Sparse Linear Algebra Toolkit for Computational Aerodynamics

Stephen L. Wood,[*], Kevin E. Jacobson[†], William T. Jones[‡], and W. Kyle Anderson[§]
NASA Langley Research Center, Hampton, VA 23681, USA

Finding solutions to sparse linear systems of equations is an essential step in Computational Engineering applications of interest to NASA. Linear systems of equations are composed and solved in almost every computational engineering application. The characteristics of linear systems vary greatly from one application to another. Accordingly, there are a wide variety of methods for the solution of linear systems of equations. The operations and methods prepared by the authors are focused on linear systems of interest to NASA, primarily those associated with Computational Fluid Dynamics (CFD), Aeroelasticity, and Aeroacoustics. The Sparse Linear Algebra Toolkit (SLAT) is a coordinated collection of software featuring operations, methods, and data structures that are useful when solving sparse linear systems of equations on modern computer architectures. The implemented operations and methods are designed and tuned for parallelism in shared memory, in distributed memory, and across the hybrid combination of distributed-shared memory. The toolkit includes novel methods and implementations for modern architectures and facilitates development of new approaches for meeting NASA's evolving computational engineering challenges using evolving computer architectures that are not available in vendor libraries. In this paper, significant features and interfaces within SLAT are presented and verified for simulations performed with NASA's CFD solver, FUN3D. The runtime and scaling performance of the Generalized Minimum Residual (GMRES) method implemented in SLAT is analyzed for the linear subproblems within the solution of turbulent Navier-Stokes equations employed in the simulation of high-lift configurations. Prior to this work, the SPARSKIT GMRES implementation was the only Krylov subspace method available within FUN3D. A strong scaling study shows the SLAT GMRES implementation facilitates accurate Reynolds-averaged Navier-Stokes CFD solutions between 15% and 56% faster than the SPARSKIT GMRES implementation.

I. Introduction

The scalability of a Computational Fluid Dynamics (CFD) method on modern computer systems directly impacts its efficacy as a tool for engineering design and optimization. A CFD method's ability to scale is most often viewed in light of two distinct goals, 1) "Strong scaling": produce a converged solution in the least wall clock time possible 2) "Weak scaling": produce higher fidelity results in the same amount of wall clock while maintaining a constant ratio of simulation work required to computational units utilized [1, 2]. Quick turnaround times are achieved by strong scaling when a fixed workload is divided into numerous small tasks executed in parallel. Strong scaling usually applies only over some finite range of utilized computational units (CPUs or GPUs) and breaks down when this number of units becomes excessively large because of Amdahl's Law and/or the overhead of parallel communication. Large capability runs are typically achieved via weak scaling, by completing more tasks of fixed size in the same length of wall time, rather than a fixed workload in less time.

The parallelization approach applied to an existing CFD method has a significant influence on its runtime and scalability. Generally, a parallelization approach consists of a subdivision of work among a pool of computational resources and a communication pattern that respects the constraints of the algorithms utilized. Partitioning, the subdivision of a CFD problem's spatial domain into parts for parallel processing, has significant influence on the performance and scalability of a CFD method [3]. The spatial domains considered in this work are described by unstructured meshes of elements where conservation laws are applied. An effective partitioning of a CFD problem

[*]Research Scientist, Computational AeroSciences Branch, MS 128, AIAA Member, stephen.l.wood@nasa.gov.
[†]Research Aerospace Engineer, Aeroelasticity Branch, MS 340, AIAA Member.
[‡]Computer Engineer, Computational AeroSciences Branch, MS 128, AIAA Associate Fellow.
[§]Senior Research Scientist, Computational AeroSciences Branch, MS 128, AIAA Associate Fellow.

divides the domain so that the workload is balanced and efficient communication patterns occur among computational units. On recent and current computational hardware, the movement of data in memory and over interconnections between computational units requires far more time than a floating point calculation [4]. Tuning a CFD method for efficient memory use and communication traffic is critical to achieving high performance [5]. Tuning a CFD method to effectively utilize heterogeneous hardware encompasses the considerations of partitioning and memory management with the additional concern of unequal computational units [6].

The algorithms utilized within a CFD method can constrain scalability with inherently sequential steps and high computational costs [7]. Frequently, and in this work, the solution of a nonlinear CFD problem also requires the solution of linearized systems of equations. The formation of the linearized system of equations is a nontrivial task that is linked to the chosen discretization [8]. Commonly, the linearized system is represented by a matrix equation, $Ax = b$, where a sparse coefficient matrix, A, and a right-hand side nonlinear residual vector, b, constitute the interactions between the solution update, x, the problem definition, and the current nonlinear solution. The solution of these linear subsystems during each nonlinear iteration of a CFD solver can account for more than 50% of the wall-time to solution (see details in Sec. V). Effective methods for the solution of a CFD solver's linear subsystems are vital to the solution of forward problems for analysis and to adjoint problems for mesh adaptation and design [9]. Krylov subspace methods are potent tools for asymmetric linear systems that can arise in CFD problems [10, 11]. These methods involve iteratively orthonormalizing trial solution vectors, commonly called search vectors, to minimize the linear residual. Preconditioning the matrix equation with an approximation of the inverse coefficient matrix, A^{-1}, can reduce the number of linear search directions required [12, 13]. Incomplete Lower Upper (ILU) factorizations developed by discarding some or all of the fill-in entries developed during the Gaussian Elimination of the sparse matrix, A, are prevalently used as preconditioners to Krylov subspace methods [14].

A set of tools are required to develop efficient linear algebra operations that meet the evolving needs of NASA projects. The challenges to software development posed by emerging computer architectures must be addressed in the design of the tool set. Existing linear algebra packages and vendor libraries strive to be general purpose and are not focused on the antisymmetric sparse linear systems that arise in CFD, Aeroelasticity, or Aeroacoustics. SPARSKIT [15], MAGMA [16, 17], PETSc [18], MKL 2019 [19], and CUDA10.2.89 [20], are well regarded numeric packages and their sparse linear algebra capabilities are widely used in scientific applications. However, these packages each target one or two computer architectures and do not have compatible software interfaces. In addition, there is limited support for multiple data types beyond real- and complex-value types. This serves as an impediment to the development and validation of the computational components that rely on gradient computations based on additional data types. As a result, none of these packages currently have all of the capabilities necessary to address the known priorities for NASA projects. Developing a toolkit in-house offers the opportunity to create coordinated components that deliver value for applications of interest. NASA has clear distribution rights to software toolkits developed in-house. Also, distributing a sparse linear algebra toolkit with NASA tool suites avoids a third-party dependency that customers would have to satisfy before obtaining engineering insights. SLAT is being introduced to address this evolving landscape of simulation fidelity demands and computational architectures. In Section II, the context and motivation for this work is described in an overview of a CFD solver's methods. The implementations of key methods are discussed in Section III. General and extensible interfaces to these implementations are presented in Section IV. Results obtained using SLAT are discussed in Section V. A code sample depicting use of SLAT's interfaces is presented in Section VI. This paper concludes with a summarization of the enabling capabilities within SLAT in Section VII.

