spacing to resolve the viscosity-affected near-wall region all the way to the laminar sublayer so as to ensure that y^+ in the wall-adjacent cell was on the order of unity.

The flow simulations were performed using ANSYS Fluent (version 15.0)[14], solving the $k-\omega$ SST steady RANS equations using a pressure-based coupled finite volume solver. Details of the solver can be found in the user manual. We only note that the cell face values required in computing the convective terms were calculated with a second-order upwind scheme, using least-squares cell-based gradients.

A 'pressure far-field' condition was specified on the inlet boundary consisting of the upstream face as well as the four side faces of the cubic flow domain. Zero gauge pressure and an absolute temperature of 303.15 K (sea level conditions) were consistently applied at the inlet for all snapshots. Only the Mach number and the flow direction were varied on the inlet per the specifications in table 1. The incoming turbulence was specified by an intensity of 1% and a hydraulic diameter equal to D. The sixth face of the cube downstream of the missile constituted the outlet boundary on which a 'pressure outlet' condition was specified. For this, the pressure, temperature and turbulence parameters were same as for the inlet; back-flow, if any, was stipulated to be normal to the boundary. The missile surface was specified to be a no-slip adiabatic wall.

4 A brief recapitulation of POD

What follows is a discussion of some of the aspects of Proper Orthogonal Decomposition that are most relevant for the present work; full details of the method have been presented in many other publications[2,15]. There are several alternatives available in POD; we will use snapshot-based vector POD on flow data with their mean (over all the snapshots) subtracted. These aspects will be clarified below.

Let $\boldsymbol{q}(\boldsymbol{x})$ represent the vector field of flow variables with the spatial position coordinate $\boldsymbol{x}$ ranging over the flow domain, Ω, say. For the 3D Euler equations, one choice of $\boldsymbol{q}$ is $[\rho, \rho u, \rho v, \rho w, p]^{\mathrm{T}}$, where p and ρ are the pressure and density respectively, and u, v and w are respectively the x, y and z Cartesian components of velocity. This particular choice is recommended in the literature[8,10], and will be argued to be particularly appropriate subsequently. In the following, we use non-dimensional quantities. In particular, linear dimensions are non-dimensionalized by the diameter of the missile D, velocity by the ambient speed of sound c_∞, density by the ambient density ρ_∞, and pressure by $\rho_\infty c_\infty^2$. Note that the non-dimensional pressure in the freestream is γ^{-1}, with γ being the ratio of the specific heats of air. Also, the ideal gas law in non-dimensional form becomes $p = \rho T/\gamma$, where temperature T is non-dimensionalized by the ambient temperature T_∞. Note that, per the discussion in Section 3, all the snapshots had the same values of c_∞ and ρ_∞, so that the above non-dimensional flow variables were directly comparable across the entire empirical database.

In actual implementation, any (infinite-dimensional) flow field under consideration $\boldsymbol{q}(\cdot)$ must be represented on a discretized grid as a finite-dimensional vector. With an abuse of notation, we denote the latter by $\boldsymbol{q}$ also, relying on the context to clarify the situation. For example, with a grid consisting of a million elements (the elements may be chosen to be nodes, cells or even faces), and for a vector flow field with five components, $\boldsymbol{q}$ will be a five million long column vector.

A particular steady operating condition of the missile (specified by, for example, the triad of its freestream Mach number, angle of attack and roll angle) is denoted by $\boldsymbol{\mu}$. Note that this is a vector; for example $\boldsymbol{\mu} = (M, \alpha, \phi)^{\mathrm{T}}$. Let such steady flow solutions be available for S operating conditions (for reference, $S = 32$ in our problem). Thus, the database of 'snapshots' is $\{\boldsymbol{q}(\boldsymbol{x}; \boldsymbol{\mu}_s)\}_{s=1}^{S}$. Here, $\boldsymbol{\mu}_s$

denotes the operating condition of the sth snapshot. For notational convenience, we will sometimes denote this database as $\{\boldsymbol{q}_s(\boldsymbol{x})\}_{s=1}^{S}$.

Let the mean flow field across the database be $\overline{\boldsymbol{q}}(\boldsymbol{x})$; that is $\overline{\boldsymbol{q}}(\boldsymbol{x}) := (1/S)\sum_{s=1}^{S}\boldsymbol{q}_s(\boldsymbol{x})$. Then, the snapshots' 'fluctuations' are given by $\boldsymbol{q}'(\boldsymbol{x};\boldsymbol{\mu}) := \boldsymbol{q}(\boldsymbol{x};\boldsymbol{\mu}) - \overline{\boldsymbol{q}}(\boldsymbol{x})$. In particular, the database of snapshot fluctuations becomes $\{\boldsymbol{q}'_s(\boldsymbol{x})\}_{s=1}^{S}$.

In mean-subtracted vector POD, we assume that any flow vector field (not only those included in the learning database) can be approximated by a linear combination of N POD modal vector fields added to the above mean flow vector field (which is calculated on the learning database only):

$$\boldsymbol{q}(\boldsymbol{x};\boldsymbol{\mu}) = \overline{\boldsymbol{q}}(\boldsymbol{x}) + \boldsymbol{q}'(\boldsymbol{x};\boldsymbol{\mu}), \qquad \boldsymbol{q}'(\boldsymbol{x};\boldsymbol{\mu}) \approx \sum_{n=1}^{N}\eta^n(\boldsymbol{\mu})\widetilde{\boldsymbol{q}}^n(\boldsymbol{x}). \tag{1}$$

Here, $\widetilde{\boldsymbol{q}}^n(\boldsymbol{x})$ is the nth POD mode defined over the flow domain Ω; note that it is a *vector* field with the same components as $\boldsymbol{q}$. In particular, if p, say, is a component of $\boldsymbol{q}$, then the corresponding component of the POD mode $\widetilde{\boldsymbol{q}}^n(\boldsymbol{x})$ is denoted by $\widetilde{p}^n(\boldsymbol{x})$. Also, η^n is the corresponding POD coefficient; it is parameterized by the operating condition $\boldsymbol{\mu}$. For later use, we denote the vector of POD coefficients for operating condition $\boldsymbol{\mu}$ as $\boldsymbol{\eta}(\boldsymbol{\mu}) := \big(\eta^1(\boldsymbol{\mu}),\cdots,\eta^N(\boldsymbol{\mu})\big)^{\mathrm{T}}$. Moreover, in agreement with the prevailing notation, we denote the POD coefficient corresponding to the sth snapshot in the learning database as $\eta^n_s := \eta^n(\boldsymbol{\mu}_s)$.

For later reference, the stacked two-dimensional arrays of finite-dimensional fluctuation snapshots and POD modes are denotes as

$$\boldsymbol{Q}' := \begin{bmatrix} \boldsymbol{q}'_1 & \boldsymbol{q}'_2 & \cdots & \boldsymbol{q}'_S \end{bmatrix}.$$

This should be recognized as a 'tall and thin' matrix, with S columns but rows that number several orders of magnitude more (the row count is about 50 million in our problem).

In contrast, in the mean-retained POD variant, the POD modes directly reconstruct the flow snapshots, and not their 'fluctuations'. In scalar POD variant, each flow variable (p, ρv, etc. in the prevailing example) is decomposed into its own set of *scalar* POD modes, weighted by its respective *independent* set of POD coefficients.

To be able to assess the efficacy of the above approximation, the POD procedure requires an inner product to be defined *a priori*. As an example, in the present work, the inner product of two flow vector fields $\boldsymbol{q}$ and $\boldsymbol{r}$ is defined as

$$\langle \boldsymbol{q}(\cdot), \boldsymbol{r}(\cdot)\rangle := \int_{\Omega}\left[q_\rho r_\rho + q_{\rho u}r_{\rho u} + q_{\rho v}r_{\rho v} + q_{\rho w}r_{\rho w} + q_p r_p\right]\mathrm{d}\Omega, \tag{2}$$

where $q_{\rho v}$ refers to the $y-$component of mass flux in the flow field $\boldsymbol{q}$, and so on, and Ω is a part or the whole of CFD solution domain. Since the inner product is by definition bilinear in its arguments, its finite dimensional representation is $\boldsymbol{r}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{q}$, where $\boldsymbol{W}$ is a symmetric positive definite matrix (it is usually diagonal, as in our problem).

The snapshot POD method[13] starts (from the original POD approach[12]) by noting that the POD modes must themselves be in the space spanned by the fluctuation snapshots. That is,

$$\widetilde{\boldsymbol{q}}^n(\boldsymbol{x}) = \sum_{s=1}^{S}\zeta^n_s\boldsymbol{q}'_s(\boldsymbol{x}); \qquad \text{or} \quad \widetilde{\boldsymbol{q}}^n = \boldsymbol{Q}'\boldsymbol{\zeta}^n, \quad \text{with } \boldsymbol{\zeta}^n := \begin{bmatrix} \zeta^n_1 & \zeta^n_2 & \cdots & \zeta^n_S \end{bmatrix}^{\mathrm{T}}. \tag{3}$$

The POD modes are determined[13,15] indirectly by calculating the weight vectors $\{\boldsymbol{\zeta}^n\}_{n=1}^{N}$ from the following eigenvalue problem

$$\frac{1}{S}\sum_{\ell=1}^{S}\langle \boldsymbol{q}'_\ell, \boldsymbol{q}'_s\rangle\zeta^n_\ell = \lambda^n\zeta^n_s, \quad s = 1,\ldots,S; \qquad \text{or} \quad \frac{1}{S}\boldsymbol{Q}'^{\mathrm{T}}\boldsymbol{W}\boldsymbol{Q}'\boldsymbol{\zeta}^n = \lambda^n\boldsymbol{\zeta}^n. \tag{4}$$

The kernel of the snapshot POD eigenvalue problem $S^{-1}\boldsymbol{Q'}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{Q'}$ is evidently symmetric and positive semi-definite (if the mean were not subtracted, then it would have been positive definite as long as no two snapshots were same). Therefore, the eigenvalues λ^n are guaranteed to be real and non-negative. The eigenvalues are ordered in decreasing order; i.e.,

$$\lambda^1 \geq \lambda^2 \geq \cdots \lambda^N \geq 0.$$

With this sequencing, $\widetilde{\boldsymbol{q}}^1$ is called the first POD mode, $\widetilde{\boldsymbol{q}}^2$ is the second POD mode, and so on; this also reflects the order of salience of their flow features in the database. One can conclude from the formulation of the snapshot method that $N \leq S$; physically, N should also be smaller than S so that the POD may filter out the inessential flow features or 'noise' instead of over-fitting the data.

The POD modes are also called POD eigenvectors or POD eigenfunctions. Owing to the special properties of the POD kernel mentioned above, the POD modes form a mutually orthogonal set. For ease of use, they are always unitized to obtain an orthonormal basis. Then, it follows from the POD expansion of eqn. (1) that the POD coefficients are orthogonal projection coefficients:

$$\eta^n(\boldsymbol{\mu}) = \left\langle \boldsymbol{q}'(\cdot;\boldsymbol{\mu}), \widetilde{\boldsymbol{q}}^n(\cdot) \right\rangle. \tag{5}$$

Remark 1 *Orthonormality: That the POD modes are mutually orthogonal is well known. In trying to prove this, we can come up with the following simple normalization rule. Let us evaluate the inner product*

$$\langle \widetilde{\boldsymbol{q}}^n, \widetilde{\boldsymbol{q}}^m \rangle = \left\langle \sum_\ell \zeta_\ell^n \boldsymbol{q}'_\ell, \sum_s \zeta_s^m \boldsymbol{q}'_s \right\rangle = \sum_s \zeta_s^m \sum_\ell \langle \boldsymbol{q}'_\ell, \boldsymbol{q}'_s \rangle \zeta_\ell^n = S \sum_s \zeta_s^m \zeta_s^n \lambda^n = S\lambda^n ||\boldsymbol{\zeta}^m||^2 \delta_{m,n}.$$

The penultimate step is a consequence of the eigenvalue problem definition (see eqn. (4)); the last step follows from the orthogonality of the eigenvectors of the symmetric POD kernel.

The above derivation shows that orthonormal POD modes arise if the following normalization is done

$$\widetilde{\boldsymbol{q}}^n = (S\lambda^n)^{-1/2} ||\boldsymbol{\zeta}^n||^{-1} \boldsymbol{Q'} \boldsymbol{\zeta}^n.$$

Comparing the above with eqn. (3) suggests that $\boldsymbol{\zeta}^n$ should be scaled such that $||\boldsymbol{\zeta}^n|| = (S\lambda^n)^{-1/2}$. Usually, most eigenvalue solvers yield eigenvectors that are unit vectors. Even if $\boldsymbol{\zeta}^n$ is not a unit vector, it is easy to find its 2-norm since S is typically not very large. On the other hand, unitizing the POD eigenvector $\widetilde{\boldsymbol{q}}^n$ is expensive, but is conveniently obviated in the above calculation.

Remark 2 *To compute POD coefficients of snapshots in the database itself, we start from eqn. (5):*

$$\eta_s^n = \langle \boldsymbol{q}'_s, \widetilde{\boldsymbol{q}}^n \rangle = \left\langle \boldsymbol{q}'_s, \sum_{\ell=1}^{S} \zeta_\ell^n \boldsymbol{q}'_\ell \right\rangle = \sum_{\ell=1}^{S} \zeta_\ell^n \langle \boldsymbol{q}'_s, \boldsymbol{q}'_\ell \rangle = S\lambda^n \zeta_s^n.$$

In the second step, we have used the expansion of eqn. (3); the third step utilizes the bilinearity of the inner-product; the last step follows from eqn. (4). Thus, the low-dimensional eigensolution at hand can be re-used for this computation, averting the expensive projection indicated in eqn. (5).

Remark 3 *Theoretically, a discontinuous function requires infinitely many basis functions for accurate realization – recall the Gibb's phenomenon in Fourier theory. Thus, without additional sophistication[8], a POD basis cannot represent a variety of flow fields with shocks at different locations. Here we remain strictly in the subsonic flow regime to avoid such issues.*

Remark 4 *The POD modes are spatial basis functions that are applicable to all choices of* $\boldsymbol{\mu}$*. As such, the* $\boldsymbol{q}$ *data for different* $\boldsymbol{\mu}$*'s must be presented to the POD algorithm on the same geometry and grid. In particular, the different angles of attack and roll angles of the missile should be represented by different angles of inflow conditions, and not by different orientations of the body or grid. Also, it is impossible to handle cases where the tail fins of the missile are deflected without additional modelling steps, and we omit such situations in this work.*

Remark 5 *The inner product underlying the POD method that determines the sense in which the POD modes optimally represent the data, need not be evaluated over the entire flow domain. Instead, the critical regions of the flow, say* Ω_P*, may be identified where the snapshots display the greatest variations*[8–10]*. The POD modes thus obtained may yet be used to represent the flow over the entire snapshot domain. Essentially, we are allowing the weighting matrix* $\boldsymbol{W}$ *to be positive* semi-*definite, with consequent implications for the results.*

5 POD-ROM approach for steady aerodynamics

The basic methodology for creating a ROM for steady aerodynamics prediction of a missile is adopted from LeGresley and Alonso[7]. This approach has been refined subsequently by Alonso et al.[8],[10] and Zimmermann and Görtz[9]. With the preceding notation of the vector field of flow variables $\boldsymbol{q}$ and the vector operating condition of the missile $\boldsymbol{\mu}$, the problem is formally stated as

$$\left(\frac{\partial}{\partial t}\mathscr{C} + \mathscr{R}\right)(\boldsymbol{q}(\boldsymbol{x}, t; \boldsymbol{\mu})) = 0 \quad \text{for } \boldsymbol{x} \in \Omega \qquad \text{subject to} \quad \mathscr{B}(\boldsymbol{q}(\boldsymbol{x}, t; \boldsymbol{\mu})) = 0 \quad \text{for } \boldsymbol{x} \in \partial\Omega. \tag{6}$$

Here, $\mathscr{C}$ is the operator that maps the primitive flow variables $\boldsymbol{q}$ to the conserved flow variables, and $\mathscr{R}$ is a shorthand notation for the terms other than the time-derivative in the conservation equations. These vector conservation equations must be satisfied over the flow domain Ω at all times. The vector of conditions that must be satisfied on the domain boundary $\partial\Omega$ (at all times) are represented as $\mathscr{B}$. Note that for a candidate *steady* solution, $\mathscr{R}$ represents the 'residual' function. Since we are interested in the steady solution of the governing equations for a particular choice of $\boldsymbol{\mu}$, say $\boldsymbol{\mu}_0$, we need to find the vector field $\boldsymbol{q}_0(\boldsymbol{x})$ such that $\mathscr{R}(\boldsymbol{q}_0(\boldsymbol{x}))$ vanishes, subject to the prescribed boundary conditions.

With the approximate vector POD expansion of eqn. (1), we can formally represent the approximate steady residual and boundary conditions as

$$\mathscr{R}(\boldsymbol{q}(\boldsymbol{x}; \boldsymbol{\mu})) \approx \widetilde{\mathscr{R}}(\boldsymbol{x}; \boldsymbol{\eta}(\boldsymbol{\mu})), \qquad \mathscr{B}(\boldsymbol{q}(\boldsymbol{x}; \boldsymbol{\mu})) \approx \widetilde{\mathscr{B}}(\boldsymbol{x}; \boldsymbol{\eta}(\boldsymbol{\mu})), \tag{7}$$

where the modified residual vector field $\widetilde{\mathscr{R}}$ and modified boundary condition vector $\widetilde{\mathscr{B}}$ depend implicitly on the POD modal basis.

Since the truncated POD expansion is an approximation, in general we will not be able to find $\boldsymbol{\eta}(\boldsymbol{\mu})$ such that $\widetilde{\mathscr{R}}$ vanishes exactly simultaneously at all points in the domain. Instead, we define a positive semi-definite functional J whose minimum will correspond to the closest approximation of the desired solution possible with the retained POD modes. The optimization problem is thus formulated as, given a $\boldsymbol{\mu}$ $(= \boldsymbol{\mu}_0$, say),

$$\min_{\boldsymbol{\eta}} J(\boldsymbol{\eta}(\boldsymbol{\mu})) := \left|\left|\widetilde{\mathscr{R}}(\boldsymbol{\eta}(\boldsymbol{\mu}))\right|\right|_{\Omega} \qquad \text{subject to} \quad \left|\left|\widetilde{\mathscr{B}}(\boldsymbol{\eta}(\boldsymbol{\mu}))\right|\right|_{\partial\Omega} = 0. \tag{8}$$

Here $||\cdot||_{\Omega}$ represents a suitable norm of a vector field (may be the $\mathcal{L}_1$ norm) over the flow domain Ω, and $||\cdot||_{\partial\Omega}$ denotes another norm of a vector field over the boundary of the domain $\partial\Omega$. The

cost function represents the error in the solution over the domain, in the sense of the norm of the residual. It will be shown subsequently that, even with the POD approximation, we can *exactly* enforce the boundary constraint where the parameter $\boldsymbol{\mu}$ typically appears explicitly. The task of finding the steady state solution $\boldsymbol{q}$ for the particular operating condition $\boldsymbol{\mu}$ is thus transformed into an optimization problem for $\boldsymbol{\eta}$.

Note that, a very similar approach is followed in the HOM too. There we are also looking to minimize the residual $\mathscr{R}$ subject to satisfaction of the boundary conditions of the form $\mathscr{B}$. The only difference is that the variables being optimized in HOM are the set of flow variables $\boldsymbol{q}$ at all grid points (a very large number), whereas the corresponding variables in the ROM are the much smaller set of POD coefficients $\boldsymbol{\eta}$. Therein lies the savings.

The description elides many of the implementation details. Over the years, several attempts have been made to arrive at efficient yet accurate models of this type[8–10]. Some of the key conclusions that are applicable to the present problem are:

- In fact, all the snapshots satisfy the no-slip condition at the body surface, and the POD modes are themselves linear combinations of these snapshots. Thus, the predicted flow solution, being again a linear combination of the POD modes, also satisfy the no-slip boundary condition. This means that $\mathscr{B}$ only encodes the far-field boundary conditions, as the wall boundary conditions are automatically satisfied by the flow solution.

5.1 Computation of the cost function

Although the flow solutions in the 'learning database' generally come from high-fidelity RANS, the residual may be computed using a low fidelity Euler solver[8–10]. This is because the lowest-order POD modes, being representative of global behaviour, cannot resolve the near-wall localized viscous/turbulent effects in any case. However, they indirectly account for these effects since they are derived from the RANS results that incorporate the no-slip condition.

We begin by writing the conservative form of the non-dimensional Euler equations for a calorically perfect gas in Cartesian coordinates[16]:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{u}) = 0, \tag{9a}$$

$$\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} u) + \frac{\partial p}{\partial x} = 0, \tag{9b}$$

$$\frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} v) + \frac{\partial p}{\partial y} = 0, \tag{9c}$$

$$\frac{\partial (\rho w)}{\partial t} + \nabla \cdot (\rho \boldsymbol{u} w) + \frac{\partial p}{\partial z} = 0, \tag{9d}$$

$$\frac{\partial}{\partial t}\left(\rho\left(\frac{p}{(\gamma-1)\rho} + \frac{V^2}{2}\right)\right) + \nabla \cdot \left(\rho \boldsymbol{u}\left(\frac{\gamma p}{(\gamma-1)\rho} + \frac{V^2}{2}\right)\right) = 0. \tag{9e}$$

In the above, the $V := \sqrt{u^2 + v^2 + w^2}$. Also, the $\rho^{-1}p/(\gamma-1)$ and $\gamma\rho^{-1}p/(\gamma-1)$ terms in the energy conservation equation may be identified as the specific internal energy and specific enthalpy respectively; both are normalized by c_∞^2 to agree with the non-dimensional kinetic energy term $V^2/2$.

