Software Design of the HyperSolve CFD Tool

Clark Pederson *
NASA Langley Research Center

March 27, 2024

Contents

*With thanks to Kyle Thompson, Andrew Hinkle, Matthew O'Connell, Chip Jackson, and Bil Kleb

1 Aerothermal Analysis at NASA

NASA missions require high-fidelity simulation capabilities for flows ranging from subsonic to hypersonic. For crewed space missions and planetary missions, CFD is used for both aerodynamic and aerothermodynamic design and analysis of entry vehicles. It can be very difficult to run physical experiments with Mach numbers or chemistry that matches flight conditions, especially for missions to atmospheres like Venus, Titan, or Jupiter. CFD plays a pivotal role in these missions by making predictions that would be difficult or impossible to measure experimentally. For the higher speed regimes, CFD codes must support the physical models of high-energy physics phenomena, including chemical dissociation and ionization at extreme temperature, gas-surface interaction, shock layer kinetics, and radiation.

There are multiple codes used at NASA with some or all of these capabilities supported. LAURA (Gnoffo, 1990) and DPLR (Wright et al., 1998) are multiblock structured-grid finite-volume solvers specializing in aerothermal analysis of vehicles involved in the entry phase of Entry, Descent and Landing (EDL) applications. Both LAURA and DPLR have been used to support NASA planetary missions, and both solvers include r-adaptation algorithms to align the outer boundary of the mesh to a bow shock and adjust the spacing near solid walls.

FUN3D (Anderson and Bonhaus, 1994) is a node-centered unstructured finite-volume solver used for simulations ranging from subsonic to hypersonic flows. The high-energy physics path of FUN3D was implemented in the early 2000s by porting most of the physical modeling from LAURA to FUN3D, and demonstrated comparable accuracy to LAURA through a suite of functional equivalence tests (Gnoffo et al., 2013).

US3D is a cell-centered unstructured-grid finite-volume solver specializing in high-energy flow physics simulations, developed at the University of Minnesota (Candler et al., 2015). The US3D code inherits much of the same capabilities as the DPLR code, and has most recently been used for simulations requiring high-resolution, such as transition prediction of the BOLT vehicle (Butler et al., 2022).

CHEM (Luke et al., 2003) is an unstructured-grid reacting flow solver implemented in the Loci framework (Luke and Reese, 1999) that is developed at the University of Mississippi, usually referred to as Loci/CHEM. Loci/CHEM has been used to support NASA missions, including the NASA Space Launch System (Hall et al., 2019; Boustani et al., 2023).

1.1 Improving Geometry Fidelity for CFD Simulations at NASA

NASA projects often involve trade studies or iterative design, where several new geometries are proposed and evaluated. The aeroheating must be evaluated for these new geometries to estimate the thermal protection system (TPS) necessary to protect the internal components or payload. Questions will frequently arise about the impact of protuberances or cavities in a vehicle. For example, after the Artemis I mission, inspections of the heatshield for the Orion crew module "revealed unexpected melting and erosion that created a gap leading to increased heating inside the bolt during Orion's reentry" (NASA Office of Inspector General (2024)). The impact of these gaps needed to be assessed before the Artemis II mission to ensure crew safety.

mesh generation is an important part of CFD, and high-quality solutions usually require high-quality meshes. "High quality" can be defined in several ways for hypersonic aerodynamics. First, the mesh must be either aligned with shocks or have fine mesh resolution around the shocks to minimize errors in the numerical fluxes. As shown by Candler et al. (2007) and McCloud (2017), shock alignment can have significant impacts on solution accuracy. Second, the mesh must be fine enough to resolve any flow phenomena such as boundary layers and isentropic expansions, as well as any other gradients in the solution that will impact the quantities of interest. Finally, the boundary layer should have a relatively fine wall-normal spacing to resolve the viscous sublayer.

To achieve these goals, aerothermal CFD analysis has traditionally used structured meshes or hexahedral-dominant meshes that are manually crafted. These meshes allow the user fine-grained control over alignment of cells, wall-normal spacing, mesh smoothness, and shock alignment. Structured-grid CFD solvers such as LAURA, and DPLR have mesh adaptation that aligns the mesh with a bow shock and adjusts the wall-normal spacing; these adaptation techniques greatly improve solution quality but are limited in their scope. For example, these techniques do little to address resolution inside cavities or in wake regions. These techniques also do not adjust the distribution of cells on the surface of a geometry, which is usually manually crafted and left fixed during mesh adaptation.

These types of structured or hexahedral-dominant meshes are difficult and time-consuming to create for complex geometries. The manual creation also leads to human errors, which means that several iterations of trial and error are required to arrive at a reasonable mesh. In one example, Thompson et al. (2023b) showed that multiblock structured meshes could be employed with an improved mesh adaptation algorithm in the LAURA code to accurately predict surface heat flux on the aft region of the LOFTID vehicle. While this approach was successful, it required significant effort and expertise to generate a suitable structured block topology for the LOFTID geometry.

Another approach for meshing is to use what is called a "simplex mesh." A simplex mesh consists of a single element type: triangles in 2D or tetrahedra in 3D. These meshes are tetrahedral all the way from the freestream boundaries to the wall; there is no structured boundary layer mesh nor prisms at the wall. These meshes are attractive for their simplicity and their direct connection to a continuous metric field that describes an optimal mesh spacing (see Loseille et al. (2007); Loseille and Alauzet (2011a,b)).

New mesh adaptation techniques are needed to improve the geometry fidelity in high-energy flow simulations in the next generation of NASA CFD tools without also requiring time-consuming, manual creation of meshes. The "CFD Vision 2030" published by NASA specifically states that, "Mesh generation and adaptivity continue to be significant bottlenecks in the CFD workflow" Slotnick et al. (2014). The report outlines a bold vision for CFD: "since a computational mesh is merely a means to enable the CFD simulation, ultimately the mesh generation process should be invisible to the CFD user or engineer." To meet this vision, the Sketch-to-Solution (S2S) was proposed to (semi-)automatically generate and adapt meshes based on a geometry definition of the domain Kleb et al. (2019). This approach has been successfully employed with the FUN3D code for aerodynamic analysis of vehicles with a high degree of geometry complexity. Because the S2S workflow generates simplex meshes, its application to aerothermal analysis has been more challenging. O'Connell et al. (2021) demonstrated that the VULCAN-CFD code has the ability to achieve accurate aerothermal predictions with the Sketch-to-Solution process

with sufficient mesh refinement.

Similar to the S2S workflow, Ekelschot and Brock (2022) coupled the US3D code with the ParMMG mesh adaptation library for entry vehicle wake-flow simulations. This approach also used prism elements to resolve the boundary layer and ParMMG to adapt the remaining domain using simplex elements; however, the accuracy of surface heat flux predictions using this method was not reported. While less automatic, McCloud (2017) demonstrated that a shock fitting approach combined with using a prismatic mesh to resolve the boundary layer significantly improved the surface heat transfer predictions with the Loci/CHEM code.