II. CFD Solver Overview

In this work FUN3D-SFE is utilized as a representative CFD solver to explore the demands placed upon a linear solver. FUN3D-SFE is a continuous finite-element discretization within FUN3D [8]. The discretization is based on a stabilized finite-element approach that includes the Streamlined Upwind Petrov-Galerkin (SUPG) scheme [21, 22], Galerkin least squares [23], and variational multiscale methods [24]. In the results shown here, only the SUPG scheme is considered.

For turbulent flows, the Negative Spalart-Allmaras One-Equation Model (SA-neg) turbulence model [25] is tightly coupled with the flow equations, yielding a nonlinear algebraic system of equations with six variables at each mesh point. A linear nodal basis is used in this study, which is designed to be second-order accurate in space. The current implementation includes capabilities for computing on tetrahedra, hexahedra, pyramids, and prisms, although all the results shown in the present paper are for pure tetrahedral mesh construction.

FUN3D-SFE is used to perform forward CFD analysis, sensitivity analysis, adjoint analysis [26], and linearized frequency-domain analysis [27]. Forward CFD analysis and adjoint problems use real-values, typically in double (64 bit) precision, to represent the domain, initial conditions, and solution [8]. Sensitivity analysis problems use double precision complex or surreal values to perform algorithmic differentiation [28]. Linearized frequency-domain analysis problems use double precision complex data to perform traditional complex arithmetic [27]. Accordingly, a linear solver for computational aerodynamics needs to support these data types and modes of operation.

A. Solving the nonlinear system

To advance the solution toward a steady state, the density, velocities, temperature, and the turbulence working variable are updated in a tightly-coupled Newton-type solver described in Ref. [8]. Here, an initial update to the flow variables is computed using a locally varying pseudotime-step parameter that is later multiplied by the current Courant–Friedrichs–Lewy (CFL) number. The CFL number is adjusted during the iterative process. The initial update, ΔQ, is the solution to a linearization of the nonlinear state that consists of the Jacobian matrix, $\partial R/\partial Q$ and residual, ∂R as shown in Eq. 1:

$$\frac{\partial R}{\partial Q}\Delta Q = -\partial R \; . \tag{1}$$

Using the full update of the variables, the L_2 norm of the nonlinear residual is compared to its value at the beginning of the iteration. If the L_2 norm after the update is less than one half of the original value, the CFL number is increased and the iterative process continues to the next iterative cycle. If the L_2 reduction target for the residual is not met, a line search is conducted to determine an appropriate relaxation factor. Here, the L_2 norm of the residual is determined at four locations along the search direction and the optimal relaxation factor is determined by locating the minimum of a cubic polynomial curve fit through the samples. If the optimally relaxed update satisfies the residual reduction target then the solution is updated using the relaxation factor and the CFL number is maintained for the next iteration. If the optimal relaxation factor is small or if the relaxed update does not satisfy the minimum residual reduction target, then the solution is not updated and the CFL number is reduced by a factor of 10.

B. Solving linear subsystems

During each iteration of the nonlinear solver, the linear system is solved using a preconditioned Krylov method. Typically, the Generalized Minimal Residual (GMRES) [10] method is used with a preconditioner based on an Incomplete Lower Upper (ILU) decomposition [11]. A Krylov subspace of dimension 300 with one possible restart is usually employed.

Within GMRES, an Arnoldi process is utilized to orthonormalize search vectors with the goal of efficiently traversing the solution space (a Krylov subspace) toward a vector that satisfies the residual tolerance. Two orthonormalization methods, Modified Gram-Schmidt [11] and Householder Transformation Arnoldi [29–31] processes, are utilized and compared in this work.

Generally, preconditioning is a way of transforming a difficult problem into one that is easier to solve. More precisely, preconditioning attempts to improve the spectral properties of the coefficient matrix. SLAT enables left-, right- or both-side-preconditioning. Right-preconditioning was performed to produce the results presented in this work. Right-preconditioning refers to right-multiplying the system of equations by a preconditioning matrix, M^{-1}, that approximates the inverse of a matrix, denoted A for generality,

$$AM^{-1}y = b, \quad x = M^{-1}y \; . \tag{2}$$

In forward CFD analysis, A is the Jacobian matrix, $\partial R/\partial Q$, y is the update to the nonlinear system, ΔQ, b is the residual $-\partial R$ and Eq. 2 represents Eq. 1 with right-preconditioning.

For Symmetric Positive Definite (SPD) problems, the rate of convergence of the conjugate gradient method depends on the distribution of the eigenvalues of A. It is likely that the transformed (preconditioned) matrix, AM^{-1}, will have a smaller spectral condition number, and/or eigenvalues clustered around 1. For the asymmetric matrices considered in this work, the eigenvalues may not be predictive of how well GMRES converges [11, 32]. However, a clustered spectrum (away from 0) often results in rapid convergence, particularly when the preconditioned matrix is close to normal [12, 14]. In this paper, two techniques for generating and applying preconditioners are discussed.

III. Implementations

The implementations introduced in this work highlight significant features and toolkit design considerations required to support CFD, Aeroelastic, and Aeroacoustic applications of interest to NASA. Capabilities not available in existing linear algebra packages and vendor libraries are noted.

A. Data types

1. Real

Standard real-valued operations on scalar numbers, $x \in \mathbb{R}$, are appropriately implemented within the standard libraries of compilers prevalently utilized in HPC [20, 33, 34]. No alteration of scalar operations is required to manipulate real-valued vectors of the vector space $\mathbb{X} = \mathbb{R}^n$. Forward analysis and adjoint problems pose no additional requirements on the data types or operations of a linear solver.