We start evaluating the cost function by integrating the above equations (sans the time derivative terms) over each grid cell. In the integration, we use Gauss' divergence theorem to convert the cell-volume integral to an integral over the surface of the cell. For example, the residuals of the

mass, $x-$momentum and energy equations over the cth cell become

$$\mathscr{R}_\rho^c = \sum_{f=1}^{N_f^c} [\rho \boldsymbol{u} \cdot \hat{\boldsymbol{n}} A]_f^c \,, \quad \mathscr{R}_u^c = \sum_{f=1}^{N_f^c} [(\rho \boldsymbol{u} \cdot \hat{\boldsymbol{n}} u + p\hat{n}_x) A]_f^c \,, \quad \mathscr{R}_e^c = \sum_{f=1}^{N_f^c} \left[\rho \boldsymbol{u} \cdot \hat{\boldsymbol{n}} \left(\frac{\gamma p}{(\gamma-1)\rho} + \frac{V^2}{2} \right) A \right]_f^c ,$$

where $[\cdot]_f^c$ denotes the surface integral over the fth face of the cth cell with area A, and N_f^c is the number of faces of the cth cell. The outward unit normal of a face is $\hat{\boldsymbol{n}}$, with Cartesian components $\hat{n}_x$, $\hat{n}_y$ and $\hat{n}_z$.

It is evident from the above expressions that we need the flow variables on the faces of each cell, say at the face centroids. Moreover, the face-integrated mass flow $[m]_f^c := [\rho \boldsymbol{u} \cdot \hat{\boldsymbol{n}} A]_f^c$ appears in every residual term, and thus must be reconstructed as accurately as possible. One option that is adopted here is to rely on the fact that mass conservation is satisfied in the above form for each steady HOM solution in the database. Since the POD modes (as well as the mean flow field) are a linear combination of the HOM solutions (see eqn. (3)), each POD mode (and the mean flow field) will satisfy mass conservation automatically if ρu, ρv and ρw on the cell faces are chosen as POD variables. Then, mass conservation is automatically satisfied by the reconstructed solution also, as it is a linear combination of the POD modes and the mean flow field (see eqn. (1)). Note that this would not have been the case had ρ, u, v and w been chosen as POD variables, since they combine nonlinearly to give the mass flux. The fifth independent variable chosen for POD is the pressure, as it is already available on the cell faces in the flow solution database. In summary, to facilitate accurate evaluation of the residual, the POD is performed on face-based data alone. Appendix B details how the mix of cell-based and face-based data available in the ANSYS Fluent solution were interpolated to the faces of the grid cells.

Given a candidate set of optimization variables (POD coefficients), we reconstruct the face-based pressure, density and three components of mass flux vector (ρu, ρv and ρw) using the POD eigenfunctions and the mean flow field. The mass flow through a face is calculated from the dot product of the mass flux vector $\rho \boldsymbol{u}$ and the pre-computed face area vector $\hat{\boldsymbol{n}} A$. The components of the velocity vector at the face are determined by dividing the respective components of the mass flow vector by the density. These are used to evaluate the Euler residuals approximately on each cell as

$$\mathscr{R}_\rho^c = \sum_{f=1}^{N_f^c} [m]_f^c \,, \quad \mathscr{R}_u^c \approx \sum_{f=1}^{N_f^c} \left([m]_f^c \, \frac{[\rho u]_f^c}{[\rho]_f^c} + [p]_f^c \, [\hat{n}_x A]_f^c \right), \quad \mathscr{R}_v^c \approx \sum_{f=1}^{N_f^c} \left([m]_f^c \, \frac{[\rho v]_f^c}{[\rho]_f^c} + [p]_f^c \, [\hat{n}_y A]_f^c \right),$$
$$\mathscr{R}_e^c \approx \sum_{f=1}^{N_f^c} [m]_f^c \left(\frac{\gamma \, [p]_f^c}{(\gamma-1) \, [\rho]_f^c} + \frac{([\rho u]_f^c)^2 + ([\rho v]_f^c)^2 + ([\rho w]_f^c)^2}{2([\rho]_f^c)^2} \right). \tag{10}$$

In the above, we approximate the face integral of a flux by the product of mass flux at the face with the relevant scalar evaluated at the face centroid. One can readily deduce the form of $\mathscr{R}_w^c$.

It will be noted that the preceding development guarantees that the residual of the mass conservation equation vanishes on all cells, to the precision mandated by the HOM solver used in the underlying empirical learning database. Thus, the residual needs to be evaluated for the four other equations only. Finally, the cost function is calculated as the 1-norm of the residuals of the four equations, normalized by the total volume of all the cells considered, V:

$$J = \mathscr{R}_u + \mathscr{R}_v + \mathscr{R}_w + \mathscr{R}_e, \qquad \mathscr{R}_q := V^{-1} \sum_{c=1}^{N^c} \left| \mathscr{R}_q^c \right|, \quad q \in \{u, v, w, e\}. \tag{11}$$

The literature on steady ROM aerodynamics[8,10,17] has a discussion on the choice of the p-norm. The 1-norm or even the 1/2-norm has been recommended rather than the more prevalent 2-norm, so as to avoid giving greater weightage to the residuals in regions of the solution domain where the HOM incurred greater errors. We have evaluated the 1/2-norm also, but did not find any significant difference in its outcome compared to the 1-norm.

One of the prominent recommendations from the literature[8–10] is that the residual need not be evaluated over the entire domain. This is because the ROM problem is already severely over-determined, with the residual having to be minimized on many more cells than there are degrees of freedom (the number of POD mode coefficients η's). Thus, critical regions may be determined from prior experience, and the residual may be evaluated in those regions alone. That is, the domain appearing in eqn. (8) may be replaced by a reduced domain Ω_R; this may or may not coincide with the POD domain Ω_P. This reduces the cost involved in the optimization.

5.2 Implementation of boundary conditions

Let us consider generic Dirichlet and Neumann boundary conditions, and analyze how they translate in the POD expansion. For this, we will consider two sub-cases: case A will have the same numerical boundary condition for all the snapshots, whereas case B will allow for different numerical values of the boundary condition in the various snapshots. Case A is exemplified by no-slip (Dirichlet) boundary condition at a wall, and zero pressure gradient (Neumann) boundary condition at an outlet. Case B is exemplified by various freestream velocities (Dirichlet) being imposed on the inlet to simulate different operating conditions.

Constant Dirichlet condition: Suppose a particular variable, say q, is constrained to be $\widehat{q}$ on (a part or the entirety of) the boundary (wall, inlet, outlet, etc.) in all snapshots. Then, the mean flow field will have $q = \widehat{q}$ on that surface, and all the POD modes will have vanishing q thereat. Thus, there is no extra constraint that needs to be imposed on a flow solution composed as a linear combination of the POD modes, plus the mean flow field. Evidently, the POD modes cannot be used to compose the solution if the boundary condition value is changed from $\widehat{q}$.

There is a caveat to the above analysis: it only holds when q is one of the 'primitive' variables whose POD modes are being computed. Consider the example of a specified mass flux boundary condition imposed at a choked nozzle throat. If mass flux is one of the POD variables, then the above statements hold. However, if density and velocity are the POD variables, then the mass flux of the solution is no longer a linear combination of the mass fluxes of the POD modes, and therefore this simplification is rendered invalid.

As an example of particular relevance, the no-slip condition at a wall is automatically satisfied by all solutions, as long as all components of velocity or mass flux are expanded in a linear combination of their POD modes (and the mean field). As another example, if the freestream pressure and/or temperature are ambient values for all supplied snapshots of missile aerodynamics (as in our learning database), then there is no need to impose these explicitly in the POD ROM.

Remark 6 *The latter example of implementation of a constant but non-zero Dirichlet boundary condition demonstrates a benefit of subtraction of the snapshot mean in the POD. For, otherwise the constant freestream pressure/density (for example) would have to be reconstructed from the POD modes, which would have to be implemented explicitly.*

Constant Neumann condition: Following the above arguments, this boundary condition also does not give rise to any constraint on the POD modes, since all of them will have zero gradient at

the particular boundary, and the mean flow field will have the specified (possibly non-zero) constant gradient at the boundary. For example, if outlet pressure gradient is constrained to be zero in flow simulations, then this is implicitly satisfied by all POD ROM solutions also.

Varying Dirichlet conditions: This case will be discussed with a relevant example – the solution of the POD ROM for a particular operating condition. The operating condition of a missile is usually specified in terms of its freestream Mach number, angle of attack, and roll angle. However, the 'primitive' variables in the POD are the three components of mass flux vector, along with density and pressure. Thus, recalling the caveat mentioned in the case of imposing a *constant* Dirichlet condition, we must first determine the freestream (i.e., the inlet boundary) mass flux vector of a snapshot, denoted here by $\widehat{\rho \boldsymbol{u}}$. Since the normalized density is unity at the inlet for all snapshots, this is actually indistinguishable from $\widehat{\boldsymbol{u}}$.

Suppose that a flow variable $\rho \boldsymbol{u}$ takes on boundary values of $\{\widehat{\rho \boldsymbol{u}}_s\}_{s=1}^S$ respectively in the S snapshots constituting the database underlying POD. Then, the mean flow field will have the average freestream mass flux $\overline{\widehat{\rho \boldsymbol{u}}} = (1/S) \sum_{s=1}^S \widehat{\rho \boldsymbol{u}}_s$ at all points of the inlet boundary. We next determine the fluctuation boundary values of the snapshots $\{\widehat{\rho \boldsymbol{u}}'_s\}_{s=1}^S$, where $\widehat{\rho \boldsymbol{u}}'_s := \widehat{\rho \boldsymbol{u}}_s - \overline{\widehat{\rho \boldsymbol{u}}}$. Then, referring to eqn. (3), the freestream mass flux corresponding to the nth POD mode can be directly found as $\widehat{\widetilde{\rho \boldsymbol{u}}}^n = \sum_{s=1}^S \zeta_s^n \widehat{\rho \boldsymbol{u}}'_s$, instead of actually evaluating the POD mode.

Finally, the freestream boundary condition constraint for a particular operating condition $\boldsymbol{\mu}_0$ (given by $\widehat{\boldsymbol{u}}_0$) requires the simultaneous satisfaction of the following set of linear equations

$$\sum_{n=1}^{N} \eta^n(\boldsymbol{\mu}_0) \widehat{\widetilde{\rho \boldsymbol{u}}}^n = \widehat{\rho \boldsymbol{u}}_0 - \overline{\widehat{\rho \boldsymbol{u}}}.$$

Let there be $\widehat{N}$ such equality constraints. Presumably, $\widehat{N} < N$, so that the above is an underdetermined system. To implement such a simple linear equality constraint in the optimization problem, we can reduce the number of optimization variables from N to $N - \widehat{N}$. Then, whenever the full variable set is required (as in the actual evaluation of the cost function), we can 'lift' the above reduced set by solving for the removed ones using the above constraint equations.

For specificity, the only equality constraints in our problem is the specification of the three freestream mass flux components $\widehat{\rho u}$, $\widehat{\rho v}$ and $\widehat{\rho w}$, so that $\widehat{N} = 3$. Also, let the set of $N - \widehat{N}$ indices of the 'kept' optimization variables (η's) be I_K and the indices of the remaining 'removed' ones be $I_R \equiv \{n_1, n_2, n_3\}$. Then, lifting involves the solution of the following linear system with a given reduced set $\{\eta^n\}_{n \in I_K}$:

$$\begin{bmatrix} \widehat{\widetilde{\rho u}}^{n_1} & \widehat{\widetilde{\rho u}}^{n_2} & \widehat{\widetilde{\rho u}}^{n_3} \\ \widehat{\widetilde{\rho v}}^{n_1} & \widehat{\widetilde{\rho v}}^{n_2} & \widehat{\widetilde{\rho v}}^{n_3} \\ \widehat{\widetilde{\rho w}}^{n_1} & \widehat{\widetilde{\rho w}}^{n_2} & \widehat{\widetilde{\rho w}}^{n_3} \end{bmatrix} \begin{bmatrix} \eta^{n_1} \\ \eta^{n_2} \\ \eta^{n_3} \end{bmatrix} = \begin{bmatrix} \widehat{\rho u}_0 - \overline{\widehat{\rho u}} - \sum_{n \in I_K} \eta^n \widehat{\widetilde{\rho u}}^n \\ \widehat{\rho v}_0 - \overline{\widehat{\rho v}} - \sum_{n \in I_K} \eta^n \widehat{\widetilde{\rho v}}^n \\ \widehat{\rho w}_0 - \overline{\widehat{\rho w}} - \sum_{n \in I_K} \eta^n \widehat{\widetilde{\rho w}}^n \end{bmatrix}. \tag{12}$$

Evidently, I_R must be chosen so as to have a well-conditioned matrix on the left hand side above. Also, all such equality constraints should be evaluated simultaneously, and not sequentially. In the above, we have omitted $\boldsymbol{\mu}_0$ for notational convenience.

Varying Neumann conditions: Such boundary conditions, if present, can also be addressed in a manner similar to the preceding varying Dirichlet conditions. However, in the reduction-lifting setting, all such equality constraints, be they Dirichlet or Neumann, should be implemented simultaneously.

5.3 Evaluation of Aerodynamic Coefficients

The ultimate aim of the POD ROM developed here is the estimation of the drag, normal-force, and side-force coefficients along with the three moment coefficients of the missile for a given operating condition (that is not in the learning database). One way of achieving this is to perform the requisite force and moment integrations over the missile surface using the snapshot reconstructed from the POD ROM solution. However, there is a much more efficient procedure that leverages the quasi-linearity of the setting.

The total surface force vector on an immersed body is given by

$$\boldsymbol{F} = \int_A \left(-p\hat{\boldsymbol{n}} + \mu\boldsymbol{\omega} \times \hat{\boldsymbol{n}}\right) \mathrm{d}A = \int_A \left(-p\hat{\boldsymbol{n}} + \mu\nabla \times \boldsymbol{u} \times \hat{\boldsymbol{n}}\right) \mathrm{d}A. \tag{13}$$

Here, $\hat{\boldsymbol{n}}$ is the local unit normal on the surface, p is the surface pressure, $\boldsymbol{\omega}$ is the surface vorticity vector, μ is the dynamic viscosity (a function of temperature $T = \gamma p/\rho$), and A is the surface area of the missile. The first term is of course accounting for pressure-based forces, whereas the second term gives the contribution of wall shear.

With the prevailing choice of the POD variables, the surface force vector for a reconstructed solution is

$$\boldsymbol{F} = \int_A \left[-\left(\overline{p} + \sum \eta^n \widetilde{p}^n\right) \hat{\boldsymbol{n}} + \mu \left(\gamma \frac{\overline{p} + \sum \eta^n \widetilde{p}^n}{\overline{\rho} + \sum \eta^n \widetilde{\rho}^n} \right) \nabla \times \left(\frac{\overline{\rho \boldsymbol{u}} + \sum \eta^n \widetilde{\rho \boldsymbol{u}}^n}{\overline{\rho} + \sum \eta^n \widetilde{\rho}^n} \right) \times \hat{\boldsymbol{n}} \right] \mathrm{d}A. \tag{14}$$

It is evident that, whereas the force due to pressure is a linear function of the POD coefficients (the optimization variables), the wall shear force is a nonlinear function due to (a) the temperature-dependence of viscosity, and (b) the choice of mass flux instead of velocity as the POD variable.

For the subsonic database being considered, the wall temperature is not expected to vary significantly. This not only means that μ may be assumed to be constant at the wall, but it also implies that pressure and density should be directly proportional. Now, the wall-normal pressure gradient is known to approximately vanish in a boundary layer, so that the wall-normal density gradient should also not be strong. But it is this gradient that appears in the wall shear stress component of the surface force. Given that the shear contribution to overall forces is small (except for the axial force component), we assume that total forces and moments (not just the pressure components) are linear functions of the POD coefficients. The advantage is that we obtain a procedure for evaluating the aerodynamic coefficients of a reconstructed solution without having to do the actual reconstruction and the surface integration.

Let the surface force for the sth snapshot in the original database be $\boldsymbol{F}_s$. This is known from the aerodynamic force coefficients (calculated by the CFD software, say) multiplied by $\rho_\infty U_\infty^2 A_{ref}/2$. Here ρ_∞ and U_∞ are the freestream density and speed, respectively, and A_{ref} is the reference area (usually square of the missile diameter). Note that, if the database is computed for various missile speeds and/or flight altitudes, the above multiplier will be different for each snapshot.

Let the mean surface force across the database be $\overline{\boldsymbol{F}}$, and the consequent fluctuation forces be $\{\boldsymbol{F}'_s\}_{s=1}^S$. Then, with the above-mentioned approximation of linearity, one obtains the surface force for a reconstructed solution as

$$\boldsymbol{F} = \overline{\boldsymbol{F}} + \sum_{n=1}^{N} \eta^n \widetilde{\boldsymbol{F}}^n, \qquad \text{with } \widetilde{\boldsymbol{F}}^n := \sum_{s=1}^{S} \zeta_s^n \boldsymbol{F}'_s.$$

Evidently, $\widetilde{\boldsymbol{F}}^n$ is the aerodynamic force corresponding to the nth POD mode. The aerodynamic moment vector on the missile can also be calculated from an analogous expression.

Figure 2: Reduced annular domain (green) around the missile (red) shown in a longitudinal section. The minimum distance of the domain cells' centroids from the missile surface is $0.2D$; the maximum distance is $2D$.

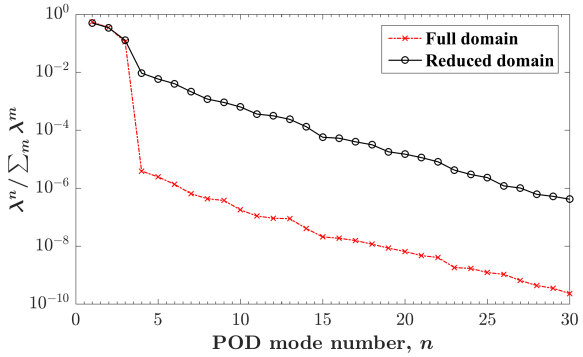


Figure 3: POD eigenspectra from two calculations – one involving the full CFD domain, and the other using data in the reduced domain shown in fig. 2.

It must be noted that the above forces and moments are necessarily obtained in a body-fitted coordinate system (that which remains anchored in the consistent grid for all snapshots). If one desires the aerodynamic coefficients in the total angle of attack reference frame, then the appropriate transformation has to be made separately. Evidently, the force and moment data for the snapshots must also be presented in the body frame to the above algorithm.

6 Results and discussion

6.1 POD results

6.1.1 POD modes

Proper orthogonal decomposition has two main ingredients – the selection of the snapshots is the obvious one; the definition of the underlying inner product is the other choice for the designer. As pointed out in the literature[10], the domain of integration Ω_P in the inner product can be a limited sub-domain in the vicinity of the missile. It should not include the very near-surface region, since the local flow structures thereat will necessarily be filtered out by the POD. The domain should also not be too far from the missile surface; otherwise, it will be irrelevant for predicting the forces and moments on the missile.

Two separate sets of POD (and ROM) calculations were performed with the set of 32 learning snapshots. The integration domain in the first POD analysis was the full flow domain which is a cube of sides approximately 160 times the missile diameter D. In the second case, the integration domain was an annular region encasing the missile (see fig. 2 and also fig. 4). It consists of the cells of the unstructured grid whose centroids were more than $0.2D$ but less than $2D$ away from any point on the surface of the missile. The reason for this particular choice will become clearer subsequently. For now we just note that the number of grid cells is reduced from about 10 million down to about 300,000 with this choice, thereby reducing the processing time by a factor of 30. Here we will evaluate if there is a penalty to be paid for this benefit.