One may question whether a simplex element mesh is really necessary. Could one not use a mixed-element mesh similar to that proposed by McCloud (2017) to obtain accurate heating? In this category of mesh, the boundary layer region is meshed with high aspect-ratio triangular- or rectangular prisms. These layers are extruded in layers that are parallel to the wall and have tight wall-normal spacing. Pederson and Schoenenberger (2023) demonstrated that both simplex meshes and mixed-element meshes gave accurate predictions for aerodynamics. Nastac et al. (2022, 2025) have demonstrated accurate aeroheating predictions using mixed-element meshes on several test cases. However, there are several shortcomings to the current state-of-the-art using prismatic boundary layer meshes.

First, none of the techniques explored in these papers involve iterative adaptation of the prismatic boundary layer mesh. Once the prismatic boundary layer is generated, it remains frozen the rest of the adaptation process. As demonstrated by Pederson and Schoenenberger (2023), the meshes with different boundary layer resolution will give slightly different predictions, even as the mesh is refined outside the boundary layer. This same shortcoming can be seen in the transonic wing case tested by Nastac et al. (2025). A naive user might diagnose this as different mesh topologies "converging" to different answers, but the underlying problem is more subtle. Any mesh with finite resolution will have discretization error. By neither refining nor adapting the prismatic mesh in the boundary layer mesh, the portion of the discretization error from that "frozen" portion is fixed and will not decrease. Ideally, the boundary layer mesh should also be adapted and refined, but this introduces an additional set of independent variables (e.g., surface spacing or wall-normal spacing). Most studies combining a prismatic boundary layer with adaptive mesh refinement of the off-body cells do not demonstrate grid convergence of both regions of the mesh.

Even if the boundary layer mesh were iteratively adapted, additional difficulties arise in small cavities or corners where multiple advancing fronts collide as the boundary layer mesh is extruded from the surface. The regular pattern of aligned hexahedra will break down in these regions and mesh generation algorithms may introduce skewed cells, pyramids, or stair-stepping of different element types. These irregular regions will pose numerical difficulties for a CFD solver if the numerical schemes are not carefully designed. For this reason, using a prismatic boundary layer does not obviate the need for robust numerical schemes that work on irregular meshes.

Unstructured mesh adaptation techniques show promise for semi-automated mesh generation and adaptation for high-fidelity analysis of vehicles with a high degree of geometric complexity. However, aerothermal analysis using purely simplex meshes is still not routine; this motivated the creation of HyperSolve.

1.2 HyperSolve: Improved Aerothermal Analysis on Simplex Meshes

The HyperSolve CFD code is a three-dimensional finite-volume solver being used to investigate the efficacy of hyperbolic methods on unstructured meshes. The core mission of HyperSolve is to provide accurate surface heat transfer predictions for NASA Entry vehicles, with a focus on unstructured mesh adaptation. Development of HyperSolve began at the NASA Langley Research Center in 2018, and several papers have been published on various facets of the solver since its inception (Thompson and O'Connell, 2019; Thompson et al., 2023a; Kyle B. Thompson and Pederson, 2024).

The HyperSolve code is an edge-based finite-volume solver, which uses a variety of methods to achieve higher-order accuracy, specifically on simplex element meshes. HyperSolve has a wide variety of capabilities:

- 2nd/3rd-order edge-based, finite-volume discretizations
- In-core mesh adaptation for multiblock structured grids and unstructured meshes
- Thermally perfect gas and reacting flow in thermal nonequilibrium
- 1-equation and 2-equation RANS turbulence model variants
- Jacobian-Free Newton-Krylov nonlinear solver
- Eulerian and Lagrangian dilute particle phase solvers

Recently, Kyle B. Thompson and Pederson (2024) showed that the HyperSolve code produces nearly identical results as LAURA for a suite of high-energy cases when run on an identical mesh. Hinkle et al. (2023) extended the HyperSolve code to simulate dusty flow in the Martian environment by leveraging the plugin-based environment described in Section 4.3.

The next sections are organized to describe the software design and core numerical methods implemented in HyperSolve:

- Section 2 describes the numerical methods implemented in HyperSolve, with a focus on the spatial discretization
- Section 3 describes the nonlinear solver methods used to solve the HyperSolve discrete system.
- Section 4 describes the software principles and practices that have been found to be very effective during the development of HyperSolve.

2 HyperSolve Numerical Methods

In this section, the spatial discretization methods implemented in HyperSolve are described. As described in the Introduction, one of the primary goals of HyperSolve is to improve the accuracy of heat flux predictions on simplex methods, without greatly increasing the degrees-of-freedom compared to traditional finite-volume schemes.

To achieve high-resolution in space, HyperSolve employs a variety of economical methods, as described by Thompson et al. (2023a). When computing solution quantity gradients with a method that is quadratically exact (i.e., quadratic least squares), Katz and Sankaran (2011) showed that third-order accuracy can be achieved in the convective fluxes without needing to store second-derivatives. The resulting scheme is highly economical, because it attains third-order accuracy with the same number of degrees of freedom and quadrature points as a second-order finite-volume scheme. Thompson et al. (2023a) showed that the improvement afforded by this scheme is substantial for hypersonic flow around a sphere as compared to a second-order scheme.

2.1 Viscous Fluxes

Viscous fluxes are computed using a variety of techniques: alpha damping is sufficient for the momentum equations, while more advanced numerical techniques have been employed for thermal diffusion.

2.1.1 Alpha-Damping Viscous Fluxes

To discretize the viscous fluxes in the momentum equations, HyperSolve uses the alpha-damping scheme by Nishikawa et al. (2017).

Viscous fluxes can be expressed as a product of diffusivity and a gradient term. For a simple example, consider a diffusivity ν and a solution u. The viscous flux would be:

$$\vec{F} = -\nu \nabla u \tag{1}$$

This flux vector can then be projected onto the face n_{jk}:

$$\phi_{jk} = -\nu \nabla u \cdot \hat{n}_{jk} \tag{2}$$

where n_{jk} is a unit vector normal to the face between nodes j and k. A simple choice for a viscous scheme would be:

$$\phi_{jk} = -\nu \overline{\nabla u} \cdot \hat{n}_{jk}, \tag{3}$$

where $\overline{\nabla u}$ is the average of the gradients at the two nodes on each side of the face, x_j and x_k. Unfortunately, this simple and consistent scheme is not stable for skewed meshes, where $\hat{n}_{jk}$ is not aligned with with the unit vector connecting the two nodes, $\hat{e}_{jk} = (\vec{x}_k - \vec{x}_j)/|\vec{x}_k - \vec{x}_j|$.

Nishikawa (2010, 2011b) derived a general formula for 2nd to 4th order accurate viscous schemes on a compact stencil:

$$\phi_{jk} = -\nu \overline{\nabla u} \cdot \hat{n}_{jk} - \frac{\nu \alpha}{|\hat{e}_{jk} \cdot \hat{n}_{jk}|} \left[\frac{u_k - u_j}{\ell_{jk}} - \overline{\nabla u} \cdot \hat{e}_{jk} \right] \tag{4}$$

where $\ell_{jk} \equiv |x_k - x_j|$ is the distance between nodes and α is a free parameter. This scheme is second-order accurate for $\alpha \neq 0$.