2. Complex-step

The complex-step approach [28] uses complex variables to aid in the determination of discretely consistent derivatives. This approach is similar to that obtained using either finite differences or automatic differentiation. However, unlike real-valued finite differences, the present approach is not subject to subtractive cancellation errors.

The central idea is to represent a real-valued variable and its real-valued derivative throughout operations as the real and imaginary components of a complex number, respectively, $z \in \mathbb{R}^2$. For example, to calculate the sensitivity, df/dx_1, of a multiplication operation, $f = x_1 x_2$, using the complex-step method a small imaginary perturbation, h_1 is applied as shown in Eq. 3. The magnitude of the imaginary perturbation should be chosen small enough so that the accumulation of products of imaginary values will not influence the real-valued solution.

$$\begin{aligned}
h_1 &= 1.0e - 30 \\
h_2 &= 0 \\
z_1 &= x_1 + ih_1 \\
z_2 &= x_2 + ih_2 \\
f &= z_1 z_2 \\
f &= (x_1 + ih_1)(x_2 + ih_2) \\
f &= x_1 x_2 - h_1 h_2 + i(x_1 h_2 + x_2 h_1) \\
df/dx_1 &= \mathrm{Im}(f)/h_1 \\
df/x_1 &= (x_1 h_2 + x_2 h_1)/h_1 \ . \qquad (3)
\end{aligned}$$

It is noteworthy that the real portion of the complex-step multiplication operation, $x_1 x_2 - h_1 h_2$, differs from the real-valued operation, $x_1 x_2$, by the product of the perturbations, $h_1 h_2$, which in this example is conveniently 0. In practical use cases where perturbations are applied to more than one variable, the real portion of the result of a multiplication operation may be influenced by the product of accumulated perturbations. There are several operations where standard complex arithmetic will not produce the desired propagation of a value and derivative pair. One such operation that is frequently used within a linear solver, absolute value, *abs*, is detailed in Table 1.

Table 1 Possible implementations of the absolute value of a complex number.

$abs(z) = abs(x + ih)$	
Complex-valued	Complex-step
$abs(z) = \sqrt{x^2 + h^2}$	$abs(z) = \begin{cases} x + ih\,, & x \;>= 0 \\ -x - ih\,, & x \;< 0 \end{cases}$

For more details on the use of the complex-step method in CFD applications, interested readers are directed to the thorough discussions in Refs. [28, 35–37]. Complex-step operations are not natively supported in MKL 2019 [19], CUDA10.2.89 [20], SPARSKIT [15], MAGMA [16, 17], and PETSc [18].

3. Surreal

The surreal approach defines a data type of the same name to represent a value and derivative pair instead of repurposing a complex number for the task as the complex-step approach does [38]. By defining a separate data type with bespoke operators for algorithmic differentiation, errors and complications arising from inadvertent use of standard complex arithmetic operations can be avoided [35]. The sensitivity, df/dx_1, of a multiplication operation, $f = x_1x_2$, with respect to x_1 can be directly calculated using surreals without any unintended interaction between the derivative and the value as shown in Listing 1.

Listing 1 Surreal multiplication implementation snippet.

```
class surreal : public std::complex<double> {
public:
  inline void val(const double& a) {std::complex<double>::real(a);}
  inline void deriv(const double& b) {std::complex<double>::imag(b);}
  inline void val(const double& a) {std::complex<double>::real(a);}
  inline void deriv(const double& b) {std::complex<double>::imag(b);}
  inline surreal operator*(const surreal&) const;
};

inline surreal surreal::operator*(const surreal& z) const
{
  return surreal(val()*z.val(),val()*z.deriv()+z.val()*deriv());
}
```

The approach of defining a custom data type and operations for a value and derivative pair can be extended to store a value and multiple partial derivatives [39]. This approach is potent for partial derivatives because it enables rapid implementation and exploration of models and algorithms for CFD solvers [8]. Surreal datatypes are not supported in MKL 2019 [19], CUDA10.2.89 [20], SPARSKIT [15], MAGMA [16, 17], and PETSc [18].

4. Complex-valued

The core algorithms of a linear solver are built upon mathematical vector operations such as dot (inner) products, vector norms, and Givens rotations. While standard complex-valued operations on scalar complex numbers, $z \in \mathbb{C}$, are appropriately implemented within the standard libraries of compilers prevalently utilized in HPC [20, 33, 34]. Operations on mathematical vectors of complex numbers are not appropriately implemented in standard compiler libraries because these libraries do not contain data structures or operations that represent mathematical vectors.

In the complex-valued vector space $\mathbb{X} \in \mathbb{C}^n$, a "canonical" inner product is the Euclidean inner product. The Euclidian inner product of two vectors, $x = (x_j)_{j=1,\dots,n}$ and $(y_j)_{j=1,\dots,n}$ of $\mathbb{C}^n$ is defined by

$$(x, y) = \sum_{j=1}^{n} x_j \bar{y}_j \,, \tag{4}$$

in which the over-bar denotes elementwise complex conjugation, $y_j = (a + ib)_j$; $\bar{y}_j = (a - ib)_j$ [11].

The Euclidean norm of a complex vector, x is defined by

$$||x||_2 = (x, x)^{1/2} = \sqrt{\sum_{j=1}^{n} x_j \bar{x}_j} = \sqrt{\sum_{j=1}^{n} a_j^2 + b_j^2} \,. \tag{5}$$

Givens rotations are a practical means of solving least-squares problems that require a slight but significant alteration for complex-valued problems [11]. The sine and cosine for the Complex Givens rotation matrix for step j are given by:

$$s_j = \frac{h_{j+1,j}}{\sqrt{|h_{jj}^{(j-1)}|^2 + h_{j+1,j}^2}} \,, \; c_j = \frac{h_{j,j}^{(j-1)}}{\sqrt{|h_{jj}^{(j-1)}|^2 + h_{j+1,j}^2}} \,, \tag{6}$$

with, $|c_j|^2 + |s_j|^2 = 1$. The Complex Givens rotations are defined as

$$\Omega_j = \begin{array}{cl} \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & \bar{c}_j & \bar{s}_j & & & \\ & & & -s_j & c_j & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix} & \begin{array}{l} \\ \\ \\ \leftarrow \text{row } j \\ \leftarrow \text{row } j+1 \\ \\ \\ \\ \end{array} \end{array} . \tag{7}$$

The sine and cosine values defined by Eq. 6 can be stored in arrays because the conjugated terms in Eq. 7 are only needed during the application of Givens rotations for each step.