Figure 3 presents the POD eigenspectra for the two cases. Each POD eigenvalue is divided by the sum of all the eigenvalues calculated. It appears that the first three eigenvalues are many orders-of-magnitude larger than all the others. Moreover, the spectral shape made up of all the other eigenvalues is very similar with both definitions of the underlying inner product. The major difference in the two eigenspectra is the quantum of decrease of eigenvalue between the third and

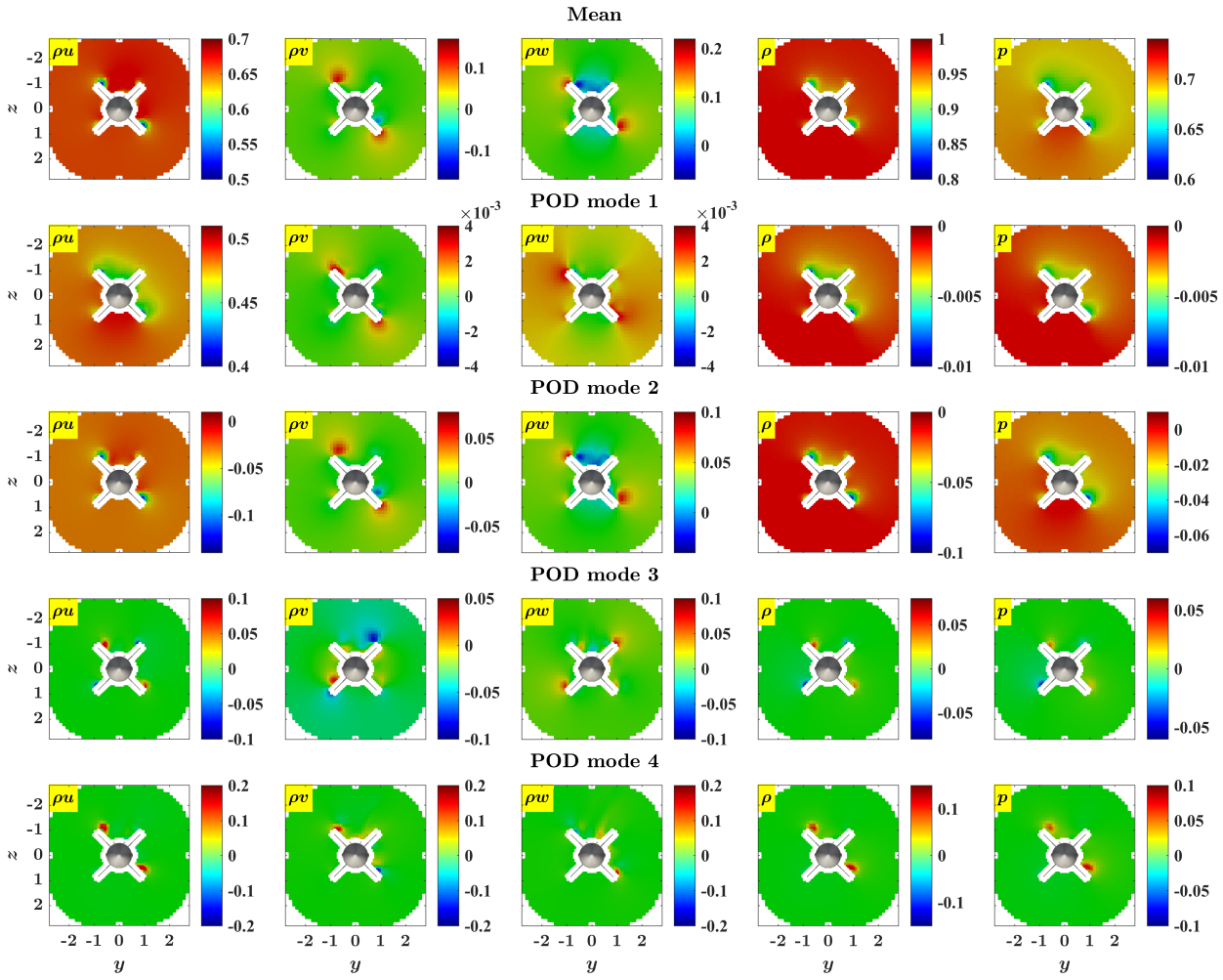


Figure 4: The five components of the mean flow field and of the first four POD eigenfunctions, at a section through the main fins of the missile.

fourth POD modes. These observations on the eigenspectra can be explained with reference to fig. 4 that presents the mean flow field and the first few POD eigenfunctions for the reduced-domain POD.

The $\overline{\rho u}$ field appears to be around 0.7 over the entire domain, except very close to the missile. This is understandable since half the snapshots were at Mach 0.6 and the other half were at Mach 0.8. Note that the mean velocity at the missile surface is zero in every snapshot, so that the near-missile mean velocity should indeed be smaller than the mean freestream velocity. Also, the above Mach numbers correspond to the velocity magnitude, which is slightly larger than the mean x-component being portrayed in the plot. The learning database has snapshots with roll angles of 45, 22.5 and 0°, that have increasing y-components of freestream velocity starting from zero. This explain the slight positive $\overline{\rho v}$ field, with higher values at the tips of the two wings around which the flow accelerates. Also, the choices of angles of attack and the roll angles in the learning database means that the z-component of the freestream flow is either zero (in case $\alpha = 0$) or positive in the snapshots. This is reflected in the overall positive $\overline{\rho w}$ field, with higher values at the same wing tips as in the $\overline{\rho v}$ case, and lower values where there might be flow retardation or even separation. The $\overline{\rho u}$ field is also reflected in the mean density and pressure fields; they relax to unity and $\gamma^{-1} = 0.714$, respectively, far away from the missile. Notice that towards the outer edge of the reduced annular

domain depicted, the mean flow field tends to be uniform.

The first POD mode is dominated by the ρu field, which is two orders-of-magnitude higher than the other components of the mass flux. This mode captures the major variation in the database between the Mach 0.6 and Mach 0.8 snapshots, which is an increased freestream u. The precise magnitude of around 0.5 away from the missile is a function of the normalization of the POD eigenfunction. For example, it is much smaller if the full HOM domain is used in the inner product, since the norm must still be unity. Thus, one should only focus on the relative values of the POD eigenfunctions. Note that the density and pressure components of the first POD mode faithfully track the ρu component, but they tend to zero far from the missile since all snapshots have the same freestream density and pressure that are already accounted for in the mean flow field.

The second POD mode is dominated by the ρw field; this is to be expected since the next major variation in the database (after the one captured by the first POD mode) is that due to different angles of attack from 0 to 20°. The third POD mode has non-zero ρv away from the missile; it captures the remaining distinctive variation of the roll angles in the database. The fourth POD mode has non-trivial values only towards the inner edge of the annular domain, and this trend persists in all higher-order modes that are not shown. All through these four modes, the behaviour of the ρu, density and pressure components appear to be very similar.

A clear explanation of the observations in fig. 3 is obtained from the results and discussion of fig. 4. The full domain differs from the reduced annular domain in its inclusion of the very near-surface region and the outer region. However, the latter sub-region is much larger in volume (and hence weightage in the inner product) than the former. Moreover, the flow field is more or less uniform in the outer region omitted in the reduced domain. This explains why the relative magnitudes of the first three POD eigenvalues (that capture the far-surface flow variations in the database) are the same with the two domain choices. It also explains why the remaining parts of the POD eigenspectra are similar in the two cases, but only of much smaller relative magnitude in the case of the full domain. Basically, both choices display similar flow features, and both can only capture the very near-surface flow phenomena in very high order of modes.

6.1.2 POD-reconstruction of aerodynamic coefficients of cases in learning database

The pressure component of the aerodynamic forces and moments on a missile is not significantly affected by flow features that are extremely close to the surface. The friction component is apparently affected by flow in the boundary layer; however, this too is driven by the outer flow, and the boundary layer physics are also included in the snapshots already. This suggests that POD, although it is very inefficient at capturing the very near-surface flow features, may still be useful for predicting the aerodynamic coefficients of a missile. This will indeed be validated by the results presented subsequently.

Before using the POD basis in predicting new cases, let us evaluate how effective they are in reproducing the aerodynamic coefficients of the cases in the learning database. In particular, fig. 5 presents the error in reconstructing the aerodynamic force and moment coefficients of all the learning cases using the first 25 POD modes (out of 32) calculated on the reduced domain. The shortcut method described in Section 5.3 is employed. The maximum magnitudes of the various coefficients achieved across the database are used to normalize the respective errors; these are also noted in the parentheses. Each subplot presents the error vs. angle of attack of the snapshots on six curves, one each for each pair of Mach number and roll angle.

In general, the maximum error occurs for lower angles of attack. This may be a consequence of the very local flow features that occur at lower α's, which are only captured by the very highest order POD modes. The axial force coefficient C_A presents the lowest fractional error. This is

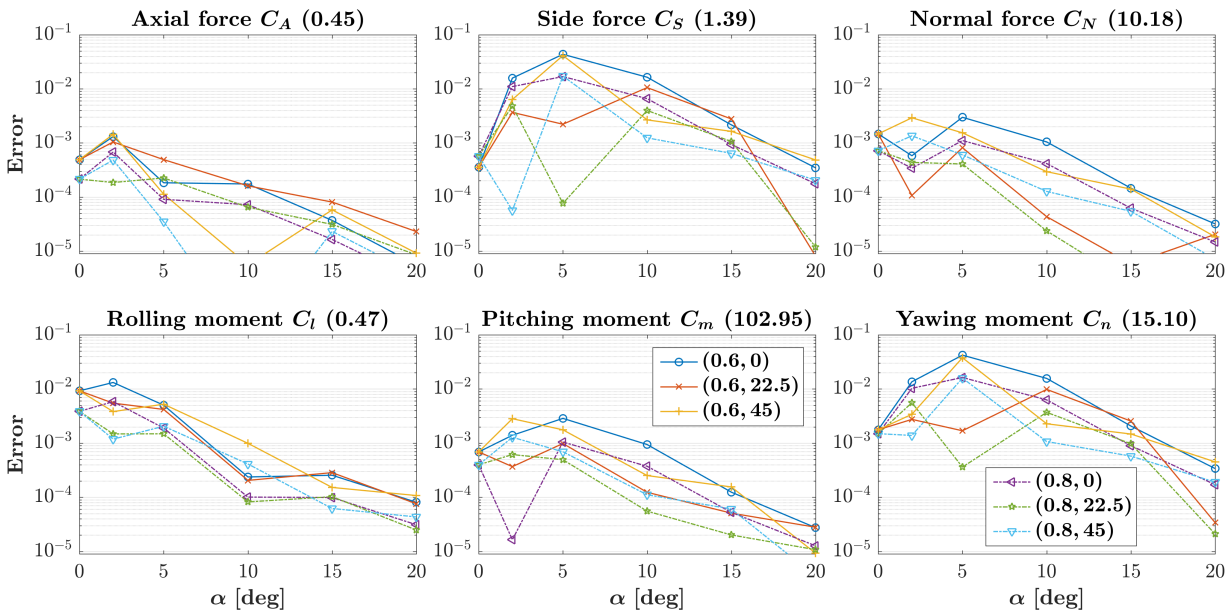


Figure 5: Error in the aerodynamic coefficients for all learning database cases, expressed as a fraction of their maximum magnitudes found in the database (indicated in parentheses in the respective titles) when the first 25 POD modes are used to reconstruct the flow field; the POD inner product is evaluated on the reduced domain depicted in fig. 2. The abscissae indicate the angle of attack, and the legend gives the Mach number and roll angle of the cases.

interesting since it is dominated by shear stresses, which are actually nonlinear functions of the POD coefficients (see eqn. (14)), but are approximated as linear functions here. The normal force and pitching moment coefficients present very low fractional errors. Their cross-sectional counterparts – viz. the side force and yawing moment coefficients – present the highest errors (although these are still less than 5%); this is probably an artifact of the order-of-magnitude smaller maximal value used in the normalization of these errors. Referring to fig. 1, for the slender missile body, the normal force is primarily responsible for the pitching moment, and the yawing moment is mostly tied to the side force. This explains why the two pairs appear to go hand in hand across all the cases in fig. 5. The rolling moment error is intermediate in magnitude, but is usually still less than 1%.

The above error analysis was performed for the case of the 25-mode POD basis on the reduced annular domain in the vicinity of the missile surface; we now extend it to all possible numbers of retained POD modes. For this, we first compact the information presented for any single aerodynamic coefficient in fig. 5 into a single number, viz. the simple average of all the fractional errors. The resulting six numbers, one for each aerodynamic coefficient, are now reported versus the number of POD modes retained in the reconstruction in fig. 6a. We show results for both the choices of POD domains – the full one as well as the reduced annular one.

Overall, it is observed that the reconstruction error indeed reduces with increasing N; however, the decay is not as rapid as observed for the POD eigenvalues in fig. 3. This is not surprising because, although POD guarantees the best reconstruction (with a given number of basis modes) of the flow field *over the entire domain*, it need not be efficient in reproducing the flow field on a boundary like the missile surface. However, it is very encouraging to find that POD still delivers an acceptable reconstruction of all the surface-integrated aerodynamic coefficients, with less than

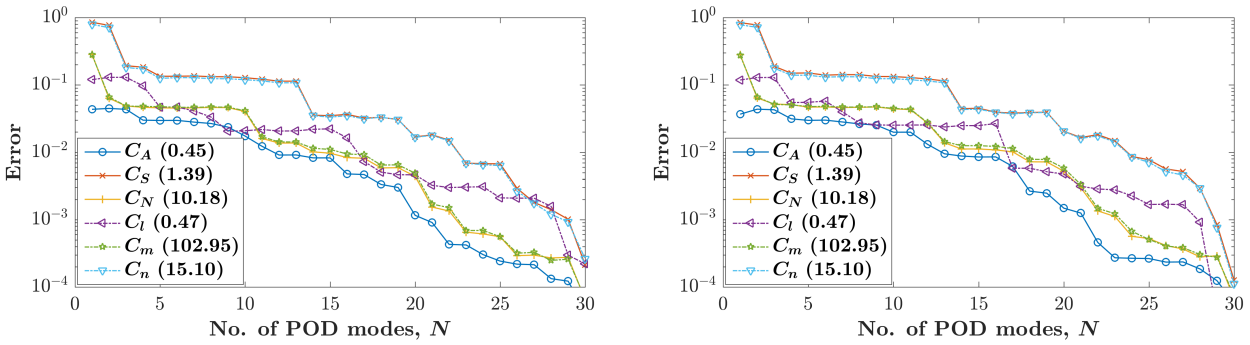


Figure 6: Average fractional error in the various aerodynamic coefficients computed from flow fields reconstructed from increasing number of retained POD modes N, with the inner product evaluated on (a) the reduced domain depicted in fig. 2, and alternatively (b) the full CFD domain. The coefficients's errors are a fraction of their respective maximum magnitudes found in the training database, as indicated in parentheses in the legend.

2% average error for $N = 20$ and less than 1% average error for $N = 25$. Also, the errors in the aerodynamic coefficients decrease almost monotonically, and they essentially vanish when $N = 31$ (not shown). The significance of the latter is realized by recalling that, with 32 snapshots in the mean-subtracted database, the highest POD mode order with non-zero eigenvalue is $n = 31$. That the reconstruction error vanishes when all non-trivial POD modes are used is not surprising in itself, other than the fact that the nonlinear shear component of the aerodynamic forces and moments are approximated as linear here (see eqn. (14)). Thus, this observation actually validates the assumption of linearity.

The overall trends observed in fig. 5 are borne out in fig. 6a too. For example, the axial force consistently displays the least error, followed by the almost identical errors in normal force and the pitching moment. Usually, the next in terms of error magnitude is the rolling moment. The side force and yawing moment show the most error, and their magnitudes are essentially indistinguishable. This reconstruction error analysis with POD modes directly gives us a lower bound for the later prediction errors with the POD-ROM. In particular, we expect that at least 25 POD modes should be used in the ROM to attain acceptable results.

We end this discussion on error analysis by noting that the results from the full-domain POD shown in fig. 6b are very similar to the reduced-domain POD results of fig. 6a being discussed above. This further corroborates the preservation of information in the cost-effective domain reduction process that was first noted in the context of their POD eigenspectra presented in fig. 3.

Given that the significantly higher cost associated with the full-domain POD does not result in any appreciable benefit compared to the reduced-domain POD, we will implicitly use the latter in all the subsequent results presented. Indeed, all the following results were also evaluated with the full-domain POD, with no significant effect on the outcome.

6.1.3 Euler residuals computed from POD reconstruction of cases in learning database

We will subsequently use the Euler equation residuals defined in eqns. (10) and (11) to determine the optimal ROM solution. Figure 7 presents the magnitudes of the residuals of the three momentum equations and the energy equation; recall that the continuity equation is satisfied to machine precision in the present context. Specifically, the 1-norm of the residuals are calculated from the

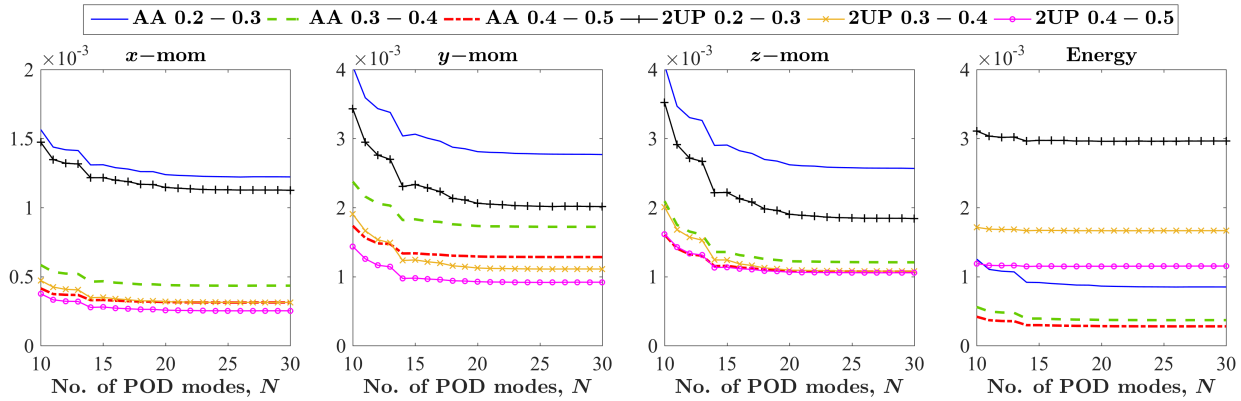


Figure 7: Residuals of the three momentum equations and the energy equation computed from flow solutions reconstructed with increasing number of POD modes derived on the reduced domain shown in fig. 2, averaged over the learning database. The legend indicates the type of interpolation used ('AA' = arithmetic average, '2UP' = second-order upwind), and the annular region's extent (in terms of fractions of D) on which the residual is evaluated.

flow fields of the learning database cases reconstructed with increasing number of POD modes N, and their average over the database are shown. One observes that the residuals asymptote uniformly with N.

This analysis affords an assessment of some of the options available in the POD-ROM approach. Firstly, the calculated residuals depend on the specific approach used to interpolate the cell-center HOM data to the faces, prior to the POD step. Figure 7 presents results from the arithmetic average ('AA') and the second-order upwind ('2UP') methods (see also appendix B). We find that the former delivers significantly lower energy residuals, but somewhat higher momentum residuals. Given the cost of the second-order upwind scheme, this result suggests that the much simpler arithmetic-average method may give acceptable predictions. We must note here that the computational cost of this choice is only borne once upfront, and does not recur in the ROM optimization step. All the POD results presented previously (i.e., in fig. 3–6) use the arithmetic average method in the pre-processing; although not shown here, essentially identical graphs are obtained with the second-order upwind method also.

The second option evaluated here is the choice of the spatial domain on which the residual is calculated. The domain has already been reduced once prior to the POD; here we consider even smaller sub-domains of this. Specifically, we assess three annular regions that are successively farther removed from the missile surface. Each annular region is of width equal to one-tenth of the missile diameter D. They start from 0.2, 0.3 and $0.4D$ away from the missile surface, respectively. As expected, the Euler residuals decrease uniformly with increasing distance of the sub-domain of evaluation from the missile surface. Even though the nearest sub-domain results in two to three times higher residual values, residuals calculated on this sub-domain will be later shown to be the most suitable for the ROM.

A subtle but important detail that is not discussed above is the geometry of grid cells chosen for evaluating the residuals. In a three-dimensional unstructured mesh, some cell faces (other than triangular ones) may be non-planar due to errors in meshing. Such faces do not have a unique normal direction, and the fluxes calculated on them by the HOM solver presumably depend on the particular method adopted to have a consistent normal. Due to the inherent redundancy in our ROM optimization problem, we chose to disregard all such cells with non-planar faces in the

1-norm of the residuals presented in eqn. (11).

It will be recalled that the unstructured mesh underlying our missile database is hexahedral. Each quadrilateral face of a hexahedron was sub-divided into two triangles, and it was flagged if the two unit normal vectors failed to match within 1e-7. The cell was retained in the residual calculation only if all its faces were planar. This reduced the cell count from $\sim$ 70K to $\sim$ 3600 in the annular zone from 0.2 to $0.3D$, from $\sim$ 37K to $\sim$ 1400 in the next sub-domain, and from $\sim$ 23K to $\sim$ 1100 in the 0.4 to $0.5D$ annular zone. Of course, the resulting reduced cell counts were still two orders-of-magnitude larger than the degrees of freedom (number of retained POD modes) in the ROM optimization. Although not demonstrated here, the residual values were much larger, and the subsequent ROM results were much worse, when all the cells in these zones were used indiscriminately in the residual evaluation.

6.2 ROM results

The POD-ROM approach is validated in two ways: (a) by reusing the learning database itself in a manner to be discussed in Section 6.2.1, and (b) by testing it on new cases that are not available in the learning database (discussed in Section 6.2.2).