Setting $\alpha = 4/3$ can yield 4th order accuracy for specific grid types, but this is achieved at the expense of monotonicity preservation. Setting $\alpha = 1$ is another common choice; for Cartesian, non-skewed meshes this choice results in a scheme identical to a central difference scheme since $\hat{e}_{jk} \cdot \hat{n}_{jk} = 1$.

Another choice is $\alpha = |\hat{e}_{jk} \cdot \hat{n}_{jk}|(\hat{e}_{jk} \cdot \hat{n}_{jk})$, which results in:

$$\phi_{jk} = -\nu \overline{\nabla u} \cdot \hat{n}_{jk} - \nu(\hat{e}_{jk} \cdot \hat{n}_{jk}) \left[\frac{u_k - u_j}{\ell_{jk}} - \overline{\nabla u} \cdot \hat{e}_{jk} \right] \tag{5}$$

This scheme has been used in several unstructured CFD codes (Weiss et al. (1997); Haselbacher (1999); Haselbacher and Blazek (2000)). Once again, it is identical to a central-difference scheme on Cartesian, non-skewed meshes.

2.1.2 Higher-Order Derivatives for the Viscous Fluxes

Several different approaches have been used for the viscous fluxes in the energy equation. Alpha-damping can be used, but it does not provide gradients that are second-order accurate. Lower accuracy in the temperature gradients directly impacts the accuracy of surface heat flux.

Early research efforts with HyperSolve focused on recasting 2nd-order PDEs as first-order systems of PDEs. The Navier-Stokes equations are formally a system of 2nd order PDEs. Nishikawa et al. (2011a; 2016; 2017) demonstrated how the Navier-Stokes equations can be discretized instead as a set of first-order PDEs with new equations for each gradient variable. For example, a simple diffusion equation:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} \tag{6}$$

can instead be discretized as the first-order system:

$$\frac{\partial u}{\partial t} = \nu \frac{\partial p}{\partial x} \tag{7}$$

$$\frac{\partial p}{\partial t} = \left(\frac{\partial u}{\partial x} - p \right) / T_r \tag{8}$$

where p relaxes to $\partial u/\partial x$ and T_r is the relaxation time. This new system of equations can be discretized using any method, including finite-volume or finite-element methods. Benefits include the following:

- Both the solution variables (such as u) and the gradients (such as $\partial u/\partial x$) are solved to the same order-of-accuracy. For a formally second-order code, this means that gradients can be second-order accurate.
- The removal of second-derivatives can speed up convergence.

Unfortunately, this technique would require an extra PDE for each gradient variable that enters the diffusive fluxes. For the compressible Navier-Stokes equations in 3D, that

translates to 20 equations. For a multi-species case, three extra PDEs are added for each species (not including the species conservation equation itself). It is easy to see how this approach becomes impractical for flows in thermal- and chemical-nonequilibrium.

This led Thompson et al. (2023a) to research a system of equations where only the temperature gradients are split off as a first-order system. This was called the "Reformulated Heat Flux System" or RHFS. Only three additional PDEs are required. This is more economical than the full first-order approach, but does not maintain the rapid convergence benefits. Thompson et al. (2023a) demonstrated that using this reformulated system produces accurate surface heat flux predictions on simplex element meshes adapted using the S2S process.

Recent research has shifted towards a technique proposed by Nishikawa (2024) called Implicit Edge-Based Gradients (IEBG). IEBG improves the accuracy of gradients, similar to the RHFS approach; the key difference is that instead of solving a time-dependent 1st order PDE for the gradient variables, the gradient variables are solved using an elliptic PDE. This is done by:

- Writing the energy equation as a first-order system of equations, just as in RHFS,
- Discretizing the first-order hyperbolic system with upwind schemes
- Dropping the time-derivative terms, which go to zero at steady-state.

Given solution values u_j at the nodes and gradient variables $\boldsymbol{g} = \nabla u$, we can linearly reconstruct the values of u at the face from the left and right side as in a U-MUSCL scheme (Burg (2005)):

$$u_L = \kappa\frac{u_j + u_k}{2} + (1-\kappa)(u_j + \frac{1}{2}\boldsymbol{g}_j \cdot \boldsymbol{\ell}_{jk}) \tag{9}$$

$$u_R = \kappa\frac{u_j + u_k}{2} + (1-\kappa)(u_k - \frac{1}{2}\boldsymbol{g}_k \cdot \boldsymbol{\ell}_{jk}) \tag{10}$$

where $\boldsymbol{\ell}_{jk} = x_k - x_j$ is the displacement vector between x_k and x_j, and κ is a real-valued parameter, usually between 0 and 1. The gradients are then solved in an implicit system as:

$$\frac{1}{V_j}\sum_{k\in N_j}\frac{1}{2}\left[c\boldsymbol{g}_j + (2-c)(\boldsymbol{g}_k - \nabla\boldsymbol{g}_j^{LSQ}\cdot\boldsymbol{\ell}_{jk})V_{jk}\right] = \frac{1}{V_j}\sum_{k\in N_j}\frac{u_L+u_R}{2} \tag{11}$$

where N_j is the set of edge-connected neighbors of j, $V_j = \sum_k V_{jk}$ is the median-dual volume around node j, V_{jk} is portion of the volume between j and k, $\boldsymbol{g}_j^{LSQ}$ is the gradient calculated using an unweighted linear least-squares technique over the neighbors k_j, The parameter c is a parameter in the source-term quadrature scheme, and has traditionally been set to $c = (3D+4)/(D+2)$ where $D = 2$ for two-dimensional meshes and $D = 3$ for three-dimensional meshes. For more details, see the paper by Nishikawa (2024).

2.2 Convective Flux

The Roe scheme is part of a larger category of Flux-Difference Splitting (FDS) schemes. These schemes can be expressed as the sum of a consistent term and a "correction" or

dissipative term. For a flux F_{jk} at a face between j and k, the primitive variables w can be linearly reconstructed at the face as:

$$w_L = \kappa\frac{w_j + w_k}{2} + (1-\kappa)(w_j + \frac{1}{2}\boldsymbol{g}_j \cdot \boldsymbol{\ell}_{jk}) \tag{12}$$

$$w_R = \kappa\frac{w_j + w_k}{2} + (1-\kappa)(w_k - \frac{1}{2}\boldsymbol{g}_k \cdot \boldsymbol{\ell}_{jk}), \tag{13}$$

where κ is a free parameter. A typical FDS scheme can be then be expressed as:

$$F_{jk} = \frac{1}{2}\left(F(w_L) + F(w_R)\right) - \frac{1}{2}|A(w_L, w_R)|(u_R - u_L) \tag{14}$$

where F is the inviscid flux function, $|A|$ is a Jacobian matrix of the flux F with respect to the conserved variables u, u_j and u_k are the solution values at the nodes, and u_L and u_R are the conservative variables computed from w_L and w_R.

The reconstruction can cause problems for A if the linear reconstruction results in unrealizable states, such as negative density. Falling back on first-order accuracy ($w_L = w_j$ and $w_R = w_k$ with no higher-order reconstruction) is common but can be avoided using the following scheme.