B. Reordering

Reordering the rows and columns of a matrix can often make its LU or QR factors sparser, thus saving storage space and memory access costs, and influencing the stability of the factorizations. Reordering a matrix can also reduce its bandwidth, thus coalescing the indirect accesses to a vector during matrix-vector product operation. The Cuthill-McKee (CMK) [40] reordering algorithm, which seeks to minimize the band-width of a matrix, is implemented within SLAT. A Min-Max Independent Set Coloring (MMISC) algorithm developed for this work and ongoing research efforts [41] are implemented within SLAT. The MMISC algorithm is developed from Luby's Algorithm [42], which forms maximal independent sets of nodes in a graph. Maximal independent sets are groups of nodes within a graph that are not directly connected to nodes within the group. Restated, a node in a maximal independent set do not share edges with any other nodes in the set. In Luby's algorithm, a random number, r, is assigned to each node prior to parallel graph traversal. Then during parallel graph traversal, an independent set is formed by selecting nodes that have maximum value of r among neighbors. The set can then be made maximal by iterating on remaining nodes until no more can be added to a set.

The Min-Max variant implemented in SLAT, forms two independent sets during each graph traversal by adding nodes with maximum values of r among neighbors to one set and nodes with minimal r values to another. Forming two sets during each traversal improves the runtime by more than 40% because the indirect memory access of the sparse matrix structure dominates the computational cost. For the unstructured 3D CFD meshes examined in this work, the Min-Max variant produces larger and more balanced sets, using fewer total colors to cover a graph, than greedy coloring algorithms and Luby's algorithm. The colors produced by MMISC can be used to reorder a matrix to facilitate parallel operations on each independent set. Reordering operations are not supported in MAGMA [16, 17]. A limited selection of "black-box" reordering operations are available in MKL 2019 [19], CUDA 10.2.89 [20]. The source code of a limited selection of reordering operations is accessible in SPARSKIT [15] and PETSc [18].

C. Preconditioners

1. Incomplete Fill-in

The sparse $n \times n$ matrices of unstructured 3D CFD problems typically contain significantly less than 5% of the total possible entries. A full LU factorization of a large sparse matrix with 5% occupancy results in a dense matrix that requires 20× more storage and memory access than the original matrix. It is often impractical to compute and store a full LU factorization. Incomplete LU can be performed with no fill-in, that is with only entries, (i, j), that are a part of the sparsity pattern, S, of the original matrix A. This is termed a fill-level of 0 and denoted as ILU(0). One potent strategy for controlling fill-in is to use a factor, k, to limit the level of fill-in entries added to the sparsity pattern, $\hat{S}$, of the LU factorization. The level of a candidate fill-in entry is determined by the access pattern that would occur during the numeric phase of the ILU factorization to operate on the entry,

$$level(i, j) = \begin{cases} 0\,, & if\,(i, j) \in S \\ min_{1 \le h < min(i,j)} level(i, h) + level(h, j) + 1\,, & otherwise \end{cases} . \tag{8}$$

The incomplete factorizations available in MKL 2019 [19] and CUDA 10.2.89 [20] do not include capabilities to introduce fill-ins.

2. ILU(k)

A classic preconditioner to GMRES is the Incomplete Lower Upper (ILU) factorization generated from the sequential Gaussian Elimination (GE) algorithm (see Algorithm 1) [11, 43, 44]. In this algorithm, a single thread of execution updates each $nq \times nq$ block of A in series. The procedure begins with $\boldsymbol{A_{11}}$ then sequentially updates all blocks in the second row of the sparsity pattern S. This algorithm then updates the third row from $\boldsymbol{A_{2,min(j)}} \in S$ to $\boldsymbol{A_{2,max(j)}} \in S$. The sequential access pattern of the GE algorithm results in the update of an arbitrary block A_{kl} reading the block elements of the k^{th} row, $\boldsymbol{A_{kj}}$, $(j < min(k,l)) \in S$, and the l^{th} column, $\boldsymbol{A_{il}}$, $(i < min(k,l)) \in S$, that have already been updated. In Algorithm 1, the factorization is formed in place (the elements of A are replaced with the elements of the factors) with L being a strictly lower triangular block matrix; the identity block matrices on the main diagonal are omitted in favor of the factored values of U, which is an upper triangular matrix. Note, zero-based indices are employed throughout the Algorithm pseudocodes.

Algorithm 1 Block ILU Factorization [43, 45].

```
1 for $i = 1$ to $n$ do
2   for $k = 0$ to $i - 1$ and $(i,k) \in S$ do
3     $\boldsymbol{A_{ik}} = \boldsymbol{A_{ik}} / \boldsymbol{A_{kk}}$
4     for $j = k + 1$ to $n$ and $(i,j) \in S$ do
5       $\boldsymbol{A_{ij}} = \boldsymbol{A_{ij}} - \boldsymbol{A_{ik}}\boldsymbol{A_{kj}}$
6     end
7   end
8 end
```

Gaussian Elimination is a sequential algorithm. When applied to block sparse matrices, it is commonly referred to as Block Incomplete Lower Upper (BILU) factorization. There is a loop-carry dependency between each of the loops over rows, i, and the columns (indices j and k). This efficient access pattern results in an effective preconditioner. However, parallelizing Gaussian Elimination relies solely on domain decomposition via partitioning. Because only one thread of execution is possible within the Gaussian Elimination algorithm, each partition is assigned one computational unit. Communication between partitions is conducted with MPI [46]. Only scalar implementations of ILU factorization and application are available in MKL 2019 [19], CUDA 10.2.89 [20], and MAGMA [16, 17]. Scalar implementations of ILU factorization and application routines require 36× more indirect memory accesses for 3D turbulent simulations.