6.2.1 Validation of the ROM on the learning database itself

The POD is guaranteed to be the most efficient at reconstructing the flow field of a snapshot in the learning database. However, this is only in an average sense, and also in the sense of reconstruction accuracy evaluated over the domain of definition of the inner product. On the other hand, we wish to use the POD basis in the optimization ROM paradigm for predicting the forces and moments integrated only on the missile surface, and that too for a case that is not included in the learning database. Before validating the POD-ROM approach on new cases, we can assess it by reusing the learning database itself. This is very useful in ascertaining the optimal values of some of the parameters of the approach itself for the specific problem at hand, and incurs no extra HOM simulation cost.

Snapshots are available for S (= 32) operating conditions in the learning database. We form S separate POD bases, each omitting one of the original snapshots. Then, each of these S POD basis sets is used in the POD-ROM approach to predict the aerodynamic coefficients of the respective omitted snapshot only. The full learning database of course has the truth values of these aerodynamic coefficients, so that the predictive effectiveness of the model can be assessed readily.

Figure 8 presents the results of this validation exercise. As in the results shown in fig. 6, we assess the performance of the approach by averaging the fractional error across all the S predictions, each for a snapshot that was omitted in calculating the respective POD basis. This method allows us to evaluate three parameter choices: (a) the effect of using a reduced domain in the POD instead of the full HOM domain, (b) the appropriate number of POD modes to retain (limited to $S-2$, since each basis is constructed from $S-1$ snapshots and the highest-order POD eigenvalue is zero when the snapshot mean is subtracted prior to POD), and (c) the choice of the annular region surrounding the missile on which to evaluate the Euler residual in the ROM. Each of these options have been assessed in Section 6.1, but these were only partial assays since the ROM was not exercised.

The prediction error decreases almost monotonically with increasing number of retained POD modes N, but saturates beyond $N \approx 25$. While the initial improvement is expected from the discussion of fig. 6, the final saturation behaviour may not have been anticipated. However, there is an obvious difference in the two scenarios that explains this. Earlier we were looking at the

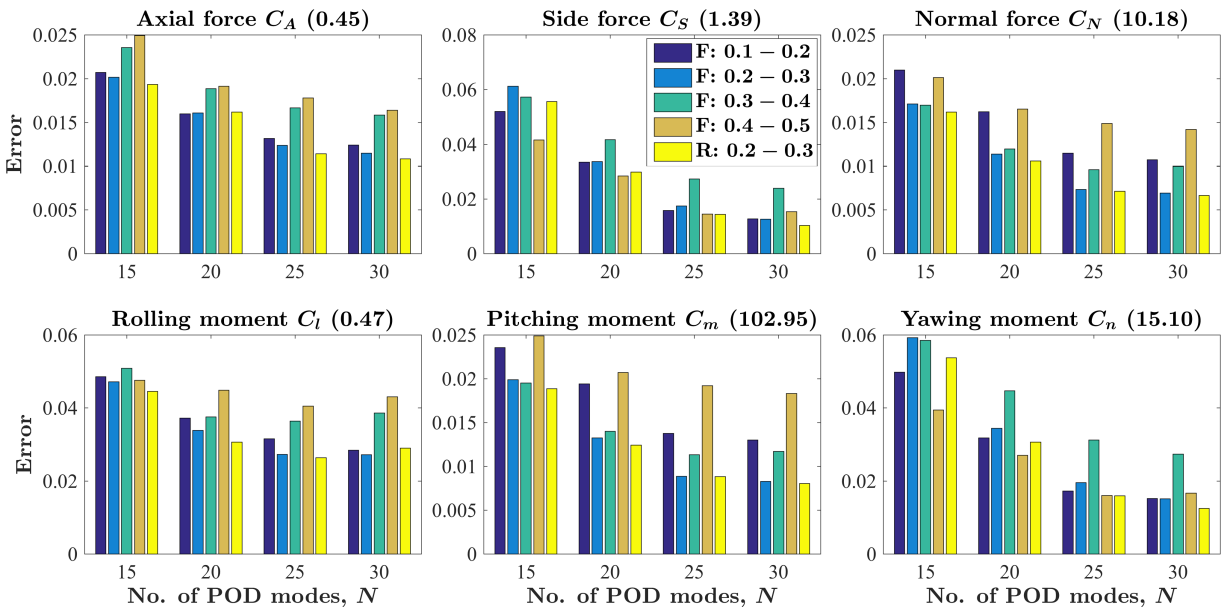


Figure 8: Average fractional errors in predicting the aerodynamic coefficients from the ROM in the 'sans one' evaluation. The POD inner product is evaluated either on the full HOM domain ('F' in the legend), or on the reduced domain pictured in fig. 2 ('R' in the legend). The legend indicates the annular region's extent (in terms of fractions of the missile diameter D) on which the residual is evaluated. The subplot titles follow the scheme of fig. 5.

vanishing of the error over the learning database when the POD basis spanned the same database. On the other hand, at present we are assessing the effectiveness of the POD basis in reconstructing the aerodynamic coefficients of a case that is outside its span. Thus, the present observation is in line with the idea of over-fitting – using fewer degrees of freedom generally results in more robust fits of data, beyond a certain optimal number of degrees of freedom.

There appears to be an optimal offset of the annular region on which the ROM residual is evaluated; overall, the most effective choice appears to be the region between $0.2D$ and $0.3D$ from the missile surface. This trend is not followed in the case of prediction of the side force and yawing moment coefficients using the smaller values of N. The probable reason for the overall trend was surmised in Section 6.1.1. Namely, Euler residuals evaluated on viscous-dominated regions close to the surface are a poor indicator of the actual accuracy of the solution; on the other hand, regions too far from the missile have diminishing correlation with the surface forces and moments.

In fact, it is based on the above result that the POD itself was performed on the snapshots abstracted to the reduced domain extending between $0.2D$ and $2.0D$ off of the missile surface, as pictured in fig. 2. Indeed, fig. 8 demonstrates that the subsequent ROM delivers almost identical results as the case where the POD is performed on the full HOM domain, as long as the residual is evaluated consistently on the $0.2D - 0.3D$ annulus. This equivalence is particularly advantageous since usage of the snapshots abstracted on the smaller domain drastically reduces the cost and time required in (a) calculating the POD modes, and (b) more importantly, performing the subsequent optimization in the ROM.

The choice of the outer extent (i.e., $2.0D$) of the reduced domain for POD was not optimized further. Since the grid coarsens rapidly away from the missile surface, this choice does not materially affect the computational efficiency of the POD-ROM.

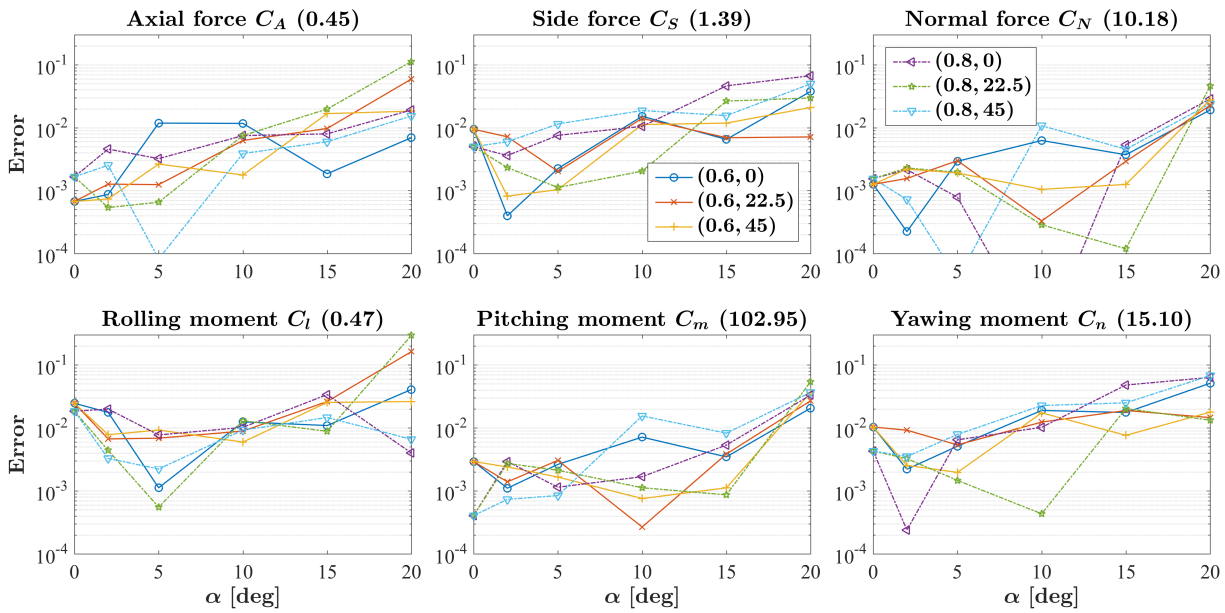


Figure 9: Fractional error in the aerodynamic coefficients when the first 25 POD modes calculated from the annular region shown in fig. 2 are used in the ROM whose Euler residual is evaluated in the narrower annular region between $0.2D$ and $0.3D$. The scheme of presentation follows fig. 5.

It may appear that the imposition of the far-field boundary condition is impossible if the POD is performed on snapshots abstracted on a near-surface domain. This apparent paradox is resolved by recalling that the freestream boundary condition is actually evaluated directly using the metadata of the snapshots (i.e., Mach number, angle of attack and roll angle) and the weight factors of the snapshots towards the various POD modes (see eqn. (12)). Basically, we are using an inner product whose induced norm is positive semi-definite (i.e., defined over a part of the domain), so that the POD modes are notionally defined over the entire HOM domain that extends to the far-field boundary.

Figure 9 presents a detailed view of the aerodynamic coefficient prediction error with the most pertinent model, viz. using the first 25 POD modes computed on the reduced $0.2D - 2.0D$ domain, and evaluating the Euler residual in the ROM on the $0.2D - 0.3D$ annular domain. One finds fairly reliable prediction in most of the cases with fractional errors limited to 2%, but the errors do increase significantly at the highest angles of attack – viz. $\alpha = 20°$, and to some extent $\alpha = 15°$. It is to be noted that cases with $\alpha = 20°$ represent an extrapolation of the learning database. Also, cases with $\alpha \geq 15°$ have massive flow separation that makes the aerodynamic coefficient vs. angle-of-attack curve nonlinear. Given these challenges, it is quite encouraging to note that even in these extreme conditions, the predictions are accurate to about 10%, barring some rolling moment estimates at $\alpha = 20°$. Of course, a direct juxtaposition with the POD interpolation results in fig. 5 shows these ROM results in poor light, but that comparison is not a fair one as those did not have any predictive value.

Table 2 gives an idea of the computational expense and time consumed for the ROM runs. The calculations were performed on a single processor in serial mode. The expense is primarily a function of the number of POD modes included in the optimization, with the time taken being at most half a minute for $N = 30$. This run time for the ROM is almost negligible compared to an HOM calculation that is performed on a grid with approximately 10 million cells.

No. of POD modes, N	**15**	**20**	**25**	**30**
Run time [seconds]	4.9 ± 0.9	9.0 ± 1.6	15.3 ± 2.7	23.0 ± 4.0
No. of iterations	30.2 ± 5.4	40.1 ± 7.1	50.8 ± 9.0	56.8 ± 10.0
No. of function evaluations	497.3 ± 88.6	856.2 ± 151.4	1323.3 ± 233.8	1763.1 ± 310.9

Table 2: Computational expense and timing of optimization ROM runs, averaged over all model evaluations. The variability is indicated in terms of the standard deviation.

Case no.	**1**	**2**	**3**
Mach number, M	0.6	0.8	0.7
Roll angle, ϕ (degrees)	11.25	11.25	22.5
Angle of attack, α (degrees)	10	10	10

Table 3: Validation cases.

6.2.2 Validation of the ROM on new cases

The validity of the ROM can be tested on snapshots involving other combinations of these parameters, as long as they result in shock-free flow. Table 3 presents the operating condition details of the validation cases employed in the present work. It will be observed that the cases differ in one condition from nearby points in the learning database matrix. Owing to time constraints, an HOM case involving variation in the angle of attack could not be generated.

Table 4 presents the various models that are evaluated in the validation tests with each of the cases described above. Two main model parameters are evaluated – (a) whether the POD inner product is calculated on the full HOM domain or on the reduced domain extending between $0.2D$ and $2.0D$ from the missile surface, and (b) the choice of the annular region for evaluating the ROM cost function.

The result of the validation tests is presented in fig. 10. Overall, we note that all of the models yield less than 4% error across all the cases evaluated. The trends observed for the various models in Section 6.2.1 are replicated here to a significant extent. In particular, the model 'E' performs best consistently in predicting the normal force and the pitch and roll moments, with maximum error of 0.7% only. In case of the other aerodynamic coefficients, model 'E' is not the best choice in all cases; however, the maximum error incurred in using it is 2.5%, which is quite reasonable.

6.3 Comparison of the ROM results with those from linear interpolation

A computationally inexpensive alternative to the optimization ROM is the interpolation approach based on POD[1] discussed in Section 2; here we assess how it performs vis-à-vis the more expensive POD-ROM method. Recall that this method avoids a direct interpolation of the aerodynamic coefficients. Instead, the first N POD coefficients of the empirical database are interpolated to estimate those corresponding to a novel case, followed by evaluation of the aerodynamic coefficients therefrom using the approach described in Section 5.3.

There are various interpolation options available, but we do not set out to weigh their relative benefits. Instead, we use the *linear* interpolation option of the `griddata` function of Scipy[18]. The

Model no.	A	B	C	D	E	F	G
POD domain	Full	Full	Full	Full	Reduced	Reduced	Reduced
ROM domain	$0.1 - 0.2$	$0.2 - 0.3$	$0.3 - 0.4$	$0.4 - 0.5$	$0.2 - 0.3$	$0.3 - 0.4$	$0.4 - 0.5$

Table 4: POD and ROM choices evaluated. All models use 25 POD modes.

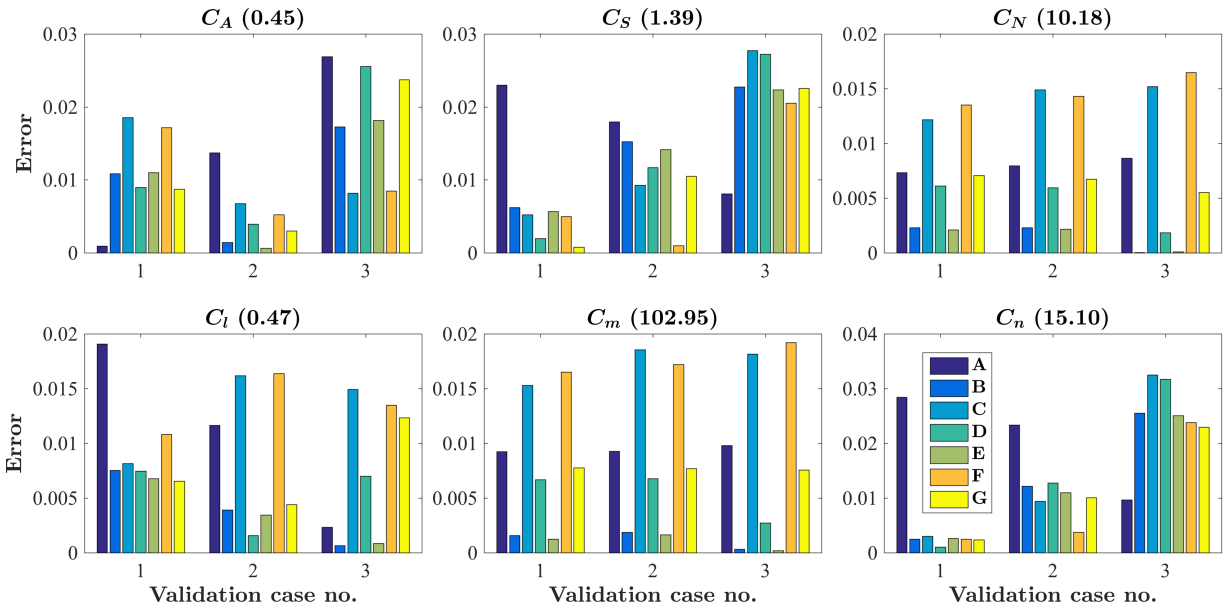


Figure 10: Validation cases

learning database forms a three-dimensional matrix of operating parameters – viz. Mach number, angle of attack and roll angle. Thus, `griddata` resorts to tessellation of the set of operating parameter triplets corresponding to the retained snapshots to 3-dimensional simplices, followed by linear barycentric interpolation of the first N POD coefficients on the particular simplex that contains the new operating parameter triplet.

The strategy outlined in Section 6.2.1 of reusing the learning database to assess the effectiveness of empirical methods is employed to evaluate the performance of the interpolation approach. In particular, we make use of the previously constructed S distinct POD bases, each omitting one of the original snapshots of the learning database. Interpolation is used to predict the aerodynamic coefficients of the omitted snapshot from each such reduced POD basis.

The linear interpolation in `griddata` fails in cases that fall outside the convex hull of the corresponding reduced database – e.g. the cases of $\alpha = 20°$, $\phi = 0°$ or $45°$ and either of the two Mach numbers 0.6 or 0.8. We resorted to nearest-neighbour interpolation in these cases, and thus they may be disregarded for the present purposes of comparison.

Figure 11 presents the result of this exercise in terms of the fractional errors in interpolation-based prediction of the six aerodynamic coefficients using the first 25 POD modes, evaluated on the database of 32 cases. The interpolation fares quite poorly in comparison with the corresponding ROM results shown in fig. 9, with errors that are almost an order of magnitude greater. Note that the worst cases (with almost 100% error) in terms of the side force, rolling moment and yawing moment are *not* due to nearest-neighbour interpolation; they pertain to cases with $\alpha = 20°$ and

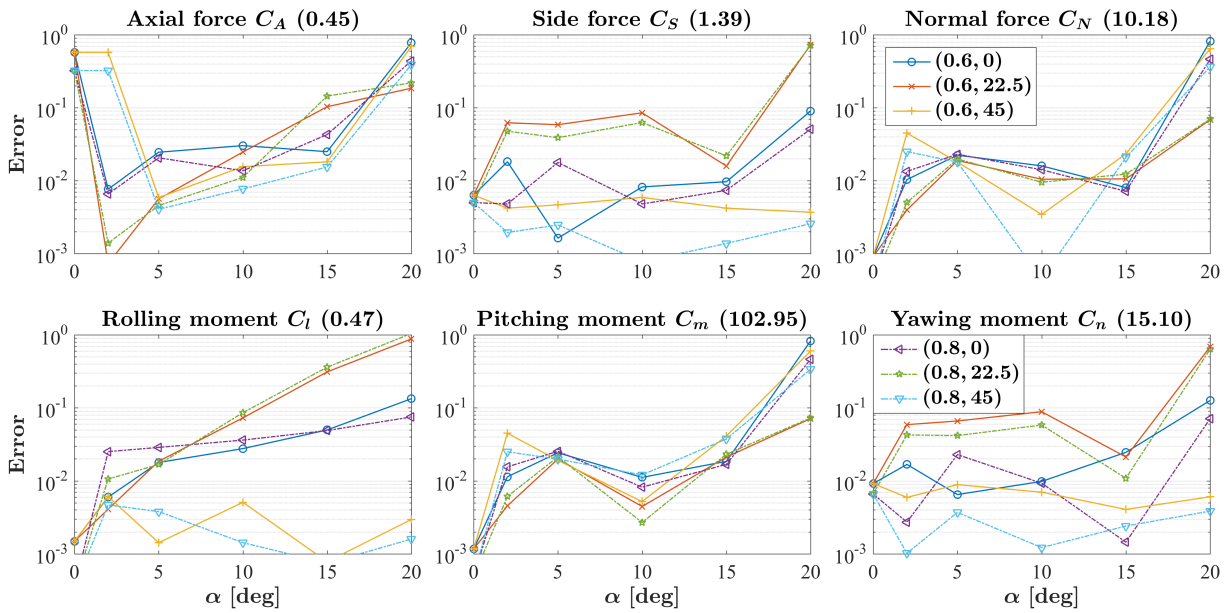


Figure 11: Fractional errors in aerodynamic coefficients predicted using linear interpolation with the first 25 POD modes defined on the reduced domain. The scheme of presentation follows fig. 5.