Nishikawa (2020) proposed calculating the Jacobian matrix using the nodal values, instead of reconstructed values:

$$F_{jk} = \frac{1}{2}\left(F(w_L) + F(w_R)\right) - \frac{1}{2}|A(u_j, u_k)|(u_R - u_L) \tag{15}$$

This permits unrealizable states (such as negative density) in the reconstructed states (such as w_R and w_L) since they no longer enter the Jacobian. While unrealizable states may still appear in u_R and u_L, the only place these terms appear is in the difference of $u_R - u_L$.

While this technique is often described as a "robust Roe" scheme, it can be applied to other Flux-Difference Splitting schemes such as HLL schemes including HLLE++ (Tramel et al. (2009)).

In pratice, HyperSolve combines the robust Roe technique with a flux extrapolation (see Katz and Sankaran (2011); Liu and Nishikawa (2016)) to arrive at the following flux form:

$$F_{jk} = \frac{1}{2}\left(F_L + F_R\right) - \frac{1}{2}|A'(u_j, u_k)|(w_R - w_L) \tag{16}$$

where A' is the Jacobian with respect to the primitives (w) and the fluxes themselves are extrapolated:

$$F_L = F_j + \frac{1}{2}\left(\frac{\partial F}{\partial w}\right)_j \boldsymbol{g}_j \cdot \boldsymbol{\ell}_{jk} \tag{17}$$

$$F_R = F_k - \frac{1}{2}\left(\frac{\partial F}{\partial w}\right)_k \boldsymbol{g}_k \cdot \boldsymbol{\ell}_{jk} \tag{18}$$

where $\partial F/\partial w$ is the Jacobian of the flux function F with respect to the primitive variables w.

3 Nonlinear Solver

Thompson and O'Connell (2019) showed that deep nonlinear convergence is often required for accurate surface heat transfer predictions. The efficient solution of any partial differential equation depends upon the nonlinear solver algorithm. For example, the LAURA code employs a nonlinear Gauss-Seidel relaxation method to achieve steady state quickly. The DPLR and US3D solvers employ a linear version of the technique that involves multiple Jacobi sweeps in the linear solver. Meshes that result from automatic mesh adaptation processes, such as S2S, typically require a more robust solution technique than body-fitted structured meshes. Thompson and O'Connell (2019) and O'Connell et al. (2021) have found that Newton-Krylov solvers are a more robust solution than relaxation-based solvers for solving these adapted meshes. Newton-Krylov solvers have also become popular as so-called "strong solvers" for higher-order methods, because they are able to achieve steady solutions that other methods cannot. This section describes the Jacobian-Free Newton-Krylov (JFNK) nonlinear solver method, as well as the linear preconditioning methods used by HyperSolve.

3.1 Jacobian-Free Newton-Krylov

Knoll and Keyes (2004) provides an excellent survey of Jacobian-free Newton-Krylov (JFNK) methods applied to a wide variety of problems. The JFNK method is essentially summarized as computing the Jacobian-update matrix-vector product via some form of Fréchet derivative. A real-valued approach computes this matrix-vector product as

$$\frac{\partial R}{\partial Q}\Delta Q = \frac{R(Q + \epsilon\Delta Q) - R(Q)}{\epsilon} \tag{19}$$

where ϵ must be carefully chosen to avoid excessive subtractive cancellation error. Alternatively, a dual-number approach, such as the implementation proposed by Fike and Alonso (2011), can be applied to eliminate the sensitivity to ϵ; however, this approach does incur a significant performance penalty compared to Eq. 19. Both Thompson and O'Connell (2019) and Damm et al. (2023) showed that the JFNK method can outperform the relaxation methods in high energy flow simulations, making it a highly competitive nonlinear solver technique.

3.2 Linear Preconditioning

Unfortunately, the linear system produced by a hypersonic CFD code usually has large condition numbers and is difficult to solve. An effective preconditioner is essential to achieve good performance from the Krylov method. Typically, the preconditioner involves computing and storing an approximate form of the Jacobian matrix, which should use an automatic differentiation method when possible to avoid unintentional errors (described in Section 3.3). This leads to an apparent contradiction: the "Jacobian-Free" Newton-Krylov method may still require a Jacobian, even if that approximate Jacobian is only used for the preconditioner. However, the requirements for effective preconditioning can be separated from the requirements for deep convergence of a Newton solver; the approximate Jacobian can be constructed solely for the design constraints of preconditioning. The HyperSolve

CFD code uses the Enigma library (Thompson and O'Connell, 2019) to employ a JFNK method with an additive Schwarz method (ASM) as the default preconditioner.

3.3 Automatic Differentiation

Most modern CFD software are built upon implicit time integration methods. These methods require some form of (often approximate) Jacobian matrix to be constructed as part of the linear system to be solved at each timestep. A significant amount of time is wasted debugging the derivatives in a Jacobian matrix that have been derived or implemented by hand. As an alternative to hand coding the Jacobian entries, automatic differentiation techniques can be used to efficiently and accurately compute the Jacobian of a function. A downside of automatic differentiation is that its performance is often lackluster compared to the hand-coded implementation. In this section, we present two techniques to significantly improve the performance of automatic differentiation: expression templates and SIMD vectorization.

3.3.1 Expression Templates

Using a naive object-oriented approach when implementing the automatic differentiation library can result in poor performance compared to implicitly programming the Jacobian derivatives. When using a language that supports generic programming via templates (e.g., C++), an approach known as expression templates can be used to significantly improve the performance of automatic differentiation. Expression templates minimize the number of times classes are instantiated by building expression trees at compile time that use lazy evaluation to compute the derivative values. By using lazy evaluation, multiple operations (e.g., addition, subtraction, log, exp, etc.) can be "chained" together into a single class. These operations are not evaluated until the assignment operator is added to a chain, which significantly reduces the number of intermediate class instantiations. An excellent description of using expression templates for automatic differentiation is described by Galbraith et al. (2015).

3.3.2 SIMD Vectorization

In practice, forward-mode differentiation is used for the evaluation of Jacobian matrices when the number of dependent and independent variables are equal (i.e., a square Jacobian Matrix). The automatic differentiation class will typically hold a set number of derivatives, which are propagated through a function. The result of the function will contain these derivatives, which are the final Jacobian values.

Because there are multiple derivatives being computed simultaneously, SIMD operations can be employed to improve the performance of the derivative computation. Ideally, this would be as simple as applying compiler derivatives to vectorize a loop; however, experience has shown that compilers will not generate optimal code with this approach. Instead, a SIMD class approach can be used to generate vector instructions explicitly. Using a SIMD library, such as the one provided by the Kokkos performance portability library (Edwards et al., 2014), has been found to decrease the time required to compute the Jacobian with automatic differentiation by more than 50% as compared to without using a SIMD class.

3.4 Pseudo-Transient Continuation

Newton or quasi-newton techniques for solving a linear system can be powerful, but they can also be expensive. Sometimes other techniques can be more efficient when dealing with initial transients, when the solution is far from the converged solution to the discretized equations. This motivates the use of globalization techniques such as pseudo-transient continuation (Kelley and Keyes (1998); Gropp et al. (2000); Mavriplis (2021)).