3. Level Scheduled ILU(k)

An alternative approach is to expose parallelism in the Gaussian Elimination method by detecting the specific dependencies between rows of the A matrix. An A matrix represents a mesh partition where the i^{th} row in the matrix represents the i^{th} mesh point in the partition, and an entry in that row, $\boldsymbol{A_{ij}}$, represents a connection between mesh points i and j. This is how the adjacency graph of a mesh partition is encoded into its A matrix. Determining dependencies within a directed graph is a fundamental problem in the field of graph theory known as topological sorting. The general topological sorting problem has multiple variants and algorithmic solutions [47]. These variants all produce groups of mesh points, called level sets, which are ordered from a starting mesh point, called the root mesh point. Mesh points within a level set are not dependent on each other and only depend on mesh points in preceding level sets. The process to construct level sets of mesh points that may be factored in parallel is shown in pseudocode in Algorithm 2. The implementation is developed from fundamental discussions of graph sorting in [47, 48]. Algorithm 2 differs from general topological sorting routines in the following ways: 1) the graph is symmetric (no constraints are placed on symmetry of values in the matrix), 2) the graph's adjacency is represented in the three array variant of Compressed Sparse Row storage (CSR) [19], 3) the root mesh point is mesh point 0, row 0 in Compressed Sparse Row storage, and 4) all mesh points will be sorted.

The level sets formed by Algorithm 2 are stored, used for the generation of a Level Scheduled Block Incomplete LU (LSBILU) Factorization as shown in Algorithm 3, and reused for the application of the factorization. In practice, it is efficient to store the level sets in a static structure like CSR rather than to traverse a dynamic structure during each application of the LSBILU preconditioner.

Algorithm 2 Topological Sort of Compressed Sparse Row Adjacency.

```
// Initialize work arrays
diagidx[n] = extract_major_diagonal( row[n+1], col[nnz] );
depth[n] = {0};
row_max_depth = 0;
std::vector<int> emptyLevel;
std::vector< std::vector<int> > level;

// Form level sets
for (i=0; i<n; ++i) {
  row_max_depth = 0;
  for (j=row[i]; j<=diagidx[i]; ++j) {
    row_max_depth = MAX(row_max_depth, depth[ col[j] ]);
  }
  depth[i] = 1 + row_max_depth;
  if (level.size() < depth[i]) {
    level.push_back(emptyLevel);
  }
  level[ depth[i]-1 ].push_back(i);
}
```

Algorithm 3 Level Scheduled Block Incomplete LU Factorization.

```
// Initialize unknowns of the fused $LU_{ij}$ matrix to values of $A_{ij}$
for (int $l=0$; $l<$level.size(); $++l$) {
 parallel for ($ln=0$; $ln<$ level[$l$].size(); $++ln$) {
  $i$ = level[$l$][$ln$];
  for $k=1$ to $i-1$ and $(i,k) \in S$ do
   $LU_{ik} = LU_{ik}/LU_{kk}$
   for $j=k+1$ to $n$ and $(i,j) \in S$ do
    $LU_{ij} = LU_{ij} - LU_{ik}LU_{kj}$
   end
  end
 }
}
```

LSBILU produces a preconditioner that is equivalent to the BILU preconditioner if operations are performed with infinite precision. OpenMP is utilized to orchestrate CPU cores in shared memory [33]. This topological parallelization approach enables an amount of concurrency that is specific to a mesh, the partition sizes created, and the reordering applied. For the sparse matrices considered in this work, the average number of independent mesh points in a level set is less than 15 when reordered with CMK. When reordered with MMIS, the average number of independent mesh points in a level set is less than 25. This suggests an upper limit to the speed up derived from this topological parallelization. Parallel implementations of ILU are not available in MKL 2019 [19], CUDA10.2.89 [20], SPARSKIT [15], or PETSc [18].

D. Linear solvers

1. Modified Gram-Schmidt

The Gram-Schmidt (GS) Arnoldi process is a method for orthonormalising a set of vectors in an inner product space, most commonly the Euclidean space, R^n, equipped with the standard inner product [11]. In this notation, n is the number of linearly independent equations in the linear system; the number of rows and the number of columns of the A matrix. The GS process takes a finite, linearly independent set of vectors, $V = v_1, ..., v_k$ for $k \leq n$, and generates an orthogonal set $V' = u_1, ..., u_k$ that spans the same k-dimensional subspace of R^n as V [43]. The application of the GS process to the column vectors of a full column rank matrix yields the QR decomposition (it is decomposed into an orthogonal matrix and a triangular matrix). When the GS process, is implemented on a computer, the vectors u_k are often not quite orthogonal, due to rounding errors. For the GS process, this loss of orthogonality is particularly bad; therefore, it is said that the (classical) Gram–Schmidt process is numerically unstable [11]. The Gram–Schmidt process can be stabilized by a small modification; this version is sometimes referred to as modified Gram-Schmidt (MGS). The modification is to orthonormalize against any errors introduced in computation of a previous vector, $u_k^{(i-1)}$ rather than only orthonormalising against the previous vectors. This approach gives the same result as the original formula in exact arithmetic and introduces smaller errors in finite-precision arithmetic [43]. The projection operator defined by MGS is

$$\mathrm{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle}\mathbf{u}\,, \tag{9}$$

which is visualized in Figure 1.

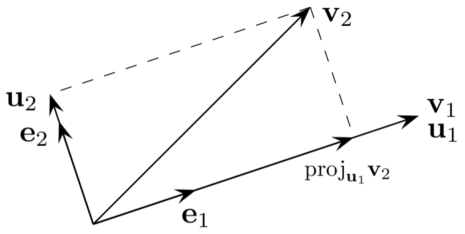


Fig. 1 The first two steps of the Modified Gram–Schmidt process [49].

Even the MGS Arnoldi process can introduce significant error if the vectors on which it operates are not sufficiently independent. The performance of an Arnoldi process can be measured by monitoring the errors in orthogonality between the vectors spanning the subspace. The orthogonality errors between a set of vectors, $(v_1, ..., v_m)$, can be calculated as follows: Let $V = (v_1, ..., v_m)$ be an $n \times m$ matrix of column vectors which are to be orthonormalized. Let $Q = (q_1, ..., q_m)$ be the computed result of applying an Arnoldi process to the columns of V. Then the orthogonality error matrix, E, is defined as

$$E = Q^T Q - I\,. \tag{10}$$

In exact arithmatic, E, would be the zero matrix. When using floating point arithmetic with unit rounding error u, Bjorck [50] shows that the orthogonality error, $||E||_{Frobenius}$, scales as

$$||E_{MGS}||_{Frobenius} \approx uk_2(V), \tag{11}$$

where $k_2(V)$ is the condition number of V.