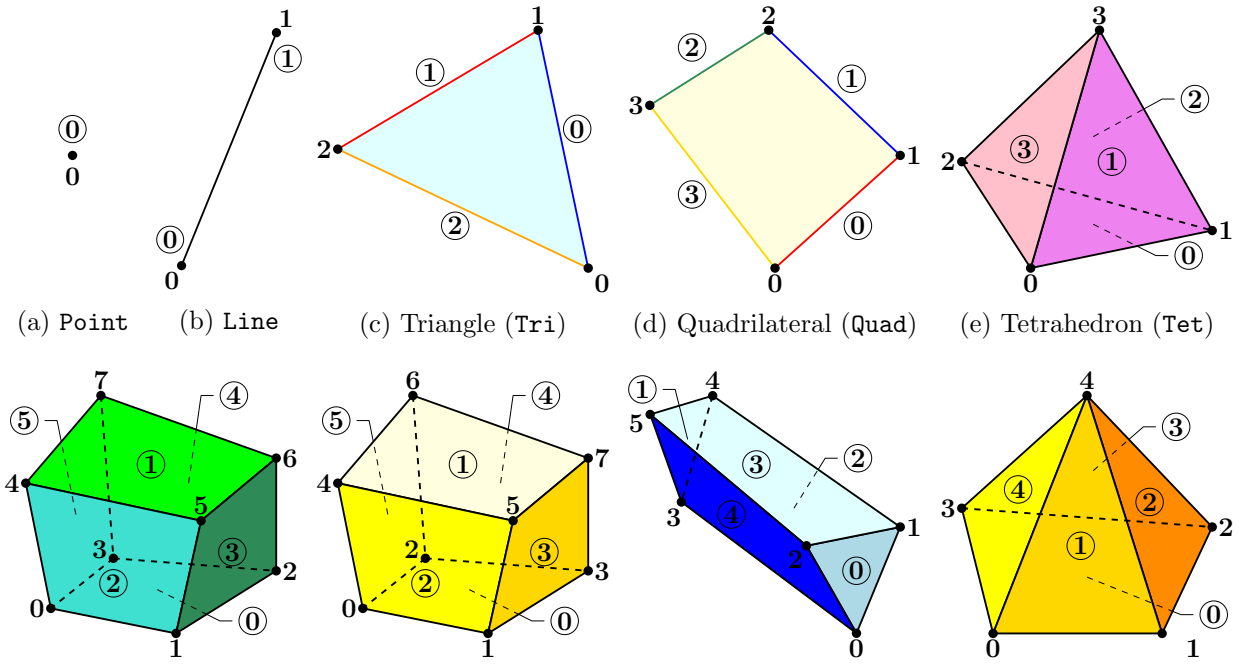
$\phi = 22.5°$ that are contained within the convex hull of the remaining cases. Large errors are also observed in the axial force predictions at $\alpha = 2°$, $\phi = 45°$.

7 Conclusion

In this report, we demonstrate the application of a reduced-order model (ROM) approach to the prediction of aerodynamic force and moment coefficients of a subsonic missile. It leverages an existing high-order model (HOM) database, where flow solutions have been calculated for certain missile operating conditions consisting of combinations of freestream parameters – viz. Mach number, angle of attack and roll angle. ANSYS Fluent was used to compute these flow solutions at DRDL, Hyderabad. The ROM developed in this project 'learns' from the HOM database, and is then able to subsequently predict the aerodynamic coefficients at any other operating condition that falls within the range of conditions evaluated in the learning database. Limited excursions outside this range (i.e., extrapolations) are also possible.

The ROM approach makes use of proper orthogonal decomposition (POD) to extract the essential flow features from the learning database. It subsequently solves an optimization problem, wherein a flow solution is sought that satisfies the boundary conditions corresponding to a new operating condition, all the while ensuring that the solution produces the minimum residual with respect to the governing equations.

The ROM is very efficient. A calculation for a single operating condition takes at most half a minute, and the errors are generally within 2% in case of interpolations, and are at worst within about 10% even for limited extrapolations. These are to be compared with the classical database interpolations that are regularly performed for prediction of aerodynamic coefficients. For routine interpolations, this method is seen to result in around 10% errors, but the errors approach 100% in some interpolation cases, and definitely in cases involving extrapolation. This attests to the superior performance of the proposed ROM approach.



(a) `Point` (b) `Line` (c) Triangle (`Tri`) (d) Quadrilateral (`Quad`) (e) Tetrahedron (`Tet`)

(f) Hexahedron (`Hex`) (g) Voxel (`Vox`) (h) Wedge (`Wedg`) (i) Pyramid (`Pyr`)

Figure 12: The nine cell types available in the `UnstructuredMeshData` class, with their formal identifiers indicated in parentheses. The nodes and faces are numbered (the latter with circles), in the particular order indicated.

To implement the ROM, a suite of Python programs is developed. A major feature of the suite is the deep access that it provides to the HOM database of the ANSYS Fluent 'case' and 'data' solution files. This allows the subsequent POD calculations, as well as the evaluation of the residuals of the governing equations – all implemented in Python. DRDL scientists are being trained on the algorithm and the software.

APPENDICES

A Python class `UnstructuredMeshData`

The POD/ROM approach requires intimate access and manipulation of CFD mesh and data (essentially, a flow snapshot in the prevailing terminology). Moreover, the mesh is assumed to be unstructured, thereby requiring significant bookkeeping. A survey of the existing file formats for storing and accessing such data revealed a lacuna. Most formats store the mesh along with the data; this is wasteful of storage since all the snapshots in a database that we deal with in this project are necessarily on the same mesh for application of POD. ANSYS Fluent[14] has one of the few file formats that store the mesh and data files separately – viz. in the `case` and `data` files. However, it too has a shortcoming for our present purpose, which will be clarified below.

We have developed a Python class to store an unstructured mesh and/or data in a flexible lightweight non-redundant format, particularly useful for reducing storage required for a database of snapshots all having the same underlying mesh. The data structure can also handle hanging nodes in the mesh. The expected filename extension of an `UnstructuredMeshData` snapshot is `.usm`. It will be clear later that this is only a notional extension as no actual file is created with it.

The major elements of a mesh are the cells, nodes, faces and edges (see fig. 12); in the present implementation we do not identify edges explicitly. All mesh storage formats record the nodes explicitly; however, there are various ways of recording the cell and/or face connectivities. We call a mesh storage format 'cell-major' if the primary data structure records the cells and their connections to the nodes; the faces, if recorded explicitly, are only specified as attributes of their parent cells. This is exemplified by the VTK (Visualization ToolKit) format[19]. The opposite is the 'face-major' data structure, as exemplified by ANSYS Fluent[14]. Our implementation uses the cell-major format, but we also provide the facility to store face definitions and face-based snapshot data.

As shown in fig. 12, the `UnstructuredMeshData` data structure can handle point cells in a 0D mesh, line cells in a 1D mesh, triangle and quadrilateral cells in a 2D mesh, and tetrahedron, hexahedron, voxel, wedge and pyramid cells in a 3D mesh. Note that the 2D elements serve as faces of the 3D mesh cells, the 1D element is the sole type of face of 2D cells, and so on. As an extreme case, the 0D `Point` element is only useful as the sole type of face in a 1D mesh. The main cell types and their node-ordering follow the VTK format[19]; the additional aspect of the present implementation is the definition of their faces. In fact, the voxel is a hexahedral cell with a different node ordering, retained for compatibility with VTK. Note that in `UnstructuredMeshData`, as in the underlying Python programming language, indices of mesh entities are assumed to start from 0, unless mentioned otherwise. However, from time to time, we will have to use 1-based indexing since the sign of the index will have to carry special meaning. This switch in indexing basis will be handled internally in the class, and will be transparent to its client routines.

The concept of mesh faces deserves additional discussion. Consider the face 0 of the hexahedral cell shown in fig. 12f, and without loss of generality assume that it is the 0th face of the entire mesh (and not just of the cell). Let the nodes of face 0 be numbered $0-1-2-3$ (or any cyclic permutation thereof). The normal of the face is taken to point along the thumb of the right hand if the fingers curl around in the given order of the nodes. Thus, the above node order of face 0 signifies that its normal points into the cell under consideration. To encode this, we would record the index of this face of the cell as -1, where the negative sign indicates that this face points into the cell and the 1 is the 1-based index of the face. We switch to 1-based indexing to allow the use of signs (-0 being of course indistinguishable from $+0$ otherwise). If on the other hand, face 0 was numbered $3-2-1-0$ (or any cyclic permutation thereof), then the index of this face of the cell would be noted as $+1$. As another example, face 3 of the tetrahedron will be noted as $+4$ (resp. -4) if its nodes are numbered in the order $0-3-2$ (resp. $0-2-3$).

For 2D grids, the only possible cell faces are lines. Here, the face normal is determined by the cross product of two vectors, $\hat{r}$ and $\hat{k}$. The $\hat{r}$ vector extends from the first to the second node of the linear face, whereas the $\hat{k}$ vector has its origin at the first node and points out of the grid plane toward the viewer. For example, in the 2D grid shown in fig. 13, if face 1 is directed from node 3 to node 4, then its normal would point from cell 0 to cell 7. Thus, the face 1 would be noted as $+2$ in cell 0 and as -2 in cell 7. These conventions of face directions are adapted from ANSYS Fluent's case file format[14].

Figure 13 also exemplifies a mesh with hanging nodes, created due to local refinement. Out of the four quadrants centred about node 4, the one in the fourth quadrant is sub-divided into four cells, and one of them is further sub-divided into four cells. In our data structure, we do not keep track of the level of refinement – that would require a tree. Instead, we only record the 'active' cells that are numbered. We do record the 'inactive' faces that are created by the hanging nodes. For example, nodes 8 and 11 are hanging on the right face of cell 7, and this inactive face is numbered 30; its 'active' child faces are 5, 10 and 16. We will also note the corresponding 'active' child cells of this inactive face – viz. 2, 5 and 8.

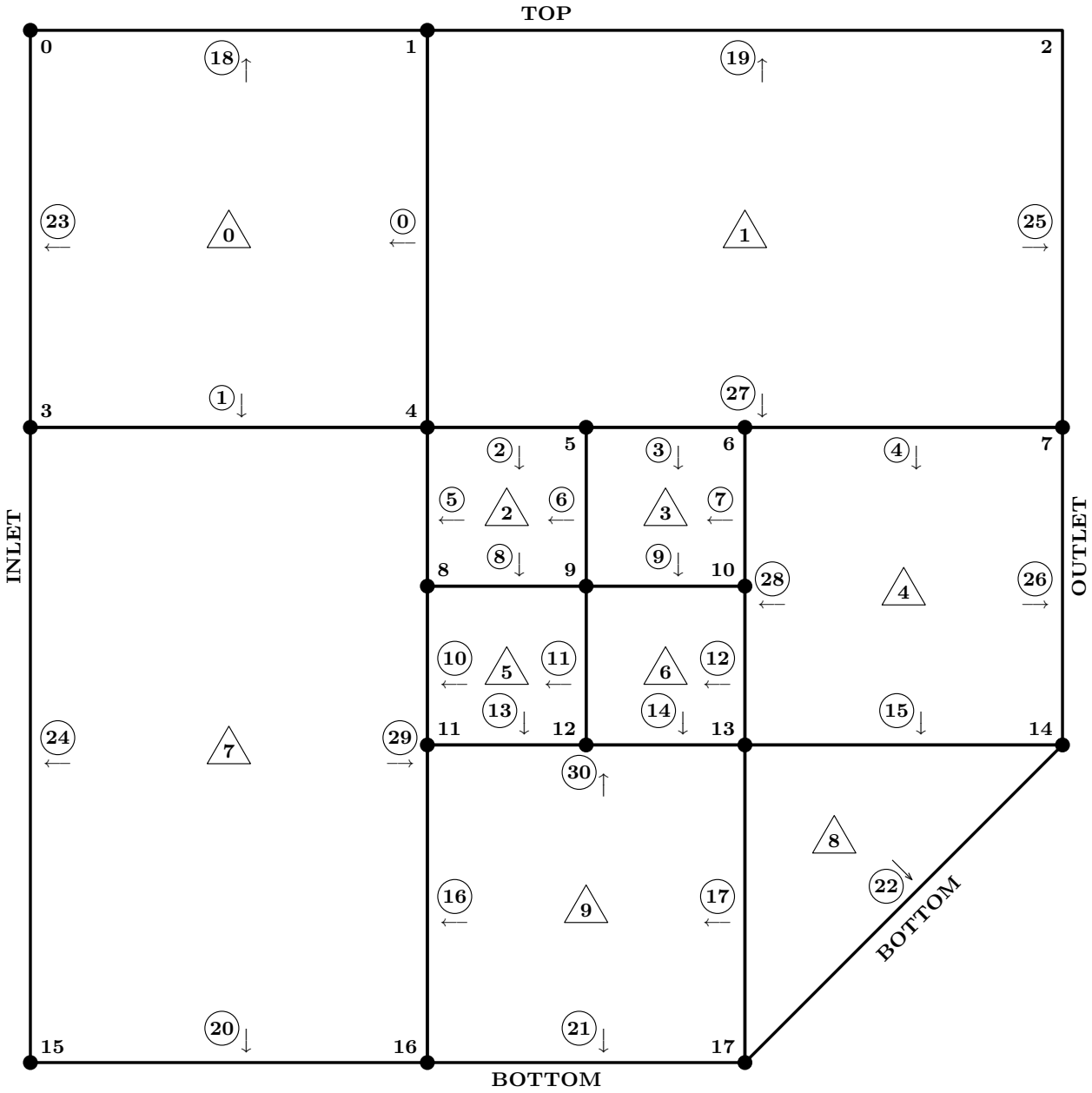


Figure 13: Example of an unstructured 2D mesh with hanging nodes. Cell indices are enclosed inside triangles and face indices are inside circles; node indices are not enclosed. The directions of the faces are indicated. The overall dimensions of the mesh are 13×13 units (including the missing triangle), and the origin is at node 15.

There are thus three types of faces in a mesh – (active) interior faces, (active) boundary faces and inactive (interior) faces. We index the faces such that these three types of faces appear as consecutively-numbered blocks in this order. We typically impose different conditions on portions of the boundary of a mesh (called 'markers'). Within the set of all boundary faces, we ensure that they again form consecutively-numbered sub-blocks for each such marker. These numbering schemes are exemplified by the 2D unstructured mesh of fig. 13, and are adapted from ANSYS Fluent's case file format [14].

The main attributes (each is a separate data structure) of the `UnstructuredMeshData` class are now described in turn. The metadata of the snapshot is stored in a text file; all the actual data structures of the mesh and data are stored in separate zipped-and-compressed `numpy .npz` files. Although this apparently causes a proliferation of files, the advantage is efficiency of retrieval and access for any one of these data structures selected. Only the first three of the following attributes (viz. `meta`, `nodes` and `cellNodes`) are mandatory for a snapshot, all the others being optional.

`meta`: The metadata of the `UnstructuredMeshData` object is stored in a text file. For the notional mesh and data file `flow.usm`, the actual metadata file is named `flow_meta.txt`. For the 2D grid exemplified in fig. 13, this is

```
NDIME= 2
NNODE= 18
NCELL= 10
NFACE= 31
FACEINTI= 1
FACEINTF= 18
FACEBNDI= 19
FACEBNDF= 27
FACEINACTVI= 28
FACEINACTVF= 31
CELLTYPSD= ( Tri, Quad )
FACETYPSD= ( Line )
CELLDATA= ( density )
NODEDATA= ( pressure )
FACEDATA= ( densityU, densityV )
MESH_FILE= flow.usm
MARKER_PREFIX= flow_marker
```

Evidently, `NDIME` is the number of dimensions of the mesh (1, 2 or 3). The total numbers of nodes, cells and faces are given by `NNODE`, `NCELL` and `NFACE`, respectively. The 1-based indices of the initial and final interior faces are noted in `FACEINTI` and `FACEINTF`; the analogous information regarding the boundary and inactive faces are noted in the next four entries, respectively.

The `CELLTYPSD` is an ordered list of types of domain elements (cells) in the mesh. For example, in case of a 3D mesh, this list can be `(Tet, Hex, Vox, Wedg, Pyr)`, or a subset thereof, depending on which types of cells are actually present. In case of a 2D mesh, this can be `(Quad, Tri)` as exemplified here; for a 1D mesh, the only option is `(Line)`. The ordering of this list is arbitrary; however, once this ordering is specified here, several of the subsequent data structures are ordered based on this. The cell-face analogue of the above cell metadata is the `FACETYPSD` entry.

The data (i.e., flow variables) can be stored on the cells, nodes or faces of the mesh. In the above example, density is stored on cells, pressure on nodes, and the two components of the momenta are specified on mesh faces.

The `MESHFILE` entry allows one to link a common mesh to multiple snapshots, thereby significantly reducing storage overhead. Suppose there are two snapshots on a common mesh with notional mesh+data files named `snapA.usm` and `snapB.usm`. We will create one mesh file, say `flow.usm`, that will store the mesh data structure in various files, all prefixed by `flow_`. Then, in the metadata files `snapA_meta.txt` and `snapB_meta.txt`, we will set `MESH_FILE` to `flow.usm` (or the full path to it, if it is in a different path). With this, we need not store the meshes of snapshots `snapA.usm` and `snapB.usm` again; these notional files will only store the non-redundant flow variable data explicitly. If `MESH_FILE` is not specified, then the mesh is also expected to be stored with the appropriately

prefixed filenames – `snapA_` and `snapB_`, respectively.

The `MARKER_PREFIX` serves a purpose similar to `MESH_FILE`, in that it too allows multiple snapshots to share the definition of their meshes' markers.

`nodes`: This is a 2D `numpy` array of nodes' coordinates, with rows ordered by the nodes' indices and columns being x-, y- and z-coordinates; either z- or both y- and z-coordinates may be omitted (in case of 2D or 1D meshes, respectively). For the notional snapshot named `flow.usm`, this array is stored in `flow_mesh_nodes.npz`. In the 2D unstructured mesh exemplified in fig. 13, we have

```
>>> nodes
array([[ 0., 13.],
       [ 5., 13.],
       [13., 13.],
       [ 0.,  8.],
       [ 5.,  8.],
       [ 7.,  8.],
       [ 9.,  8.],
       [13.,  8.],
       [ 5.,  6.],
       [ 7.,  6.],
       [ 9.,  6.],
       [ 5.,  4.],
       [ 7.,  4.],
       [ 9.,  4.],
       [13.,  4.],
       [ 0.,  0.],
       [ 5.,  0.],
       [ 9.,  0.]])
```

`cellNodes`: This is a 1D `numpy` array of objects of the same length as `cellTypsD` in `meta`; each object is either empty (if there are no cells of the corresponding type in the mesh), or a 2D array of cells' node indices (in 0-based indexing) with cells along rows and their nodes along columns. Recall that all cells of the same type have the same number of nodes. The cell nodes must be ordered as shown in fig. 12. For a snapshot named `flow.usm`, this data structure is stored in `flow_mesh_cellNodes.npz`. In the example of fig. 13, we have

```
>>> cellNodes
array([array([[17, 14, 13]]),
       array([[ 4,  1,  0,  3],
              [ 7,  2,  1,  4],
              [ 9,  5,  4,  8],
              [10,  6,  5,  9],
              [14,  7,  6, 13],
              [12,  9,  8, 11],
              [13, 10,  9, 12],
              [16,  4,  3, 15],
              [17, 13, 11, 16]])], dtype=object)
```

Evidently, the first 2D subarray corresponds to the sole triangular cell, whereas the second one gives the node connectivity of all the quadrilateral cells.

`cellMapInv`: This gives the mapping from the entries of `cellNodes` to the indices of the cells in the overall mesh. It is an array of the same length as `cellTypsD`; each entry is either empty (if there are no cells of the corresponding type in the mesh), or a 1D array giving the (0-based) index of the corresponding cell of this type in the overall mesh. If all mesh cells are of the same type, or if their ordering renders this data structure trivial, `cellMapInv` is `None`. For the 2D mesh of fig. 13, it is

```
>>> cellMapInv
array([array([8]),
       array([0,1,2,3,4,5,6,7,9])], dtype=object)
```

If the `cellMapInv` is non-trivial for the snapshot `flow.usm`, it is stored in `flow_mesh_cellMaps.npz`; otherwise this file need not exist.

`cellMap`: This is the direct mapping from the cell indices to the entries of `cellNodes`; in that sense, `cellMapInv` is its inverse map. It is a 2D `numpy` array, with rows corresponding to the cell indices in the mesh. The first of its two columns gives the index in `cellTypsD` or `cellNodes` corresponding to the cell type. The second column gives the row index into the corresponding entry of `cellNodes` for this cell type – i.e., the 2D subarray. This is again `None` if all mesh cells are of same type, or if the ordering is trivial; it is stored in the same file as `cellMapInv`. For the prevailing example of fig. 13, we have

```
>>> cellMap
array([[ 1,  0],
       [ 1,  1],
       [ 1,  2],
       [ 1,  3],
       [ 1,  4],
       [ 1,  5],
       [ 1,  6],
       [ 1,  7],
       [ 0,  0],
       [ 1,  8]])
```

`cellFaces`: As mentioned at the outset, the `UnstructuredMeshData` data structure is cell-major in format, and thus the preceding cell-node connectivity is mandatory. One can optionally record the face information of the mesh also, in the `cellFaces` data structure. It has the same format as `cellNodes` above, but it stores the signed indices of faces of each cell (in 1-based indexing), instead of its nodes. The convention for recording face indices of a cell has been described above. The unique ordering of the faces for each type of cell has been depicted in fig. 12. For example, for a quadrilateral face, the first (linear) face runs from the first to the second node, the second face runs from the second to the third node, and so on. Corresponding to fig. 13, the data structure is

```
>>> cellFaces
array([array([[ 23, -16,  18]]),
       array([[ -1,  19,  24,   2],
              [ 26,  20,   1,  28],
              [ -7,  -3,   6,   9],
              [ -8,  -4,   7,  10],
              [ 27,  -5,  29,  16],
```

```
               [-12,   -9,   11,   14],
               [-13,  -10,   12,   15],
               [ 30,   -2,   25,   21],
               [-18,   31,   17,   22]])], dtype=object)
```

In conjunction with `cellNodes`, this data structure is sufficient for fully specifying all the faces of the mesh. For the snapshot named `flow.usm`, it is stored in `flow_mesh_cellFaces.npz`.