For example, one can perform time-marching using a Backward Euler method.

$$\left(\frac{V_i}{\Delta t}\delta_{ij} + \frac{\partial R_i}{\partial Q_j}\right)\Delta Q_j = -R_i(Q^n) \tag{20}$$

where V_i is the volume of a computational cell and Δt is a timestep. This is a common temporal discretization in CFD codes used to advance the solution to a steady state. When a strong linear solver is not available, such CFD codes will often use a relatively low upper limit (such as a CFL number of 5) on the timestep and simply iterate in the hope that the solution will reach a converged state.

With small timesteps, a Backward Euler discretization can be useful for dealing with stiffness induced by initial transients. As the timestep grows very large, this discretization recovers a Newton scheme because the $V/\Delta t$ term goes to zero. Using an appropriate ramping technique, the timestep Δt can be adjusted from small timesteps during the initial nonlinear iterations and large timesteps during the later nonlinear iterations.

3.5 Estimating Timesteps

In order to improve the efficiency of the nonlinear solve, the solver should maximize the "timesteps" in the pseudo-transient continuation. Some regular regions of the flow may allow for large timesteps without numerical difficulties. However, other regions may be more stiff or be close to unrealizable behavior.

One specific difficulty encountered by compressible codes is unphysical "undershoots" in density, temperature, or species fractions. An undershoot occurs when the nonlinear solver decreases the solution below the value of the exact solution to the discretized equations. This may be a transient behavior that is later corrected by the nonlinear solver, but even transient states can cause difficulties for numerical methods. If the exact solution is close to zero, an undershoot can lead to densities, temperatures, or mass fractions below zero. Luke et al. (2003) proposed a technique for limiting the timestep to prevent unrealizable undershoots in density or temperature. The underlying principles can be described as follows:

Consider the value of density, ρ, at node i and timestep n. A forward Euler technique would result in the following update:

$$\frac{\rho_i^{n+1} - \rho_i^n}{\Delta t^n} = -R_{i,\rho}^n \tag{21}$$

where $R_{i,\rho}$ is the residual for the density equation at node i. Using this equation, one can set $\rho_i^{n+1} = 0$ to estimate the largest timestep possible that would result in a realizable state:

$$\Delta t^n \leq \rho_i^n / R_{i,\rho}^n \tag{22}$$

Often an adjustable limit between 0 and 1 is added to this scheme as a safety factor.

There are many possible variations on this technique. With HyperSolve, robustness is improved if timesteps are limited to keep the temperature positive (or both temperatures in a two-temperature system). Luke et al. (2003) proposed using an implicit method for calculating the timestep.

This timestep estimation approach is similar to the Switched Evolution Relaxation (SER) technique by Mulder and Van Leer (1985), where the timestep was set to be inversely proportional to the residual:

$$\Delta t^n \propto 1/||R^n|| \tag{23}$$

The key difference is that Mulder and van Leer appear to have used a norm over all the equations. This could be more restrictive than the current approach, since quantities like momentum can be positive or negative; the momentum residuals do not provide information about the realizability of a solution.

The current timestep limit is also similar to but distinct from "percent update limiting" or a "limit on the fractional update," where updates to a solution are limited to a percentage of the current solution value. The key advantage of timestep limits is as follows: A limit on an update is applied after a linear solve to reign in a large update. This may be too late; if the linear solver is burdened with too large of a timestep, a percent update limit cannot reduce the stiffness of the linear system. Taking large timesteps and then limiting the updates can result in wasted work by the nonlinear solver. Timestep limiting is applied *before* the linear solve and can reduce the stiffness of the linear system of equations.

4 Software Development Practices and Principles

The reality is that many research scientists are not software engineers by training; they are either unaware of or do not prioritize long-term software craftsmanship. New software will *always* grow in scope to meet new demands, and that broader scope must be developed carefully.

A simplified software life-cycle can be described as follows:

1. A new, simple prototype is created.
2. Capabilities are added to expand the scope.
   (a) The complexity of the software increases
   (b) Updates and maintenance become more difficult.
   (c) Steps (a) and (b) repeat until...
3. Developers become disillusioned with the codebase.
4. A new, simple prototype is created to replace the previous codebase.

Some of this software cycle is inevitable: all codebases will eventually be replaced, and sometimes better codebases are more easily replaced than poorly understood codebases. Nevertheless, an understanding of this cycle is important to avoid wasted rework. If developers manage complexity well, they can benefit from years of experience that have been invested into the software. On the other hand, if developers simply write new code without paying attention to the growing complexity, the software will rapidly become unmaintainable.

Sustainable software development does not come easily. In accordance with the second law of thermodynamics, fighting disorder takes work. This work can include:

- Defining abstract interfaces
- Writing unit tests
- Refactoring
- Deleting unused code

These practices are rarely taught in a numerical methods classes, but they are an important part of managing a complex code-base that is continually growing in scope.

In this section, some key concepts that have significantly improved the software development process are described: the SOLID principles, test-driven development, and a plugin-based environment.

4.1 Human Scaling

Research software is the product of many different subject matter experts in diverse fields. For example, a hypersonic CFD code may require experts in nonlinear solvers, finite-rate chemistry, turbulence, numerical methods, and mesh generation. Each of these are deep research fields; only very rarely is someone an expert in most of these fields. While software development gurus often promote "cross-functional" teams, this is very difficult for cutting-edge research codes. Software development should be planned so each expert can contribute meaningfully in their own subject area, without needing to know the minutiae of other research areas.

In high-performance computing, it is typical to measure how a code "scales" as more compute resources are added. A code scales well if each new CPU or GPU can be used efficiently. One can also strive for **strong human scaling**, where new developers can work efficiently without slowing the rest of the team. Just as with parallel computing, strong human scaling requires careful planning about what needs to be communicated and when.

In order to efficiently use researchers with deep subject matter expertise, it is useful to design decoupled software that communicates with abstract interfaces. A poor software architecture would have each researcher trying to carefully weave their code into other researcher's code and praying that there are no code conflicts. On the other hand, a proper plugin-based architecture can allow experts to independently develop new software capabilities. When their code needs to interact with someone else's code, they can agree on abstract interfaces that communicate inputs and outputs clearly.

4.2 Software Design: SOLID Principles

The SOLID principles are concepts put forth by Martin and Coplien (2009) that promote "clean code", i.e., well-organized and understandable code which promotes sustainable long-term development. SOLID stands for

- **Single responsibility principle**: There should never be more than one reason for a class to change.
- **Open-closed principle**: Software entities should be open for extension, but closed for modification.
- **Liskov substitution principle**: Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
- **Interface segregation principle**: Clients should not be forced to depend upon interfaces that they do not use.
- **Dependency inversion principle**: Depend upon abstractions, not concrete implementations.

Adhering to these principles while developing code has been found to substantially reduce the rigidity and fragility of new software. HyperSolve's development standards seek to incorporate all of these principles. In particular, the dependency inversion principle is a key feature which will be discussed in more detail in Section 4.3.

4.3 Dependency Inversion Principle: Plugin Architecture

The Dependency Inversion Principle replaces the tight coupling between software components shown in Figure 1 by inserting an abstract interface between the high-level and low-level software components, as shown in Figure 2. Using this principle promotes the

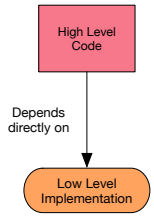

Figure 1: Typical software strategy: high-level concepts depend directly on low-level details.