Algorithm 4 Modified Gram-Schmidt Arnoldi [11, 51].

```
// Choose a vector $v_1$ with $||v_1||_2 = 1$
for $j = 1, 2, ..., m$ do
  // Compute an orthonormal vector $w_j$
  $w_j := Av_j$
  $n1 = ||w_j||_2$
  for $i = 1, 2, ..., j$ do
    $h_{ij} = w_j^T v_j$
    $w_j := w_j - h_{ij} v_i$
  end
  $h_{j+1} = ||w_j||_2$
  // Check for loss of orthogonality
  $n2 = h_{j+1}$
  $n3 = (n1 + 10^{-6} * n2) - n1$
  if $n3 < \epsilon$ then
    for $i = 1, 2, ..., j$ do
      $n1 = w_j^T v_j$
      $h_{ij} = h_{ij} + n1$
      $w_j := w_j - n1 v_i$
    end
    $h_{j+1} = ||w_j||_2$
  end
  // Compute a new Arnoldi vector $v_{j+1}$
  $v_{j+1} = w_j / h_{j+1,j}$
end
```

2. Householder Transformation

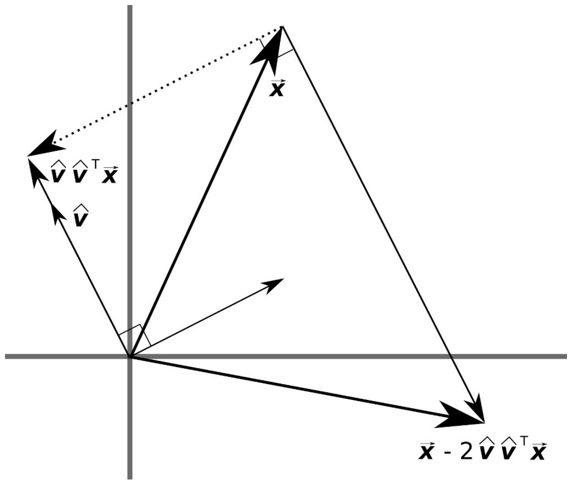
An alternative orthonormalization procedure based on the use of Householder transformations is shown by Walker [29] to be reliable even if the vectors to be orthonormalized are not very independent. A Householder transformation is a linear transformation given by the matrix, P,

$$P = I - 2\hat{v}\hat{v}^T$$

where I is the identity matrix and $\hat{v}$ is a unit vector. Px is the reflection of x about the hyperplane passing through the origin with normal vector, $\hat{v}$.

$$\begin{aligned} Px &= (I - 2\hat{v}\hat{v}^T)x \\ &= x - 2\hat{v}\hat{v}^T x \end{aligned}$$

Since $\hat{v}\hat{v}^T x$ is the projection of x onto $\hat{v}$, then Px reflects x about the hyperplane with normal, $\hat{v}$ as shown in Figure 2 [52]. To orthonormalize the columns of $V = (v_1, ..., v_m)$, one determines Householder transformations $P_1, ..., P_m$ such



$\vec{x}$
$\hat{v}\hat{v}^T\vec{x}$
$\hat{v}$
$\vec{x} - 2\hat{v}\hat{v}^T\vec{x}$

Fig. 2 Diagram representing the Householder transformation.

that $P_m...P_1V = R$, is an upper-triangular matrix. It is natural to determine $P_1, ..., P_m$ inductively by the requirement that $P_k...P(v, ..., v_k)$ be upper-triangular for $k = 1, ..., m$ if $P_m...P_1V = R$. This requirement is met if and only if for $k = 2, ..., m$ the first $k - 1$ components of the Householder vector determining P_k are zero. This process is described in Algorithm 5. Since $V = P_1...P_mR$, the matrix Q consisting of the first m columns of $P_1...P_m$ gives the desired orthonormalization of the columns of V. If Q is computed in floating point arithmetic with unit rounding error u, then Bjorck [50] shows that the orthogonality error scales as

$$||E_{Householder}||_{Frobenius} \approx u \ . \tag{12}$$

Comparing Figures 1 and 2 provides a visual sense of the difference between the projections within MGS and the reflections within a Householder Transformation Arnoldi process and an indication of how orthogonality errors differ between the approaches.

Algorithm 5 Householder Transformation Arnoldi [11, 29–31].

```
// Choose a vector $v_1$ with $||v_1||_2 = 1$
for $j = 1, 2, ..., m+1$ do
  // Compute the Householder unit vector $w_j$ such that
   $(w_j)_i = 0, i = 1, ..., j-1,$ and
   $(P_j v_j)_i = 0, i = j+1, ..., n,$ where $P_j = I - 2w_j w_j^T$
  $h_{j-1} = P_j v_j$
  $v_j = P_1 P_2 ... P_j e_j$
  if $j \le m$
    // Compute a new Arnoldi vector $v_{j+1}$
    $v_{j+1} := P_j P_{j-1} ... P_1 A v_j$
  end
end
```

The linear systems considered in this work indicate that the Householder Transformation Arnoldi implementation developed by Walker [29] has better numerical properties than the modified Gram-Schmidt implementation, especially in the final GMRES iterations when the residual is reduced below single precision, 10^{-8}. Walker's Householder implementation uses slightly less storage than the Gram-Schmidt implementation; however, it requires additional arithmetic. The increase in arithmetic is always less than a factor of three per search direction [29]. There are linear systems where the reduced orthogonality error produced by the Householder Arnoldi process enables an approximate solution to be found in significantly fewer search directions than required when orthonormalization is performed using MGS [53]. Since the arithmetic cost of an approximate solution obtained by GMRES is proportional to the number of search directions required to meet the residual tolerance, the Householder Transformation Arnoldi process is more efficient than MGS for linear systems sensitive to orthogonality error. Orthonormalization by the Householder Transformation Arnoldi process is not available in MKL 2019 [19], CUDA10.2.89 [20], SPARSKIT [15], MAGMA [16, 17], or PETSc [18].