`faceFaceTr`**:** This is a 2D array encoding the parent-to-child relation of inactive faces; it is omitted if there are no hanging nodes in the mesh. Each row corresponds to an inactive face, starting with the first one occurring in the mesh (i.e., the `FACEINACTVI` entry in the `meta` attribute). The first column gives the number of children of this face. In magnitude, the entries of this many subsequent columns give the 1-based indices of these child faces. The signs of these column entries carry meaning: a positive sign implies that the child face is aligned with the parent, whereas a negative sign connotes the opposite. There may be empty columns (the dummy entry is '0') at the end of some rows since all inactive faces need not have the same number of children. The total number of columns is determined by the maximum number of children across all parent faces. This data is stored in the same file as `cellFaces`. For the 2D grid of fig. 13, we have

```
>>> faceFaceTr
array([[  3,   3,   4,   5],
       [  2,   8,  13,   0],
       [  3,  -6, -11, -17],
       [  2, -14, -15,   0]])
```

Evidently, the rows correspond to the inactive faces 27, 28, 29 and 30, respectively.

`cellNebrs`**:** This is an optional data structure again of the same format as `cellNodes`, but storing the 1-based indices of cells neighbouring each cell, instead of its nodes. The order of the neighbours is the same as the respective intervening faces in `cellFaces`. An entry of '0' indicates that the corresponding face is on a boundary of the mesh (hence, 0-based indexing could not be used). A negative entry signifies the magnitude of the (1-based) index of the inactive (parent) face interfacing with several adjacent neighbours. Corresponding to fig. 13, it is

```
>>> cellNebrs
array([array([[  0,   5,   10]]),
       array([[  2,   0,   0,   8],
              [  0,   0,   1, -28],
              [  4,   2,   8,   6],
              [  5,   2,   3,   7],
              [  0,   2, -29,   9],
              [  7,   3,   8,  10],
              [  5,   4,   6,  10],
              [-30,   1,   0,   0],
              [  9, -31,   8,   0]])], dtype=object)
```

For the snapshot named `flow.usm`, `cellNebrs` is stored in `flow_mesh_cellNebrs.npz`. This data structure is redundant as it can be reconstructed from the other mesh connectivity data structures described above, but only at significant computational cost. Thus, if the cell-to-neighbouring-cell connectivity is required in calculations, storing the above `cellNebrs` may be computationally

efficient.

`faceNebrTr`**:** This is identical in structure to `faceFaceTr`, but it contains the (unsigned) 1-based indices of the respective active neighbour cells (that have the corresponding child faces). In case of fig. 13, this is

```
>>> faceNebrTr
array([[  3,   3,   4,   5],
       [  2,   4,   7,   0],
       [  3,   3,   6,  10],
       [  2,   6,   7,   0]])
```

This data is stored in the same file as `cellNebrs`, and is redundant just like it; it is stored for computational efficiency.

`faceNodesCells`**:** This is a 1D `numpy` array of objects of the same length as `faceTypsD` in `meta`; each object is either empty (if there are no faces of the corresponding type in the mesh), or a 2D array listing the indices of nodes and adjacent cells of faces, with the faces along rows. The last two columns contain the 1-based indices of the two adjacent cells of the face; a '0' indicates a boundary face or an inactive face. The previous columns contain the indices of the faces' nodes in 0-based indexing (all faces of the same type have the same number of nodes). The ordering of the face nodes is such that the face normal points from the first adjacent cell to the second one. We ensure that boundary or inactive faces are ordered such that their '0' entries appear in the last column. All faces are lines in the 2D unstructured mesh exemplified in fig. 13, so that we have

```
>>> faceNodesCells
array([array([[ 1,  4,  2,  1],
              [ 3,  4,  1,  8],
              [ 4,  5,  2,  3],
              [ 5,  6,  2,  4],
              [ 6,  7,  2,  5],
              [ 4,  8,  3,  8],
              [ 5,  9,  4,  3],
              [ 6, 10,  5,  4],
              [ 8,  9,  3,  6],
              [ 9, 10,  4,  7],
              [ 8, 11,  6,  8],
              [ 9, 12,  7,  6],
              [10, 13,  5,  7],
              [11, 12,  6, 10],
              [12, 13,  7, 10],
              [13, 14,  5,  9],
              [11, 16, 10,  8],
              [13, 17,  9, 10],
              [ 1,  0,  1,  0],
              [ 2,  1,  2,  0],
              [15, 16,  8,  0],
              [16, 17, 10,  0],
              [17, 14,  9,  0],
              [ 0,  3,  1,  0],
              [ 3, 15,  8,  0],
```

```
[ 7,  2,  2,  0],
[14,  7,  5,  0],
[ 4,  7,  2,  0],
[ 6, 13,  5,  0],
[16,  4,  8,  0],
[13, 11, 10,  0]])], dtype=object)
```

Note that this data structure is redundant as it can be deduced from `cellNodes` and `cellFaces`, albeit at great computational cost; thus, we can still optionally maintain it for computational efficiency. For a snapshot named `flow.usm`, `faceNodesCells` is stored in `flow_mesh_faceNodesCells.npz`. This data structure is adapted from the method employed in ANSYS Fluent for storing face connectivity information.

`faceMap` **and** `faceMapInv`**:** These are the face-index counterparts of `cellMap` and `cellMapInv`. That is, they give the mapping of the face ordering of the overall mesh to and from the entries of `faceNodesCells`. The maps are trivial for the example 2D mesh (in fact for any 2D mesh), since all faces in a 2D mesh are of `Line` type. If they are non-trivial, they are stored in the same file as `faceNodesCells`.

`data`**:** The flow variables of a snapshot can be specified on nodes, cells or faces of the mesh, as discussed in the context of `meta`. They will be stored as double-precision 1D arrays of lengths equalling the numbers of nodes, cells and faces in the mesh, respectively. Each variable is stored in its own separate binary file. For example, corresponding to the example `meta` listed earlier, we will have the following four files: `flow_density_cell.bin`, `flow_pressure_node.bin`, `flow_densityU_face.bin` and `flow_densityV_face.bin`.

B Extraction of solution data on faces from the HOM solutions

An HOM will provide some of the flow variables on grid nodes, cell centroids and/or face centroids. For example, in the ANSYS Fluent data used for the present application, the face-integrated mass flow ($\int \rho \boldsymbol{u} \cdot \hat{\boldsymbol{n}} \mathrm{d}A$) and pressure were available on all the faces of all grid cells. On the other hand, the three components of velocity, pressure, temperature and density were provided at the centroid of each grid cell. All variables except the density were provided on all the boundary faces too; however, the latter was readily calculated from the available pressure and temperature data using the ideal gas law. It may appear that the mass flow and cell-based temperature are redundant in this setting. However, HOMs use sophisticated (possibly iterative) interpolation techniques to obtain requisite face-based data from cell-based data, which are more costly to recompute than to store[20].

To facilitate accurate evaluation of the residual in the present application, the POD was performed on face-based data alone. As mentioned above, pressure and mass flow were available already on all faces. Three different methods were used to interpolate the remaining cell-based data – viz. the three components of velocity and density – to the interior faces of the grid. In the first approach, a simple arithmetic average of the values of the flow variable at the two adjacent cells was deemed as the value at the intermediate face. In the second method, the second-order upwind interpolation scheme was employed, mimicking the HOM solver[20]. In the final method, second-order upwind approach was used for the velocity components, but arithmetic average was employed for density. The results demonstrate that the much cheaper first option yields results that are indistinguishable from those obtained from the costlier second option.

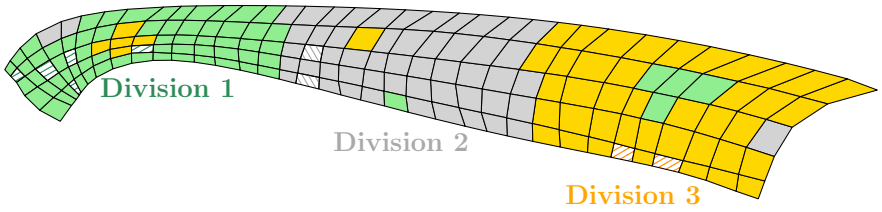


Figure 14: Schematic of (boundary) marker of a 3D mesh divided into three parts by their face indices, for the purpose of calculation of the distance of domain cells from the marker. Solid-filled faces are considered, whereas cross-hatched faces are disregarded for being too close to other faces in the marker.

Owing to the various approaches used in the HOM solver for interpolating the different variables, it was found that the mass flow calculated from the velocity and density data did not match exactly with the values provided on the faces in the solution data. The latter values did agree with the mass conservation equation, in terms of zeroing of the net mass flow on faces of each cell. However, this 'reliable' face-integrated mass flow cannot be included as a separate variable in the POD/ROM owing to mathematical redundancy. Instead, the following approach was used to amend the interpolated velocity so as satisfy continuity exactly with the overall interpolated flow data. The dominant component of the normal vector was determined for each cell face in the grid. Then, the corresponding component of the face velocity was updated without changing the other components or the density, so that the resulting face-integrated mass flow matched its available data. Subsequently, the face-based mass flux vector $(\rho u, \rho v, \rho w)^{\mathrm{T}}$ required for the POD-ROM model were calculated by multiplying the interpolated (and corrected) components of the velocity vector with the interpolated density. The requisite face-based density is also obtained in the process. The face-based pressure data was directly available in the solution.

C Identifying an annular region around the missile

As discussed in the text, one may wish to identify cells in a domain mesh that are within a certain range of distances from a boundary marker. This requires a many-to-many distance evaluation – calculating the distance from each domain cell to every marker face – that can be extremely expensive. For example, in the missile aerodynamics database used here, there are about 10 million domain cells and about 0.5 million missile surface faces; naively, this would require 10^{13} distance evaluations!

Here we incorporate two simultaneous approaches to reduce the computational burden. First of all, the many-to-many distance evaluation is embarrassingly parallellizable. Thus, we divide the total number of faces into a certain number of divisions – say as many as the number of processors available (see fig. 14). This division was done by the face indices. Usually, meshing softwares index mesh entities such that consecutively indexed entities are found physically near each other; however, this is neither guaranteed, as depicted in fig. 14, nor is it necessary in our approach. The distance is calculated from each domain cell centroid to every marker face centroid in a division, and the minimum distance found across all marker faces is considered as the distance of the particular cell from the marker in the particular division. These cell-to-marker-division distances are stored in files. Once all marker divisions have been addressed, we minimize over their individual distance results to determine the final distance values from the overall marker.

Although the above parallelization does reduce the time by one or two orders of magnitude (depending on the number of processors and memory available to each), the number of calculations

needed is still too large. To identify approaches that can significantly reduce the computation effort, we refer back to the intended use of the distance values calculated. They would be used to identify the cells that are far enough away from the marker, but not too far. Obviously, these limits do not need to be precisely implemented, as there is no definite rule for setting them in the first place. That is, if we wish to ignore cells that are within $0.1D$ of the missile, then it is still alright if we end up including a few cells that are at a distance between $0.095D$ and $0.1D$ from the surface. In other words, we do not need infinite precision in estimating the distance of a cell from the marker. To exploit this, we can omit from consideration those marker faces that are too close to its neighbours (as determined by a specified tolerance), since inclusion of these faces will not materially affect the subsequent choice of the domain cells to consider in any case. The proposed approach is exemplified in fig. 14 wherein some marker faces are flagged that are too close to its neighbours, and thus may be disregarded. Since the flow close to a wall marker has sharp gradients, marker face meshes are typically highly resolved. Thus, judicious choice of a high-enough tolerance, say $0.01D$, can significantly reduce the number of marker faces to be considered.

To implement the above step efficiently, we use the algorithm described below.

1. Maintain two sets of marker faces – one for the faces to be definitely retained (say F_r), and the other for faces that yet to be interrogated for potential retention or rejection (say $F_?$). Initially, all marker faces are placed in $F_?$ and F_r is empty.

2. Take any marker face from $F_?$ as reference, move it to F_r, calculate its distances from all remaining entries of $F_?$, and remove those faces from $F_?$ that are nearer to the reference face than the specified tolerance.

3. Take another marker face from $F_?$ as reference. Note that this face has already been deemed to be distant enough from all other faces in F_r, and thus may be retained (i.e., moved to F_r). Now repeat the previous step for this reference face, to further pare down $F_?$.

4. Repeat the above step until $F_?$ is empty. Evidently, the number of repetitions is fewer than the total number of marker faces.

There is a further efficiency that is to be realized. As mentioned above, (marker) faces with adjacent indices are more likely to be found physically close to each other. Thus, if we randomize the choice of the successive reference faces then we may eliminate more proximal faces at each step, than if we choose them sequentially per their indices.

Evidently, the two approaches towards efficiency are mutually inclusive; we can apply the set reduction algorithm to each division of the marker that is being processed in parallel. However, there is an added benefit of parallelization in the set reduction. Suppose that there are n_c domain cells and n_f marker faces divided into n_d divisions (parallel processes). Then, the computational complexity of the set reduction step in each division is $(n_f/n_d)^2$, and that of the cell-to-face distance calculation is $(n_f/n_d)n_c$. It is evident that, whereas the overall cost across all divisions is unchanged for the distance calculation $(n_f n_c)$, it is reduced by a factor of n_d for the set reduction step. Given that n_f is a significant fraction of n_c, this represents a substantial saving and actually makes the computation tractable on a modest-sized processor cluster with shared memory.

D Identifying and using domain cells with 'plane' faces only

Due to issues in grid generation, some faces of 3D grid cells may not be plane. For ease of calculation of face fluxes, it will be useful to identify and work with only those cells that have planar faces.

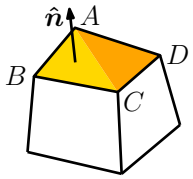


Figure 15: Schematic of cell with a possibly non-planar face.

Of course, this is only a problem if a face has more than 3 nodes (e.g. a quadrilateral) in 3D. An example is shown in fig. 15.

All points (including the nodes) on a plane face must satisfy the equation:

$$\boldsymbol{P} \cdot \hat{\boldsymbol{n}} = d,$$

where, $\boldsymbol{P}$ is the position vector of the test point on the face, $\hat{\boldsymbol{n}}$ is the face-normal vector, and d is the signed distance (along the normal) of the face from the origin. The face normal is found from any three consecutive nodes of the face (ABC in the example), assuming that the face is planar. The normal is calculated in the usual manner – taking the cross-product of the directed line segments $\overrightarrow{BC}$ and $\overrightarrow{BA}$, and unitizing the result. Since these three nodes have been used to obtain the face normal, they must yield the same d (up to machine precision) in the above equation. The remaining nodes of the face (i.e. node D in the example) should also yield the same d, unless the face is not plane. Hence, our planarity check is the equality of the d values evaluated from all nodes of a face. In particular, we calculate the 1-norm of the differences between the d values of the nodes (second one w.r.t. first one, third one w.r.t. second one, ... , last one w.r.t. penultimate one). If this 1-norm is less than a specified tolerance (say 1e$-$7), then the face is considered plane. To avoid numerical round-off errors, the face centroid is set as the origin for the face nodes' coordinates in the above expression. If all the faces of a cell are planar, then it is deemed a plane-faced cell.

E Program architecture

The program developed consists of several packages, each consisting of tens of individual Python codes. We first describe a general-purpose library developed called `MyPythonCodes`, some of whose functionalities are used in other packages. Then we describe the sequence of commands to be issued that are specific to the current project; these latter commands pertain to programs available in the `SteadyROM/MissileDatabase_PY` and `SteadyROM/SteadyAeroDyn_PY` libraries.

The folders where these library are placed should be in the `PYTHONPATH` of the environment. For example, suppose that `MyPythonCodes` and `SteadyROM` code packages are placed in `${HOME}/Research`; then we can include the following line in `${HOME}/.bashrc`:

```
export
   PYTHONPATH=${HOME}/Research:${HOME}/Research/SteadyROM/SteadyAeroDyn_PY:
   ${HOME}/Research/SteadyROM/MissileDatabase_PY:${PYTHONPATH}
```

E.1 The `MyPythonCodes` library

This library consists of three sub-libraries that are detailed below.

E.1.1 `MyPythonCodes/tools`

This is the most general library; its architecture is depicted in fig. 16. Some of the codes are adapted from the Python interface suite of SU2[21].

E.1.2 `MyPythonCodes/mesh`

This library implements several routines for accessing and manipulating CFD mesh and data. In particular, it contains the `UnstructuredMeshData` class described in appendix A; it also implements various programs that access CFD mesh and data available in various standard formats, like `VTK`[19] and `SU2`[21], and converts them to the `UnstructuredMeshData` format. The architecture of this library is presented in fig. 17. Note that these programs (as also in all other programs developed) depend on the `MyPythonCodes/tools` library, but these dependencies are not shown explicitly.

E.1.3 `MyPythonCodes/FluentReader`

As discussed above, the implementation requires intimate access to the CFD mesh and data of the snapshots. For this project, the data is originally available as ANSYS Fluent's case and data files. One option that was explored at the outset was the use of Fluent's built-in VTK exporter. However, for the mesh involved in the snapshot database under consideration, the exported mesh had an incorrect topology as illustrated in fig. 18. This was observed for all snapshots in the database, and not just for a single file. Apparently, the exporter could not handle some hanging nodes near the surface of the missile. There are other available Python implementations that purport to provide access to output from ANSYS Fluent, a prominent example being the `{vtkFLUENTReader` class of the `VTKPython` library. This would have been more useful as the access could be obtained programmatically. However, this approach also suffered from the same issue of the spurious mesh. Possibly, both approaches use the same fundamental routine that is flawed.

To address this problem, we have developed the `MyPythonCodes/FluentReader` library under discussion, which allows one to read ANSYS Fluent's case and data files. For the various reasons delineated in appendix A, we convert the mesh and data so retrieved to our native `UnstructuredMeshData` format, and store the latter. The structure of the library is documented in fig. 19. A four-step approach is taken.

1. The Fluent case and data files are parsed into the metadata and data. The metadata is stored in series of text files – Pythons 'pickle' format is used for ready access.

2. The data is transformed into a structure that can be stored the most efficiently – this step is especially necessary in case of meshes consisting of a mix of cell types. The resulting data is saved in a set of unformatted binary files – these are called `FluentBits`.

3. ANSYS Fluent stores the mesh connectivity information in face-major structure; see discussion in appendix A. That is, for each cell face in the mesh, it provides information on the type of the face, its node indices, and the indices of the two cells on either side of it. This data structure is similar to the `faceNodesCells` attribute of the `UnstructuredMeshData` class discussed; the latter is actually adapted from this. On the other hand, most other file formats, including `UnstructuredMeshData` store the connectivity information in cell-major format. Thus, a code is written to convert from face-major to cell-major mesh structure. This effort was particularly complicated by the presence of hanging nodes in the Fluent mesh; however, this problem was addressed satisfactorily.