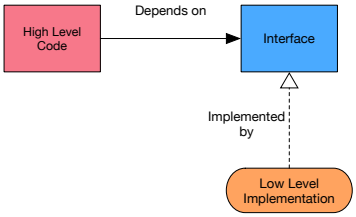

Figure 2: Dependency inversion: A high-level concept depends on the abstract interface and a low-level detail implements the interface.

use of software "plugins": software components that add functionality to the application **without having to modify the source code of the application**. O'Connell et al. (2018) describes a plugin framework to promote reusability of software components in many applications related to CFD. The HyperSolve CFD code heavily uses this framework to reduce code fragility and emphasize the separation of concerns. This framework is similar to that of the CoolFluid CFD project (Lani et al., 2013). By using plugins, the source code dependencies between software components are eliminated at compile-time. Instead, dependencies are on abstract classes, which the plugin modules implement. These plugins are loaded at runtime, enabling a highly flexible system that can be reconfigured rapidly by both developers and users. In this section, we describe the key concepts of this plugin architecture at both the high- and low-level implementation of the HyperSolve code.

4.3.1 Plugin Loading

Plugins in the HyperSolve code are shared libraries written in C++ with specific functions exposed that implement an abstract class interface defined in a central location. The plugin is loaded at runtime using the `dlopen` and `dlsym` functions to bind the function in the library to application scope. Typically, this function is a factory method that produces a C++ object of abstract type, also defined a central location. Applications depend on the abstract types, rather than the concrete implementation, which decouples any source code dependencies between the application and plugin provider. Reading and partitioning a mesh from disk is an example where plugin architecture can be used effectively to decouple software dependencies. The abstract concept of a mesh is defined, and the plugin produces an object that fits the mesh abstraction. Each plugin may use a different algorithm (e.g., recursive bisection, k-way partitioning, etc.) to compute the part vector that produces a partitioned mesh, but the details of that algorithm are hidden from the application, as they should be.

4.3.2 Self-registering Subcommands

The concept of self-registering objects is an effective way to implement object creation without long and complicated and fragile if-statement logic. Plugins can be self-registering by adopting a naming convention and installing them to a predetermined location so that they are easily discoverable. For example, if a plugin is named according to the convention `<application>_<subcommand>.so`, then a simple find command can easily parse all available subcommands for a given application. This software pattern is used by the HyperSolve development team to effectively group a set of applications such that they are easily discoverable and to automatically provide documentation on their usage. The basic design is as follows:

1. Define a C++ abstract base class for an application subcommand.

```
class SubCommand {
  public:
    virtual std::string description() const = 0;
    virtual void run(CommandLineMenu m) = 0;
    virtual CommandLineMenu menu() const = 0;
    virtual ~SubCommand() = default;
};

```

2. Implement the concrete subcommand class

```
class MeshConverter : public SubCommand {
  public:
    std::string description() const {
        return "Convert a mesh file format based on extension."
    }
    void run(CommandLineMenu m) {
        // Add implementation details here
    }
    CommandLineMenu menu() const {
        CommandLineMenu m;
        m.addArgument("-m", "Input mesh filename");
        m.addArgument("-o", "Output mesh filename");
        return m;
    }
};
extern "C" {
    std::shared_ptr<inf::SubCommand> getSubCommand() {
        return std::make_shared<MeshConverter>();
    }
}
```

3. Build the concrete class as a shared library named `hypersolve_convert.so` in a predetermined plugin location.

4. Implement an application program (i.e., `hypersolve`) to load any subcommand found in the plugin location at runtime, based on the user input. A minimal example could look like the following:

```
int main(int argc, const char* argv[]) {
    auto args = getCommandLineArgs(argc, argv);
    std::string plugin_name = "hypersolve_"
                            + getSubcommandFromArgs(args)
                            + ".so;"
    auto subcommand = loadSubcommandPlugin(plugin_name);
    auto menu = subcommand->menu();
    menu.parse(args);

    int exit_code = 0;
    if (menu.helpRequested()) {
        menu.printHelp()
    } else {
        try {
          runner.run(m);
        } catch(std::exception& e) {
          // add error handling logic here...
          exit_code = 1;
        }
    }

    return exit_code;
}
```

Users can now execute the command with a help option to get documentation

```
$ hypersolve convert --help
sub-command: convert
description: Convert a mesh file format based on extension.
options:
  -m <value> [REQUIRED]
        Input mesh file

  -o <value> [REQUIRED]
        Output mesh filename
```

Using this pattern, all command line parsing can be implemented once in the application program, which greatly simplifies the implementation of each subcommand. More importantly, this approach allows new developers to easily add capabilities and utilities to a program without having to understand the details of a how a monolithic application operates.

4.4 Alternatives to Virtual Dispatch

Plugins and abstract base classes are sometimes criticized because they use something called “virtual dispatch” to resolve function pointers at runtime. These extra operations result in code that runs slightly slower than similar code that has no base classes and hard-coded, compile-time behavior. Some programmers view this as a non-starter: They want the absolute best performance and do not want virtual dispatch at all.

A desire for top-level performance is not an excuse for poor software architecture. If one *must* have top-level performance, there are alternatives to abstract base classes where

abstractions can be handled at compile-time instead of runtime. These are techniques are referred to as static dispatch (as opposed to virtual dispatch through virtual functions).

One simple example is a template-based strategy design pattern. In the strategy design pattern, there are several possible choices for a step in an algorithm. These choices are encapsulated as independent objects that can be used interchangeably (recall the Dependency Inversion Principle). Creating different strategies can be done at runtime using templates.

Start with a context. For example, we want to calculate the production of turbulent kinetic energy in a RANS model but we want to leave the method for calculating production flexible. In this example, we have a class "SST" that depends on an abstract "ProductionModel" with a function "calc":

```
template <typename ProductionModel>
class SST {
 public:
  SST() = default;
  double calculateProduction(double k,
                             double omega,
                             double shear,
                             double vorticity) {
    const double eddy_viscosity = k / omega;
    return production.calc(eddy_viscosity, shear, vorticity);
  };
 private:
  ProductionModel production;
};
```

Next, define different strategies as structs:

```
struct KatoLaunderProduction {
  double calc(double eddy_viscosity,
              double shear,
              double vorticity) {
    return eddy_viscosity * shear * vorticity;
  }
};
struct VorticityProduction {
  double calc(double eddy_viscosity,
              double shear,
              double vorticity) {
    return eddy_viscosity * vorticity * vorticity;
  }
};
```

Finally, write the code that uses both the context and the strategies:

```
double k = 0.1; double omega = 10.0;
double shear = 1.0; double vorticity = 0.1;

SST<VorticityProduction> modelA;
auto Pk = modelA.calculateProduction(k, omega, shear, vorticity);

SST<KatoLaunderProduction> modelB;
Pk = modelB.calculateProduction(k, omega, shear, vorticity);
```

This example is intended solely as an example of how templates can be used to provide abstraction that is handled at compile-time, rather than runtime. For legal reasons, it is not a portion of actual HyperSolve code. There are many other related techniques, such as the Curiously Recurring Template Pattern (CRTP), that allow for abstraction without pointers or virtual interfaces.