3. Flexible GMRES

The flexible GMRES (FGMRES) variant of the GMRES method with right preconditioning allows variable preconditioning of each search vector. In this work, a single preconditioner is applied throughout the search for an approximate solution to a linear system. The flexible implementation is utilized as a building block for ongoing explorations of guided preconditioning during the search for an approximate solution. The only difference between FGMRES and the standard GMRES using MGS is that the preconditioned vectors, z_j, are saved and used to update the approximate solution. It should be noted that when the preconditioner is constant throughout ($M_j = M$ for $j = 1, ..., m$) then the new method is mathematically equivalent to the standard GMRES using MGS with right preconditioning [54].

IV. Interfaces

The interfaces introduced in the work are part of a broader effort to define interfaces for computational engineering components [55, 56]. The "tinf_" prefix of function names provides disambiguation and signifies the goal of supporting interoperable components by providing generic interfaces that are extensible beyond the "time $t + 1$" problem solution to represent the "time $t + \infty$" solution. Inspired by Buzz Lightyear [57], the design is extensible "to infinity and beyond". The common thread of this effort is the management of an individual component state in an opaque *object* pointer, or *handle*. The pointer is created by the interface implementation and subsequently supplied to functions that perform a desired action. The pointer is opaque and cannot be interpreted outside the component implementation but allows multiple instantiations of a given component, each with independent state.

Again, the broader effort is detailed in Ref. [56], but of interest here is the parallel communications component. This component will be exchanged with SLAT to enable the interaction of partitioned data. The communications component is referred to as *Iris* in the broader effort (the messenger of the Olympian gods). With respect to SLAT, the Iris handle is provided so that the SLAT implementation can utilize it in internal invocation of the Iris interface. Similarly, Iris is also used to produce a synchronization pattern (i.e., send/receive pairs) for communication of partition boundary data. Again, an opaque pointer is used to represent this pattern and is meaningful only to the Iris implementation. The synchronization pattern is provided to SLAT, where needed, for its internal use of Iris to perform these operations.

A. Hybrid BLAS

The increasing heterogeneity of computer architectures is motivating development of linear algebra algorithms that exploit multilevel parallelism and mixed-precision [4–6, 58, 59]. While the form and action of these operations is an open and evolving research area, the fundamental operations of linear algebra remain pertinent tools for exploratory and production use cases. To support current concurrency options, and the possibility of new modes of concurrency, an extensible type enumeration (enum), TINF_CONCURRENCY_TYPE, has been defined as shown in Table 2.

To support the data types used in operations of forward analysis, sensitivity analysis, design, adaptation, and linearized frequency-domain in an extensible way an enum, TINF_DATA_TYPE, has been defined as shown in Table 3.

To support the variety of matrix storage formats utilized within CFD solvers and their coupled components in an extensible way, an enum, TINF_MATRIX_TYPE, has been defined as shown in Table 4. The names of the matrix storage formats follow the prevalent naming conventions [19, 20, 60, 61]. The DENSE format is logically defined as an $m \times n$ matrix with 100% element occupancy. Three storage arrays with generalized names, value, index1, and index2, are used to define each sparse matrix format. The Column Oriented format, COO, represents a sparse matrix as a list of

Table 2 enum TINF_CONCURRENCY_TYPE.

TINF_SEQUENTIAL	1
TINF_THREAD	2
TINF_MPI	3
TINF_MPI_THREAD	4

Table 3 enum TINF_DATA_TYPE.

TINF_INT32	1
TINF_INT64	2
TINF_FLOAT	3
TINF_DOUBLE	4
TINF_CHAR	5
TINF_BOOL	6
TINF_CMPLX_FLOAT	7
TINF_CMPLX_DOUBLE	8
TINF_CMPLX_STEP_FLOAT	9
TINF_CMPLX_STEP_DOUBLE	10
TINF_SURREAL_FLOAT	11
TINF_SURREAL_DOUBLE	12

(row, column, value) tuples in index1, index2, and value, respectively. The COO format can be convenient for matrix assembly operations. The Compressed Sparse Row format, CSR, represents a sparse matrix by storing nonzero values in the val array, the index1 array contains the index of the first value of each row followed by the total number of nonzero scalar values, and the column indices of each scalar nonzero value in the value array are stored in the index2 array. The Compressed Sparse Column format, CSC, is similar to CSR except that values are read first by column, the index1 array contains the index of the first value in each column followed by the total number of nonzero scalar values, and a row index is stored for each value in the index2 array. The CSR and CSC formats can be convenient for matrix-vector products. The Block Sparse Row format, BSR, is similar to the CSR format except that each column index corresponds to a group of consecutive scalar in the values array that form a square row-major dense block. The Block Sparse Column format, BSC, has a relationship to the BSR format that is similar to the relationship between CSC and CSR; BSC is similar to the CSR format except that each row index corresponds to a group of consecutive scalars in the values array that form a square column-major dense block. The Block Sparse Row Column format, BSRC, is a variant of the BSR format that stores values of square dense blocks in column-major order. The Block Sparse Column Row format, BSCR, is a variant of the BSC format that stores values of square dense blocks in row-major order. Storing a matrix in one of the Block Sparse storage format variants rather than CSR or CSC reduces the number of indirect memory accesses required during an arithmetic operation by the square of the block size and is recommended when possible. It should be noted that all the implementations within SLAT will operate with a block size of one when processing the CSR and CSC formats. A storage format conversion utility is implemented within SLAT.

The TINF_CONCURRENCY_TYPE, TINF_DATA_TYPE, and TINF_MATRIX_TYPE enums are extensible and sufficient to define a wealth of linear algebra operations. To introduce the interface design, three operations are presented in this work, the L2 norm of an array, the dot product (inner product) of two arrays, and a sparse matrix-vector product operation.

The tinf_norm2 function shown in Listing 2 calculates the L2 norm, $r = ||x||_2$, of a vector, x, and stores the value in a scalar, result. The output argument, result, is underlined in Listing 2 to distinguish it from the input arguments.

Table 4 enum TINF_MATRIX_TYPE.

TINF_DENSE	1
TINF_COO	2
TINF_CSR	3
TINF_CSC	4
TINF_BSR	5
TINF_BSC	6
TINF_BSRC	7
TINF_BSCR	8

The extensible TINF_DATA_TYPE and TINF_CONCURRENCY_TYPE enums along with the iris_context, and iris_sync_pattern arguments that are active for MPI parallelism enable the calculation of L2 norms of 12 different data types using four modes of concurrency. The interfaces presented in this work along with the iris_context and iris_sync_pattern arguments are part of a broader effort to define interfaces for computer aided engineering applications [56].