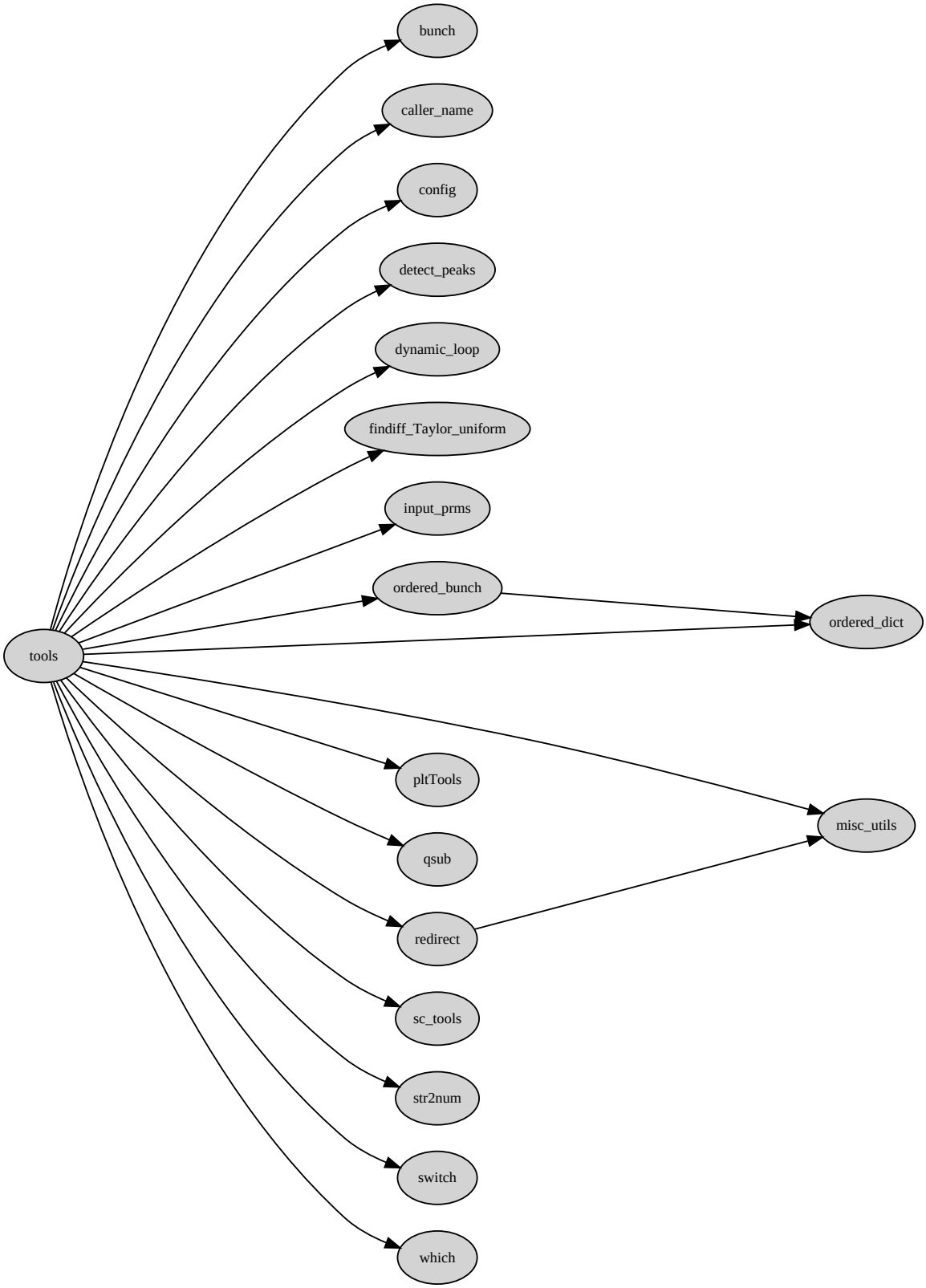


Figure 16: Architecture of `MyPythonCodes/tools`.

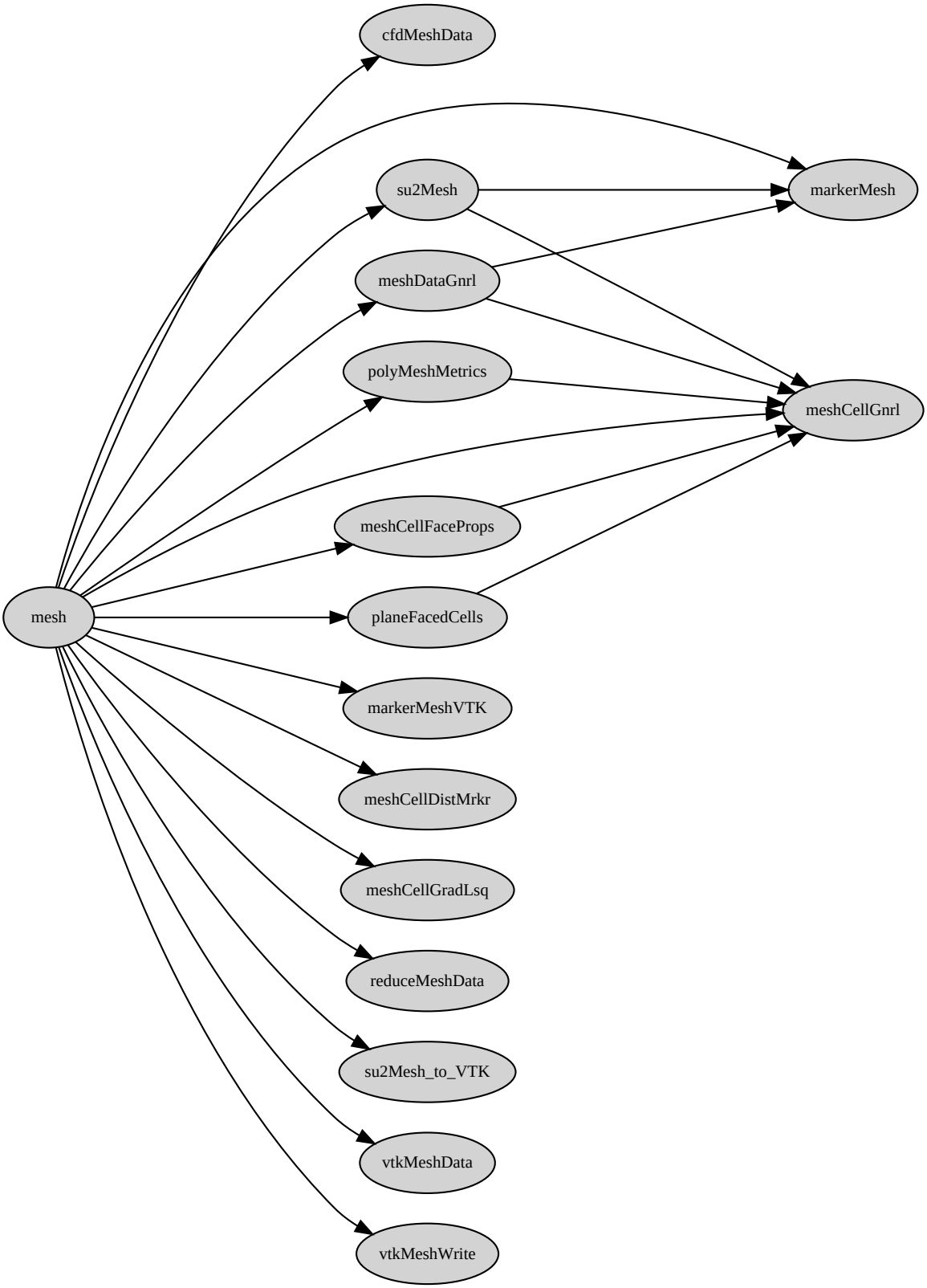
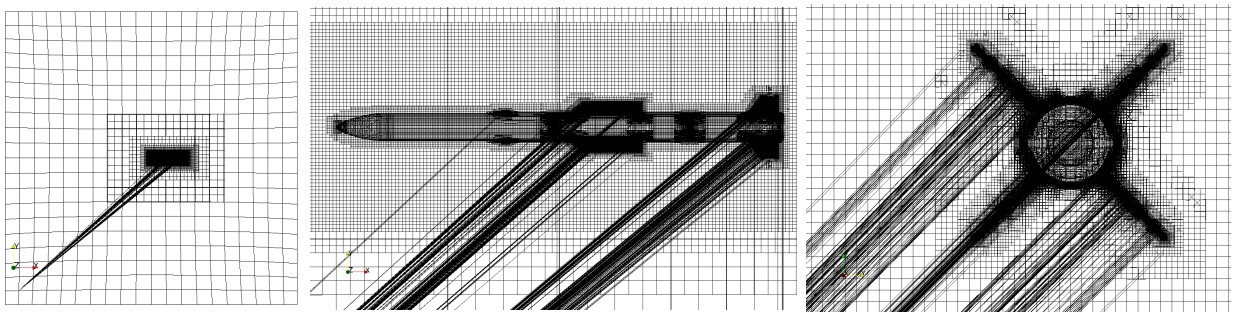


Figure 17: Architecture of `MyPythonCodes/mesh`.



(a) Overall $X-Y$ view (b) Zoomed $X-Y$ view (c) Zoomed $Y-Z$ view

Figure 18: Issues with the off-the-shelf Fluent-to-VTK convertor.

4. Finally, another code converts this set of files into the `UnstructuredMeshData` format. Similar codes can be written to convert to any other format (e.g., VTK, SU2, Tecplot, etc.).

The library is designed to handle ANSYS Fluent output on two- and three-dimensional unstructured meshes.

E.2 Libraries specific to the project

For the present project, we have written two Python packages – viz. `MissileDatabase_PY` and `SteadyAeroDyn_PY`. Both of these are provided within the folder `SteadyROM`. The first library processes the snapshot database provided (in ANSYS Fluent's case and data format), and reformats the same into a standard database file system; it thus relies heavily on the `MyPythonCodes/FluentReader` library described above. The second library performs the POD and ROM on any such standard snapshot database. The actions of the first package may be substituted for any other input database (say for data generated in another CFD package), but this will be transparent to the second package. This affords a large degree of code reuse.

We now detail the specific commands to do the two tasks above – viz. snapshot database preprocessing, and actual POD+ROM calculation.

E.2.1 Commands for processing raw snapshots

Here, we describe in detail the typical set of Python commands to create the snapshot database from the ANSYS Fluent case and data files. We assume that the files are named using the following convention:

```
<prefix>_phi<roll>_m<Mach>_a<AoA>.[cas,dat]
```

We also assume that both the case and data files for any Fluent flow solution (snapshot) to be processed are present in the same folder, and have the same file name except of course the extensions. We will store the processed snapshots in the in-house `UnstructuredMeshData` format. The programs used for this can be found in the library `<PathToSteadyROM>/MissileDatabase_PY`.

1. We first extract the mesh that is common across all the snapshots from any snapshot's Fluent case file `<CASFILE>` by running the Python program:

```
python <PathToSteadyROM>/MissileDatabase_PY/missileProcMeshFluent.py
   -c <CASFILE> -f <CFGFILE> -r <FOLDER> -s -g -d <D>
```

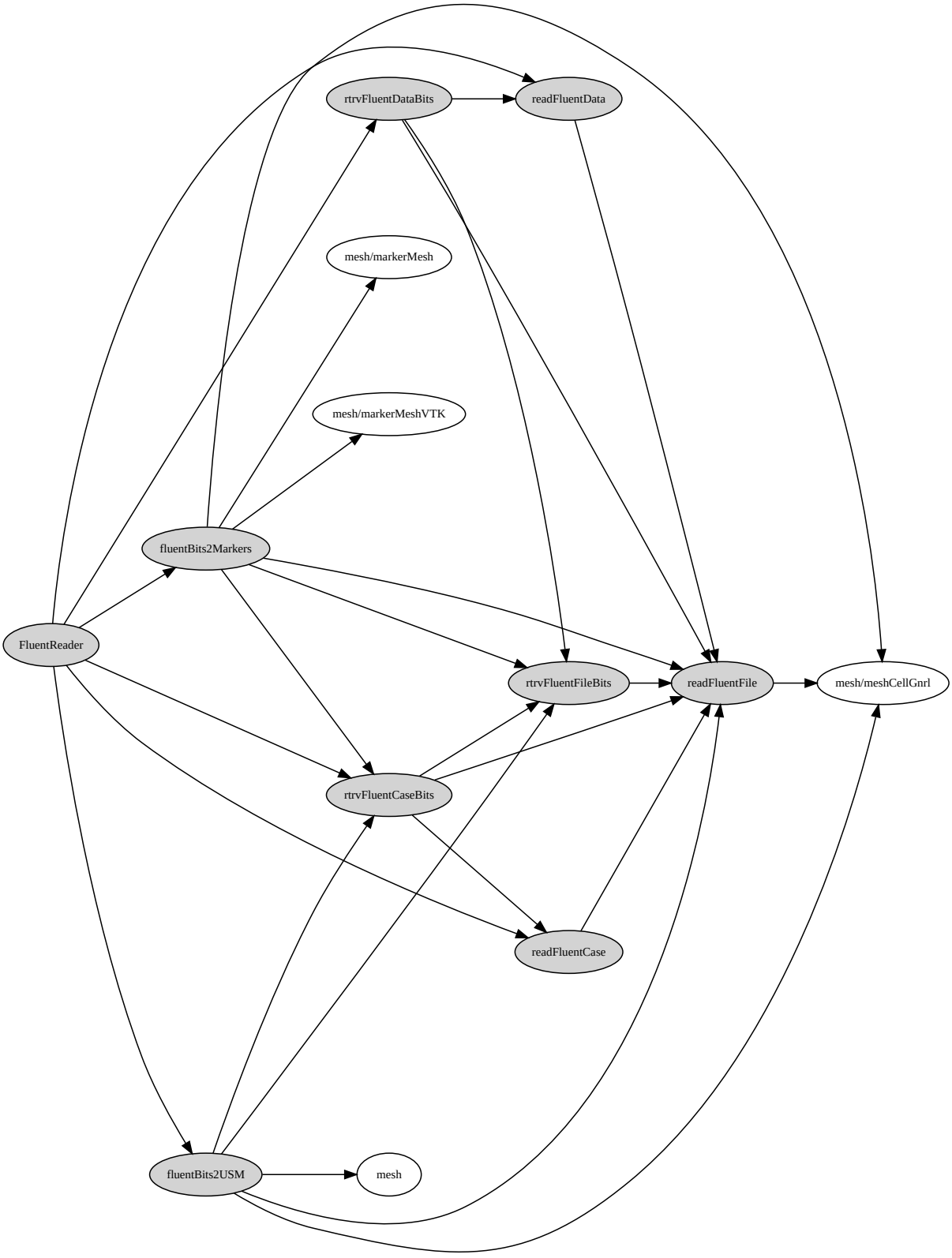


Figure 19: Architecture of 'MyPythonCodes/FluentReader'.

Since this is the first program discussed, we have explicitly listed the command; for later programs, we will simply list the following kind of detailed usage.

```
Usage: missileProcMeshFluent.py [options]

Options:
  -h, --help  show this help message and exit
  -c CASFILE  process Fluent case file CASFILE
  -f CFGFILE  use configuration from CFGFILE
  -r FOLDER   output resulting mesh to FOLDER
  -s          store wall zones in separate markers
  -g          calculate weights for cell-based gradient evaluation
  -d D        specify level D of debugging info
```

The `-s` switch indicates that one wishes to separately extract all (boundary) markers of the mesh with wall boundary condition, instead of lumping them all in one 'wall' marker (which is the default behaviour if the switch is not supplied).

The `-g` switch indicates that one wishes to calculate and store the weights of cell-centroid values towards the least-squares cell-based gradient computation. This calculation should be avoided (by not supplying the switch) if such gradients will not be needed (see below).

The `-d D` option controls the level of verbosity of debugging information streamed to `stdout`. The default is quiet operation (i.e., `D` is 0).

It is suggested to use a case with roll angle of 45° as the reference mesh, since this is the nominal roll orientation, and the Fluent mesh for other roll angles are rotated with respect to this. A sample configuration file `CFGFILE` for controlling the run is listed below:

```
% Snapshot prefix
VOLUME_FLOW_FILENAME= flow
% File format for storage of mesh & data
OUTPUT_FORMAT= UNSTRUCTUREDMESHDATA
% Non-dimensionalization
REF_DIMENSIONALIZATION= FREESTREAM_VEL_EQ_MACH
% Characteristic length: missile diameter (m)
CHAR_LEN= 0.315
% Ratio of specific heats
GAMMA_VALUE= 1.4
% Ambient density (kg/m**3)
DENSITY_AMB= 1.15496
% Ambient temperature (Kelvin)
TEMPERATURE_AMB= 303.15
% Gas constant for air (SI units)
GAS_CONSTANT= 287.058
```

The first entry gives the filename (sans extension) that will be used to store the mesh. The second entry explicitly specifies the storage file format. In the prevailing example, an `UnstructuredMeshData` mesh file named `flow.usm` is created in the specified output `FOLDER`. The third entry indicates that the flow variables will be non-dimensionalized such that the freestream velocity equals the freestream Mach number, the freestream density equals unity, and the freestream pressure equals `1/GAMMA_VALUE`. Moreover, the fourth entry (the characteristic length) will be used for non-dimensionalizing all lengths (including the nodes' coordinates

in the mesh). Actually, only these first four relevant for the mesh extraction program. These, along with the subsequent thermodynamic constants, are needed in the next step that reuses this configuration file.

Note that this process is lengthy (it takes about 2 hours!), but needs to be done only once for the whole database. The main time is spent in determining the cell-major data structure of the `UnstructuredMeshData` from Fluent's face-major data structure.

A user may choose to extract the mesh without calculating the gradient weights in the first iteration (by omitting the `-g` switch), and later calculate the gradient weights only. For this later run, the case file (with `-c` option) should *not* be specified, so as to avoid reprocessing the mesh.

2. Now, we extract and process the data of all the snapshots; this can be done sequentially or in parallel. The command to process a single snapshot is

```
Usage: missileProcFaceDataFluent.py [options]

Options:
  -h, --help  show this help message and exit
  -c CASFILE  process Fluent case file CASFILE
  -r REFFLDR  use reference common mesh from REFFLDR
  -o FOLDER   output resulting data to FOLDER
  -s INTERP   interpolation option INTERP: AA, U2 or A2
  -p          extract primitive or conserved variables
  -d D        specify level D of debugging info
```

This program creates a sub-folder in `<FOLDER>` with the name of the case file `<CASFILE>` sans the file extension, and extracts the data therein. As an example, suppose `<CASFILE>` is `msl_phi45_m0.6_a05.cas`. Then, the final result of this program is the creation of the notional `UnstructuredMeshData` snapshot file `flow.usm` in the path `<FOLDER>/msl_phi45_m0.6_a05`. The run is automatically configured by the configuration file used in the previous program, a copy (actually a symbolic link) of which was created in `REFFLDR` by the previous program.

The mesh is linked from the `<REFFLDR>` where it has been extracted by the previous program. It is not extracted again as it is (assumed to be) topologically identical to the common mesh extracted in the previous step. However, some cells and/or faces in the mesh may need re-numbering (strangely!), which is done automatically here.

The POD-ROM requires data to be specified on the mesh faces, whereas ANSYS Fluent stores most of the data on cells. The spatial discretization (or interpolation) to be done is specified by the `-s` option. The default, 'AA', commands simple arithmetic averaging of the two adjacent cell values to obtain the intervening face's value. Other options include 2nd-order upwinding for all variables (commanded by '-s U2'), or such upwinding for the velocity vector field only but arithmetic averaging for other cell-based velocities (commanded by '-s A2'). If the two latter methods are to be used, then the previous mesh extraction should be performed with the `-g` option so as to calculate the necessary weights for gradient computation once and for all.

The `-p` switch signifies that the primitive velocity vector field is to be extracted; if this switch is not given then the default behaviour is to extract the corresponding conserved (momentum) vector field instead.

The `-d` controls the verbosity of printing as before.

The data correction procedure discussed in appendix B is implemented in this program.

3. The identification of the annular region around the missile, as discussed in appendix C, requires the distance of each domain cell to be calculated from the surface. This is done by the following program.

```
Usage: missileCalcSurfaceDists.py [options]

Options:
  -h, --help    show this help message and exit
  -f MESH       calculate cell distances of MESH
  -m MARKER     calculate distances from MARKER in MESH
  -t TOLERANCE  approximate distances up to TOLERANCE
  -n NDIVS      divide marker into NDIVS divisions
  -d IDIV       calculate distances for division no. IDIV
  -a            assemble previously calculated distances from various
                divisions?
```

There are two modes of invoking this program. We can do the entire calculation in one call, whence only the `-f` and `-m` arguments, and optionally the `-t` argument, should be specified. In the alternative approach (see appendix C), the marker is notionally divided into `NDIVS` divisions. The distance of all domain cells from each such division (numbered by `IDIV`) is calculated in a separate (possibly parallel) process. In these processes, the `-n` and `-d` arguments must be specified, in addition to the other arguments mentioned above. Once all these individual preliminary calculations are complete, the program should be called one last time, this time with the mandatory arguments `-f`, `-m`, `-n` and `-a` to compile the preceding individual results into the final distance values.

As an example with `NDIVS` $= 3$, we make the following program calls:

```
>> ... -f flow.usm -m wall -t 0.01 -n 3 -d 1
>> ... -f flow.usm -m wall -t 0.01 -n 3 -d 2
>> ... -f flow.usm -m wall -t 0.01 -n 3 -d 3
>> ... -f flow.usm -m wall -n 3 -a
```

All except the final command can be issued in parallel (barring memory constraints); the final command should be issued after all the other other processes have completed.

The overall result of either calling approach is the creation of an addendum to the reference `MESH`. In particular, for the `MESH` named `flow.usm`, this program creates the zipped and compressed `numpy` file `flow_mesh_cellDistMrkr.npz`. The above program can be called with the names of the various markers in the mesh; all the distance data so generated get appended to the above file.

4. At this stage, we can choose to work with a reduced domain for all the snapshots, possibly using the above-calculated cell-to-surface distances to identify the domain cells to retain. If so, we invoke the following program.

```
Usage: missileReduceMeshData.py [options]
```

```
Options:
  -h, --help  show this help message and exit
  -c CFGFILE  use configuration from CFGFILE
  -f FOLDER   FOLDER where all data are stored
  -x PREFIX   data sub-folders' common PREFIX
  -o PATH     output PATH
  -d D        specify level D of debugging info
```

A sample `CFGFILE` for controlling the run is listed below:

```
% Definition of sub-domain of mesh to retain
DEF_SBDMNS= {DISTS : ( wall | 0.2 | 2.0 )}
```

This entry defines the sub-domain of the mesh to retain. In this example, it is instructed to use (previously-calculated) cell distances from the 'wall' (boundary) marker of the mesh, and to retain those cells whose distances fall in the range 0.2 to 2.0. Note that the mesh length has previously been non-dimensionalized by the characteristic length.