There are some inherent trade-offs. Static dispatch means that the code depends on all possible variants being defined at compile-time. The developer cannot write code that depends on a yet-undefined concrete implementation. Therefore, templates and compile-time abstractions can be used for the inner workings of a plugin, but they cannot be used to define a plugin. If virtual dispatch is used instead of compile-time dispatch, the different parts of the CFD solver only need to know about the abstract interface; discovery of concrete implementations can be delayed until runtime.

Due to these trade-offs, abstract base classes and plugins are favored by the HyperSolve development team over compile-time abstractions. Nevertheless, static dispatch has been used successfully in limited cases, such as the expression templates in automatic differentiation. This section is provided merely as a demonstration that speed and abstraction are not mutually exclusive.

4.5 TDD: Test Driven Development

Unfortunately, software is often not tested until the last step in the development process. This often leads to nearly incessant amounts of debugging newly written software and its unintended side effects. Test-driven development (TDD), a concept credited to Beck (2002), advocates that the software test should be written as the first step. A robust unit testing framework should be the first thing stood up in a new software package. Writing a unit test before implementing a new feature might seem unnecessarily tedious; however, it has been found to significantly reduce the overall amount of time required by the programmer to complete the feature as compared to a "test last" approach. Unit tests are also important for shared codebases with many developers because they describe the behaviors and features that other developers want to preserve.

Test-driven development is not only important when developing new codes; it is also an important part of supporting legacy codes. Fowler (2018) describes a principled approach to supporting pre-existing codebases without adequate test coverage. Before refactoring, porting code to new hardware, or adding new features, it is important to identify existing functionality that you want to preserve. The recommended strategy is to write unit tests that encapsulate that functionality. Once the tests are in place, it is straightforward to test if the code changes have unintended impacts on existing features. This is not always easy, since many legacy codebases do not have a modular software architecture that allows units of code to be tested in isolation. However, investments in unit testing during development usually payoff in less time spent debugging code later.

5 Summary

Recent efforts at NASA have focused on developing advanced unstructured mesh adaptation techniques and discretization technologies that reduce the burden of mesh generation on the CFD practitioner. The HyperSolve CFD code focuses specifically on leveraging advanced numerical methods to enable accurate aerothermal predictions for NASA missions with high-fidelity geometry requirements. HyperSolve uses economical discretization methods to achieve higher-order accuracy on adapted simplex element meshes. Notably, this is achieved without requiring mixed-element meshes with prisms in the boundary layer. A JFNK nonlinear solver with nonlinear preconditioning is employed to solve the HyperSolve discretization efficiently. Automatic differentiation, using expression templates and SIMD instructions, is highlighted as an essential tool for the correct and efficient implementation of the Jacobian matrix required for preconditioning the linear system.

A focus on software architecture has enabled the rapid development of the HyperSolve code. Adherence to the SOLID principles promotes code that is both readable and maintainable. Specifically, the dependency inversion principle encourages the composability of features without creating highly fragile and rigid software. The dependency inversion principle is also used to enable the self-registration of subcommands, a technique that simplifies the organization and documentation of the software's capabilities. Development of HyperSolve is ongoing at the NASA Langley Research Center. Version 1.0 was released in 2025 and is available to US persons at `https://software.nasa.gov/`.

References

Anderson, W. K. and Bonhaus, D. L. (1994). An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23(1):1–22.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman, Amsterdam.

Boustani, J., Applebaum, M., Eppard, W., Steva, T., and Hall, L. H. (2023). A computational study of plume modeling for space launch system abort scenarios. AIAA Paper 2023-0239.

Burg, C. (2005). Higher Order Variable Extrapolation for Unstructured Finite Volume RANS Flow Solvers. In *17th AIAA Computational Fluid Dynamics Conference*, Toronto, Ontario, Canada. American Institute of Aeronautics and Astronautics.

Butler, C., Araya, D., McKiernan, G., and Wheaton, B. M. (2022). Supersonic transition measurements during the bolt flight experiment descent phase. AIAA Paper 2022-4099.

Candler, G., Barnhardt, M., Drayna, T., Nompelis, I., Peterson, D., and Subbareddy, P. (2007). Unstructured Grid Approaches for Accurate Aeroheating Simulations. In *18th AIAA Computational Fluid Dynamics Conference*, Miami, Florida. American Institute of Aeronautics and Astronautics.

Candler, G. V., Johnson, H. B., Nompelis, I., Gidzak, V. M., Subbareddy, P. K., and Barnhardt, M. (2015). Development of the us3d code for advanced compressible and reacting flow simulations. AIAA Paper 2015-1893.

Damm, K. A., Gibbons, N. N., Jacobs, P. A., and Gollan, R. (2023). Application of a jacobian-free newton-krylov method to the simulation of hypersonic flows. AIAA Paper 2023-2295.

Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Ekelschot, D. and Brock, J. (2022). Enabling metric-based mesh adaptation for advanced compressible flow simulations using us3d. AIAA Paper 2022-1866.

Fike, J. and Alonso, J. (2011). The development of hyper-dual numbers for exact second-derivative calculations. AIAA Paper 2011-886.

Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston.

Galbraith, M. C., Allmaras, S., and Darmofal, D. L. (2015). A verification driven process for rapid development of cfd software. AIAA Paper 2015-0818.

Gnoffo, P., Wood, W., Kleb, B., Alter, S., Padilla, J., and White, J. (2013). Functional equivalence acceptance testing of fun3d for entry, descent, and landing applications. AIAA Paper 2013–2558.

Gnoffo, P. A. (1990). An upwind-biased, point-implicit relaxation algorithm for viscous, compressible perfect-gas flows. NASA TP 2953.

Gropp, W., Keyes, D., McInnes, L. C., and Tidriri, M. D. (2000). Globalized Newton-Krylov-Schwarz Algorithms and Software for Parallel Implicit CFD. *The International Journal of High Performance Computing Applications*, 14(2):102–136.

Hall, L. H., Eppard, W., Applebaum, M., and Purinton, D. (2019). Modeling and simulation techniques for the nasa sls service module panel separation event; from loosely-coupled euler to full-coupled 6-dof, time-accurate, navier stokes methodologies. AIAA Paper 2019-1843.

Haselbacher, A. and Blazek, J. (2000). Accurate and Efficient Discretization of Navier-Stokes Equations on Mixed Grids. *AIAA Journal*, 38(11):2094–2102.

Haselbacher, A. C. (1999). *A Grid-Transparent Numerical Method for Compressible Viscous Flows on Mixed Unstructured Grids*. Doctoral, Loughborough University, Loughborough, UK.

Hinkle, A., Hosder, S., Johnston, C. O., and Thompson, K. B. (2023). Radiation interaction with particulates in planetary entry flows. AIAA Paper 2023-3732.

Katz, A. and Sankaran, V. (2011). Mesh quality effects on the accuracy of CFD solutions on unstructured meshes. *Journal of Computational Physics*, 230:7670–7686.

Kelley, C. T. and Keyes, D. E. (1998). Convergence Analysis of Pseudo-Transient Continuation. *SIAM Journal on Numerical Analysis*, 35(2):508–523.

Kleb, W., Park, M., Wood, W., Bibb, K., Thompson, K., and Gomez, R. (2019). Sketch-to-solution: An exploration of viscous cfd with automatic grids. AIAA Paper 2019–2948.

Knoll, D. and Keyes, D. (2004). Jacobian-free newton–krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357 – 397.

Kyle B. Thompson, Matthew D. O'Connell, A. D. H. and Pederson, C. C. (2024). Validation of the hypersolve cfd solver for entry descent and landing applications. AIAA Paper 2024-1973.

Lani, A., Villedie, N., Bensassi, K., Koloszar, L., Vymazal, M., Yalim, S. M., and Panesi, M. (2013). Coolfluid: an open computational platform for multi-physics simulation and research. AIAA Paper 2013-2589.

Liu, Y. and Nishikawa, H. (2016). Third-Order Inviscid and Second-Order Hyperbolic Navier-Stokes Schemes for Three-Dimensional Inviscid and Viscous Flows. In *46th AIAA Fluid Dynamics Conference*, Washington, D.C. American Institute of Aeronautics and Astronautics.

Loseille, A. and Alauzet, F. (2011a). Continuous Mesh Framework Part I: Well-Posed Continuous Interpolation Error. *SIAM Journal on Numerical Analysis*, 49(1):38–60.

Loseille, A. and Alauzet, F. (2011b). Continuous Mesh Framework Part II: Validations and Applications. *SIAM Journal on Numerical Analysis*, 49(1):61–86.

Loseille, A., Dervieux, A., Frey, P., and Alauzet, F. (2007). Achievement of Global Second Order Mesh Convergence for Discontinuous Flows with Adapted Unstructured Meshes. In *18th AIAA Computational Fluid Dynamics Conference*, Miami, Florida. American Institute of Aeronautics and Astronautics.

Luke, E. A. and Reese, D. S. (1999). *A rule-based specification system for computational fluid dynamics*. PhD thesis.

Luke, E. A., Tong, X.-L., Wu, J., Tang, L., and Cinnella, P. (2003). A Chemically Reacting Flow Solver for Generalized Grids.

Martin, R. C. and Coplien, J. O. (2009). *Clean code: a handbook of agile software craftsmanship*. Prentice Hall.

Mavriplis, D. J. (2021). A residual smoothing strategy for accelerating Newton method continuation. *Computers & Fluids*, 220:104859.

McCloud, P. L. (2017). Best Practices for Unstructured Grid Shock Fitting. In *55th AIAA Aerospace Sciences Meeting*, pages 1–14, Grapevine, Texas. American Institute of Aeronautics and Astronautics.

Mulder, W. A. and Van Leer, B. (1985). Experiments with implicit upwind methods for the Euler equations. *Journal of Computational Physics*, 59(2):232–246.

Nakashima, Y., Watanabe, N., and Nishikawa, H. (2016). Hyperbolic navier-stokes solver for three-dimensional flows. AIAA Paper 2016–1101.

NASA Office of Inspector General (2024). Nasa's readiness for the artemis ii crewed mission to lunar orbit. Technical Report IG-24-011, National Aeronautics and Space Administration, Washington, D.C.

Nastac, G., Miller, F., Schneider, D., Jacobson, K., Jones, W. T., and Opgenorth, M. (2025). Toward Adaptive Mixed-Element Unstructured Grids for Simulations of Viscous Flows. In *AIAA SCITECH 2025 Forum*, Orlando, FL. American Institute of Aeronautics and Astronautics.

Nastac, G., Tramel, R. W., and Nielsen, E. J. (2022). Improved Heat Transfer Prediction for High-Speed Flows over Blunt Bodies using Adaptive Mixed-Element Unstructured Grids. In *AIAA SCITECH 2022 Forum*, pages 1–26, San Diego, CA & Virtual. American Institute of Aeronautics and Astronautics.

Nishikawa, H. (2010). Beyond Interface Gradient: A General Principle for Constructing Diffusion Schemes. In *40th Fluid Dynamics Conference and Exhibit*, Chicago, Illinois. American Institute of Aeronautics and Astronautics.

Nishikawa, H. (2011a). New-generation hyperbolic navier-stokes schemes: O(1/h) speed-up and accurate viscous/heat fluxes. AIAA Paper 2011–3043.

Nishikawa, H. (2011b). Robust and accurate viscous discretization via upwind scheme – I: Basic principle. *Computers & Fluids*, 49(1):62–86.

Nishikawa, H. (2020). Robust numerical fluxes for unrealizable states. *Journal of Computational Physics*, 408:109244.

Nishikawa, H. (2024). Updates to Implicit Edge-Based Gradient Methods. In *AIAA SCITECH 2024 Forum*, Orlando, FL. American Institute of Aeronautics and Astronautics.

Nishikawa, H. and Liu, Y. (2017). Hyperbolic navier-stokes method for high-reynolds-number boundary layer flows. AIAA Paper 2017–0081.

Nishikawa, H., Nakashima, Y., and Watanabe, N. (2017). Effects of high-frequency damping on iterative convergence of implicit viscous solver. *Journal of Computational Physics*, 348.

O'Connell, M., White, J., and Thompson, K. (2021). Towards an automated unstructured grid adaptation workflow with vulcan. AIAA Paper 2021-2701.

O'Connell, M. D., Druyor, C. T., Thompson, K. B., Jacobson, K. E., Anderson, W. K., Nielsen, E. J., Carlson, J.-R., Park, M. A., Jones, W. T., Biedron, R. T., Kleb, B., and Zhang, X. (2018). T-infinity: The dependency inversion principle for rapid and sustainable multidisciplinary software development. AIAA Paper 2018–3856.

Pederson, C. and Schoenenberger, M. (2023). Exploring the Accuracy of RANS Simulations for Mars Entry Vehicles. In *AIAA Aviation 2023 Forum*, San Diego, CA. American Institute of Aeronautics and Astronautics.

Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D. (2014). Cfd vision 2030 study: A path to revolutionary computational aerosciences. NASA CR-2014-218178, Langley Research Center.

Thompson, K. B., Nishikawa, H., and Padway, E. (2023a). Economical third-order methods for accurate surface heating predictions on simplex element meshes. AIAA Paper 2023-2629.

Thompson, K. B. and O'Connell, M. D. (2019). Streamlined convergence acceleration for cfd codes. AIAA Paper 2019-3709.

Thompson, K. B., Wise, A. J., and Hollis, B. R. (2023b). Improvements to the laura mesh adaptation algorithm. AIAA Paper 2023-4332.

Tramel, R., Nichols, R., and Buning, P. (2009). Addition of Improved Shock-Capturing Schemes to OVERFLOW 2.1. In *19th AIAA Computational Fluid Dynamics*, San Antonio, Texas. American Institute of Aeronautics and Astronautics.

Weiss, J., Maruszewski, J., and Smith, W. (1997). Implicit solution of the Navier-Stokes equations on unstructured meshes. In *13th Computational Fluid Dynamics Conference*, Snowmass Village,CO,U.S.A. American Institute of Aeronautics and Astronautics.

Wright, M. J., Candler, G. V., and Bose, D. (1998). Data-parallel line relaxation method for the navier-stokes equation. *AIAA Journal*, 9(36):1603–1609.