With this, all the snapshots present in sub-folders starting with `PREFIX` in `FOLDER` are reduced, and the results are stored in correspondingly named sub-folders in `PATH`.

Note that after this reduction step, the file structure of the original and reduced data will be identical. Both will have all the available snapshots stored in sub-folders within a single data folder. This data folder will also contain a link to the original configuration file that was used in the first program, with a particular filename, viz. `config_CFD.cfg`.

5. The identification of all plane-faced cells in the mesh, as discussed in appendix D, is performed by the following program.

```
Usage: identifyPlaneFacedCells.py [options]

Options:
  -h, --help  show this help message and exit
  -f MESH     calculate cell distances of MESH
  -t TOL      tolerate up to TOL out-of-plane faces
  -d D        specify level D of debugging info
```

A tolerance value of 1e−7 was found to be useful in applications. Given a mesh file named `flow.usm`, say, the program creates the file named `flow_mesh_iPlaneFacedCells.npy` within the same folder, which contains the `numpy` array of identified cell indices of the mesh.

6. Agglomeration of snapshots available under a data folder in a formatted database is performed by the following program.

```
Usage: missileAgglomerateDb.py [options]

Options:
  -h, --help  show this help message and exit
  -c CFGFILE  use configuration from CFGFILE
  -p SRCPATH  data snapshots available in SRCPATH
  -x PREFIX   snapshot sub-folders' prefix
  -o PATH     output PATH
```

A sample CFGFILE that controls the run is presented below:

```
% Mach numbers of the snapshots
MACHS= ( 0.6, 0.8 )
% Angle of attacks of the snapshots
AOAS = ( 0, 2, 5, 10, 15, 20 )
% Roll angles of the snapshots
ROLLS = ( 0, 22.5, 45 )
```

This indicates the combinations of Mach number, angle of attack and roll angle of the snapshots expected to be available in SRCPATH in appropriately-named sub-folders having the common PREFIX.

All these specified snapshots are copied from SRCPATH to PATH as a standard file structure. In particular, a sub-folder named SNAPSHOTS is created in PATH, and separate numbered sub-sub-folders within it store each snapshot. The common underlying mesh is stored in a separate sub-sub-folder named MESH. Additionally, a text file named snaps_info.txt is created within SNAPSHOTS containing the parameter value combinations corresponding to each snapshot. Finally, a configuration file named config_CFD.cfg is written out in SNAPSHOTS containing the essential metadata of the database, e.g., the common names of files (typically flow), their format (usually UnstructuredMeshData), their type of non-dimensionalization (typically FREESTREAM_VEL_EQ_MACH), etc.

7. Now that the snapshot database has been processed in the desired format, we require the user to manually paste into the SNAPSHOT sub-folder of the above PATH a comma-separated-variable (.csv) file containing the aerodynamic coefficients of each of the snapshots. Its name should be precisely Snaps_ADCoeffs.csv. It should be of the form as below:

```
Roll,Mach,AoA,ZONE,CA,CS,CN,Cl,Cm,Cn
0,0.6,0,total,0.33,0.,0.,0.,0.,0.
0,0.6,2,total,0.33,-0.002,0.59,-0.008,-6.,0.001
0,0.6,5,total,0.32,0.009,1.725,-0.01,-17.77,0.22
...
```

The first three columns are fixed (but not in their order). The fourth column is also fixed. There can be more aerodynamic coefficients than the ones shown. Also, the rows can be in any order, as long as all the snapshots in the database are accounted for. The aerodynamic coefficients are assumed to be calculated by the CFD solver suite (ANSYS Fluent, in this case).

E.2.2 Commands for POD

The POD modes of the snapshot database are computed now using the following program; it is to be found in the <PathToSteadyROM>/SteadyAeroDyn_PY library.

```
Usage: POD_calculation.py [options]

Options:
  -h, --help    show this help message and exit
  -d DO         which part of calculation to DO
  -c CFGFILE    use configuration from CFGFILE
```

```
-o PATH        output PATH
-i INDEX       row INDEX of kernel to do if DO = 3
-n NPOD        further augment up to NPOD POD modes if DO = 7
-v VERBOSITY   specify level of VERBOSITY
```

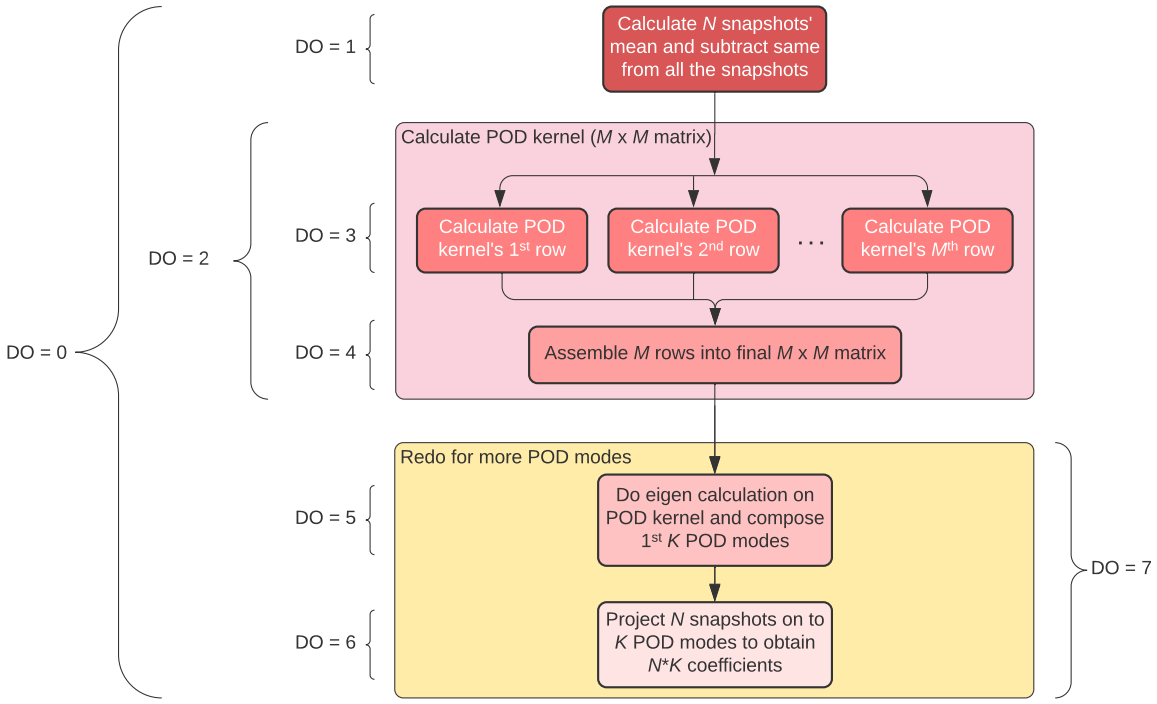


Figure 20: Flowchart of POD calculation steps

The calculation of the POD modes, shown in the schematic of fig. 20, may be quite time-consuming. Fortunately, the most costly step (`DO = 3`) is embarrassingly parallel. Thus, we can invoke the above program in several serial steps, of which this one step is in turn invoked in parallel calls. This behaviour is decided by the `DO` switch as follows.

```
0: Perform entire calculation (all following steps) at once
1: Calculate snapshots' mean, and subtract same from snapshots
2: Calculate entire POD kernel (matrix) at once (subsumes next two steps)
3: Calculate one row of kernel (call as many times as no. of snapshots)
4: Assemble POD kernel from individual rows calculated in above step
5: Perform eigen-decomposition of kernel and compose POD modes
6: Project all snapshots onto all POD modes computed
7: Extend number of POD modes computed and used in above two steps
```

Evidently, the POD calculation can be accomplished with either `DO = 0`, or by serially invoking with `DO = 1, 2, 5, 6`, or by calling with `DO = 1, 3, 4, 5, 6`. The last option involves multiple calls with `DO = 3`. For a call with `DO = 3`, we specify the particular row `INDEX` of the the POD kernel (matrix) to compute. This should run from 0 to one less than the number of snapshots in the database. Such calls can be made in parallel. Once all rows are calculated, we can assemble

the kernel from them using one call with `DO = 4`. The resulting file structure containing the POD modes and auxiliary data are stored in a folder named `POD` within the prescribed output `PATH`.

Subsequently, if we decide to compute more POD modes (and project the snapshots on to these extra modes), then we can call the program with `DO = 7`. In this case, we have to specify the new total number of POD modes with the `-n NPOD` argument.

The computation is configured by `CFGFILE`, a sample of which is given below.

```
% Snapshot folder with path
SNAP_PATH= <path_to_and_including_SNAPSHOTS>
% Names of variables on which to perform POD
VAR_NAMES= (density, pressure, densityU, densityV, densityW)
% No. of POD modes to calculate and store
NPOD= 25
% Type of inner product to use
IP_TYP= 0
% Name of module where 'BCFreestream' class is to be found
BCFREESTREAM_MDL= MissileDatabase
% Name of module where 'ADCoeffsAccess' class is to be found
ADCOEFFS_MDL= MissileDatabase
% Use vector POD approach? Or scalar?
POD_VECTOR= YES
% Subtract the snapshots' mean from the snapshots before doing POD?
POD_SUBTRACT_MEAN= YES
% Definition of sub-domain for evaluating inner product
%DEF_SBDMNS= {DISTS : ( wall | 0.2 | 2.0 )}
```

Here, `SNAP_PATH` is the path to the snapshots folder. In the previous section, we have described how the snapshots will be placed in a folder named `SNAPSHOTS`. It is the path to this folder that must be specified. The flow variables to be decomposed are specified in the list `VAR_NAMES`. It will be recalled from the previous section that the `missileProcFaceDataFluent.py` program had specified whether to extract primitive (velocity) or conserved (momentum) variables. The `VAR_NAMES` options available now are decided by that earlier choice. If in case we had chosen to work with primitive velocity variables instead, then this should have been `VAR_NAMES= (density,pressure,u,v,w)`. The number of POD modes to calculate (less than the number of snapshots available in the database) is set in `NPOD`. The type of inner product to use is determined by `IP_TYP`. Although there are several options available, this should typically be set to 0, as exemplified here. Both the `BCFREESTREAM_MDL` and `ADCOEFFS_MDL` should be set to `MissileDatabase` as of now (note that their parent folders are assumed to be in the `PYTHONPATH`, as stated at the outset of appendix E). In the future, if alternative access methods to the aerodynamic coefficients' database or the freestream boundary conditions are implemented, then their container modules may be inserted here instead.

The remaining items are optional. If the vector POD approach (involving all flow variables) is to be pursued (which is recommended), then `POD_VECTOR` should be set to `YES`, as shown. Else, it can be set to `NO`, or omitted from `CFGFILE` altogether. Similarly, if the snapshots' mean is to be subtracted from all the snapshots prior to POD calculations (which is recommended), then `POD_SUBTRACT_MEAN` should be set to `YES`. Else, it can be set to `NO`, or even omitted. Finally, one can choose a sub-domain of the over all mesh to include in the evaluation of the integral involved in the inner product. An example is given in the `DEF_SBDMNS` definition. However, it is found in applications, that the full mesh may be used advantageously. In that case, this should be omitted from the configuration file; it is commented out here.

E.2.3 Commands for ROM

The aerodynamic coefficients for a new operating condition (not in the original database) are calculated using the above-computed POD modes in a Reduced Order Model (ROM) using the following program; it is to be found in the `<PathToSteadyROM>/SteadyAeroDyn_PY` library.

```
Usage: ROM_solution.py [options]

Options:
  -h, --help    show this help message and exit
  -c CFGFILE    use configuration from CFGFILE
  -o PATH       output PATH
  -v VERBOSITY  specify level of VERBOSITY
```

The output file structure of this call is stored in a folder names `PROJECT` within the prescribed output `PATH`. The optimized values of the POD coefficients can be found in `optim_out.txt`, and the corresponding aerodynamic coefficients are written out to `soln_ADCoeffs.csv` in the same format as the `Snaps_ADCoeffs.csv` described at the end of appendix E.2.1.

The computation is configured by `CFGFILE`, a sample of which is given below.

```
% Set of parameters defining case
PARAM_NAMES= ( Roll, Mach, AoA )
% Mach number of case (based on free-stream speed of sound)
Mach= 0.8
% Angle of attack of case (degrees)
AoA= 10.0
% Side-slip angle of case (degrees)
Roll= 11.25
% Folder where POD data is to be found
POD_PATH= <PathToPOD>
% Optimization objective function (only option is EULER) w/ scale factor
OPT_OBJECTIVE= EULER * 1000000.0
% Sub-domain where objective function is to be evaluated
DEF_SBDMNS= [{ICELLSFILE:<_iPlaneFacedCells.npy>} ^ {DISTS:(wall|.2|.3)}]
% Perform optimization in RAM by loading all POD modes or not
OPT_OBJECTIVE_RAM= YES
% Store the residuals on all cells or not (useful for debugging)
OPT_OBJECTIVE_STORE= NO
% Optim. constraint fncs. w/ optional scale factors, separated by ';'
%   E.g.: (Constraint=Value) * Scale; use '>','<' or '='.
% Equality constraint available now is BC_FAR.
OPT_CONSTRAINT= ( BC_FAR = 0.0 )
% Module w/ 'BCFreestream' class, for evaluating this b.c.
OPT_CONSTRAINT_BCFAR_MODULE= MissileDatabase
% Optimization design variables to use. There are four options:
%   Option A1 (vector POD): {IMPLICIT:(25|2.0)}
%   Option A2 (scalar POD): {IMPLICIT:(0,3,5,2,1|4.0)}
%   Option B1 (vector POD): {EXPLICIT:(0|2|3);(0|-2|4);(0|1.5|2)}
%   Option B2 (scalar POD): {EXPLICIT:(0|2|3);(3|-1|4);(0|1|2);(2|5|7)}
DEFINITION_DV=  { IMPLICIT : ( 25 | 2.0 ) }
% Interpolate learning d/b for initial values of design variables, or not
INTERP_INIT_DV= YES
% Optimization accuracy
```

```
OPT_ACCURACY= 1.e-12
% Maximum no. of optimization iterations
OPT_ITERATIONS= 100
% Whether to reconstruct the solution from the POD modes, or not
OUT_RECON= NO
% Module w/ 'ADCoeffsAccess' class, for outputting solution's A/D coeffs
OUT_ADCOEFFS_MODULE= MissileDatabase
```

Most of the entries are self-explanatory; we discuss the ones that are not.

`DEF_SBDMNS`: This can define the intersection (indicated by '^') of several different sub-domains, for evaluation of the optimization objective function. These individual sub-domains can be defined by either specifying a file containing identified cells (`ICELLSFILE` option), or by giving a distance range from a marker (`DISTS` option). The latter has already been discussed in the context of the mesh domain reduction application earlier. The former can, for example, refer to the cells identified as plane-faced earlier, by specifying the file created by the `identifyPlaneFacedCells.py` program. As per the example, we are indicating that the objective function should be evaluated on all those cells that are between 0.2 and 0.3 units distant from the wall marker of the mesh *and* are plane-faced. Although not exemplified here, we may also specify the sub-domain as the union of one or more rectangular parallelepipeds in a 3D mesh using the `RECT3` option (or the `RECT2` option for rectangles in 2D).

`DEFINITION_DV`: The design variables (POD coefficients) to be optimized may be specified either explicitly or implicitly; the actual format depends on whether vector or scalar approach was used in POD calculations. The initial values of the POD coefficients are left unspecified in the `IMPLICIT` approach, and a common absolute bound factor is set on their possible values during optimization. For vector POD approach, we simply specify the number of first few POD modes to consider and their common bound factor (25 and 2.0 respectively, in the example of `Option A1`). In the scalar POD approach, we specify the first few POD modes to consider from each separate POD calculation (i.e., flow variables), followed by their common bound factor (4.0, in the example of `Option A2`). In particular, the first zero, three, five, two and one POD modes are to be considered from the zeroth to fourth calculations; the corresponding variables are determined by the `VAR_NAMES` entry in the POD configuration file. The bound factor of a particular POD coefficient is multiplied by its maximum absolute value found in the original learning database of snapshots to arrive at the actual bound value. The initial values of all POD coefficients are set to 0 if `INTERP_INIT_DV= NO` or unspecified; else they are obtained by interpolating the learning database using the parameter values (Mach number, angle of attack, and roll angle). Alternatively, the initial values of the POD coefficients are individually set in the `EXPLICIT` approach, as are their respective absolute bound values. The first atom of a parenthetical argument is the POD calculation index (starting from 0), the second atom is the actual initial value, and the last is the absolute bound value to be respected in the optimization. The first occurrence of a particular calculation index refers to its first POD mode, the second occurrence pertains to its second POD mode, and so on. For vector POD approach, the only allowable calculation index is 0; in the example of `Option B1`, the first three vector POD modes are to be considered, their respective initial values are 2, -2 and 1.5, and their respective absolute bound values are 3, 4 and 2. For the scalar POD example of `Option B2`, the first two modes from the zeroth POD calculation are considered with respective initial values 2 and 1, and absolute bound values 3 and 2; no mode from the first calculation is to be included; only the first mode from the second calculation is considered with initial value 5 and bound 7; only

the first mode from the third calculation is to be included with initial value -1 and bound 4.

References

[1] T. Bui-Thanh, M. Damodaran, and K. E. Willcox. Proper orthogonal decomposition extensions for parametric applications in compressible aerodynamics. In *21st Applied Aerodynamics Conference, AIAA Paper 4213*, 2003.

[2] P. Holmes, J. L. Lumley, G. Berkooz, and C. W. Rowley. *Turbulence, coherent structures, dynamical systems and symmetry*. Cambridge University Press, 2012.

[3] N. Aubry, P. Holmes, J. L. Lumley, and E. Stone. The dynamics of coherent structures in the wall region of a turbulent boundary layer. *Journal of Fluid Mechanics*, 192:115–173, 1988.

[4] B. R. Noack, K. Afanasiev, M. Morzynski, G. Tadmor, and F. Thiele. A hierarchy of low-dimensional models for the transient and post-transient cylinder wake. *Journal of Fluid Mechanics*, 497:335–363, 2003.

[5] E. Caraballo, J. Little, M. Debiasi, and M. Samimy. Development and implementation of an experimental-based reduced-order model for feedback control of subsonic cavity flows. *ASME Journal of Fluids Enginnering*, 129 (7):813–824, 2007.

[6] A. Sinha, A. Serrani, and M. Samimy. Initial development of reduced-order models for feedback control of axisymmetric jets. *International Journal of Flow Control*, 2(1):39–60, 2010.

[7] P. A. LeGresley and J. J. Alonso. Investigation of non-linear projection for pod based reduced order models for aerodynamics. In *39th Aerospace Sciences Meeting, AIAA Paper 926*, 2001.

[8] D. Alonso, J. M. Vega, and A. Velazquez. Reduced-order model for viscous aerodynamic flow past an airfoil. *AIAA Journal*, 48(9):1946–1958, 2010.

[9] R. Zimmermann and S. Görtz. Non-linear reduced order models for steady aerodynamics. *Procedia Computer Science*, 1(1):165–174, 2010.

[10] D. Alonso, J. M. Vega, A. Velázquez, and V. de Pablo. Reduced-order modeling of three-dimensional external aerodynamic flows. *Journal of Aerospace Engineering*, 25(4):588–599, 2012.

[11] D. Amsallem, M. Zahr, Y. Choi, and C. Farhat. Design optimization using hyper-reduced-order models. *Structural and Multidisciplinary Optimization*, 51(4):919–940, 2015.

[12] J. L. Lumley. The structure of inhomogeneous turbulent flows. In A. M. Yaglom and V. I. Tatarsky, editors, *Atm. Turb. and Radio Wave Prop.*, pages 166–178. Nauka, Moscow, 1967.

[13] L. Sirovich. Turbulence and the dynamics of coherent structures, Parts I-III. *Quarterly of Applied Mathematics*, XLV(3):561–590, 1987.

[14] *ANSYS Fluent 15 Users Guide*. ANSYS, Inc., Lebanon, NH 03766, USA, 2015.

[15] C. W. Rowley. *Modeling, Simulation, and Control of Cavity Flow Oscillations*. PhD thesis, California Institute of Technology, 2002.

[16] J. D. Anderson. *Computational fluid dynamics: the basics with applications*. McGraw-Hill Inc., 1995.

[17] D. Alonso, A. Velazquez, and J. M. Vega. Robust reduced order modeling of heat transfer in a back step flow. *International Journal of Heat and Mass Transfer*, 52(5):1149–1157, 2009.

[18] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[19] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit (4th ed.)*. Kitware, 2006.

[20] *ANSYS Fluent 12 Users Guide*. ANSYS, Inc., Lebanon, NH 03766, USA, 2012.

[21] F. Palacios, M. R. Colonno, A. C. Aranake, A. Campos, S. R. Copeland, T. D. Economon, A. K. Lonkar, T. W. Lukaczyk, T. W. R. Taylor, and J. J. Alonso. Stanford university unstructured (su2): An open-source integrated computational environment for multi-physics simulation and design. In *51st AIAA Aerospace Sciences Meeting, AIAA Paper 0287*, 2013.