Listing 2 $r = ||x||_2$.

```
int32_t tinf_norm2 ( const int32_t select_tinf_data_type,
        const int32_t select_tinf_concurrency_type, const int64_t n, const void *const x,
        const int64_t incx, const void *iris_context,  const void *iris_sync_pattern,
        void *const result )
```

select_tinf_data_type	identify the data type of the x array and result
select_tinf_concurrency_type	identify the concurrency execution type
n	number of scalar values in x
x	dense input array of size n
incx	spacing between scalar values in x
result	L2 norm of x, output
iris_context	IRIS communications context
iris_sync_pattern	IRIS communications synchronization pattern
returns	error code, TINF_SUCCESS or TINF_FAILURE

The tinf_dot function shown in Listing 3 calculates the dot product (inner product), $r = x \cdot y$, of two arrays, x and y, and stores the value in a scalar, result. The output argument, result, is underlined in Listing 2 to distinguish it from the input arguments. The use of enums to select the data type and concurrency mode enables 48 different procedures to be executed.

Listing 3 $r = x \cdot y$.

```
int32_t tinf_dot ( const int32_t select_tinf_data_type,
        const int32_t select_tinf_concurrency_type, const int64_t n, const void *const x,
        const int64_t incx, const void *const y, const int64_t incy,
        const void *const iris_context,  const void *const iris_sync_pattern, void *const result )
```

select_tinf_data_type	identify the data type of the x array and result
select_tinf_concurrency_type	identify the concurrency execution type
n	number of scalar values in x
x	dense input array of size n
incx	spacing between scalar values in x
y	dense input array of size n
incy	spacing between scalar values in x
result	inner product of x and y, output
iris_context	IRIS communications context
iris_sync_pattern	IRIS communications synchronization pattern
returns	error code, TINF_SUCCESS or TINF_FAILURE

The tinf_matvec function in Listing 4 performs a sparse matrix-vector product operation, $y = Ax$, with a block-sparse row-major matrix,A, and dense column vector, x, and stores the result as a dense column vector, y. The output argument, y, is underlined in Listing 2 to distinguish it from the input arguments. The use of enums to select the data type, matrix storage format, and concurrency mode enables 384 different procedures to be executed. An introduction to the use of tinf_matvec is included in Section VI.

Listing 4 $y = Ax$.

```
int32_t tinf_matvec ( const int32_t select_tinf_data_type,
        const int32_t select_tinf_concurrency_type, const int32_t select_tinf_matrix_type,
        const int64_t num_rows, const int32_t block_size, const int64_t nnz,
        const void *const a_val, const int64_t *const index1, const int64_t *const index2,
        const void *const x, const void *const iris_context, const void *const iris_sync_pattern,
        void *const y )
```

select_tinf_data_type	identify the data type of the x array and result
select_tinf_concurrency_type	identify the concurrency execution type
select_tinf_matrix_type	identify the matrix storage type
num_rows	number of block rows in the A matrix
block_size	number of scalar rows in each block
nnz	number of non-zero blocks
a_val	contiguous array of matrix values
index1	stores the locations in the a_val array that start a major storage direction followed by nnz
index2	stores the minor storage direction indices of the blocks in the a_val array
x	dense input array of size num_rows∗block_size
y	dense output array of size num_rows∗block_size
iris_context	IRIS communications context
iris_sync_pattern	IRIS communications synchronization pattern
returns	error code, TINF_SUCCESS or TINF_FAILURE

B. Reordering

The three functions in Listing 5 define a generic and extensible interface to reordering operations. In Listing 5, the output arguments are underlined and the input/output arguments are italicized to distinguish them from the input arguments. SLAT's implementation of the reordering interface provides CMK and MISC and facilitates the evaluation of ongoing research [41] in applications. The ipar and dpar arrays are used to specify options (e.g., reverse) and return information from reordering operations (e.g., bandwidth). SLAT's implementation of the reordering interface requires ipar_length >= 16 and dpar_length >= 16 and specifies ipar[0] := starting node. Part of the design of the tinf_ interfaces allows other implementations to define larger arrays if necessary. The tinf_reordering_create function creates a reordering context for a particular reordering method (e.g., CMK) that is not tied to a particular single matrix. The tinf_reordering_generate_symbolic function is used to create the permutation arrays old_to_new and new_to_old for a particular sparsity pattern defined by the a_index1, a_index2, and a_diag arrays. The tinf_reordering_apply function can be used to apply the the permutation arrays and store the reordered matrix in the ra_index1, ra_index2, ra_diag, and ra_val arrays. A full description and exemplar use of the reordering interface will be included in the documentation when SLAT is released.

C. Preconditioners

The three functions in Listing 6, define a generic and extensible interface to preconditioning operations. In Listing 6, the output arguments are underlined and the input/output arguments are italicized to distinguish them from the input arguments. SLAT's implementation of the preconditioning interface provides symbolic ILU(k) fill-in, ILU, and LSBILU and facilitates the evaluation of ongoing research in applications. The ipar and dpar arrays are used to specify options (e.g., fill level) and return information from reordering operations (e.g., fill-ratio). SLAT's implementation of the reordering interface requires ipar_length >= 16 and dpar_length >= 16 and specifies ipar[0] := level of fill. Part of the design of the tinf_ interfaces allows other implementations to define larger arrays if necessary. The tinf_preconditioner_create function creates a preconditioning context for a particular reordering method (e.g., LSBILU) that is not tied to a particular single matrix. The tinf_preconditioner_generate_symbolic function is used to create the sparsity pattern of the preconditioner based upon the A matrix defined by the a_index1, a_index2, and a_val arrays. The tinf_preconditioner_initialize_numeric is used to initialize or reinitialize the preconditioning matrix with the values of the A matrix in the a_val array. The tinf_preconditioner_apply function can be used to apply the the preconditioner to the x array and store the result in the y array. An introduction to the use of the preconditioner interface is included in Section VI. A full description and exemplar uses of the preconditioner interface will be included in the documentation when SLAT is released.

D. Linear Solvers

The three functions in Listing 7 define a generic and extensible reverse communication interface (RCI) to linear solvers. In Listing 7, the output arguments are underlined and the input/output arguments are italicized to distinguish them from the input arguments. The linear solver interface has been included with the FUN3D 13.6-717bd48 release [62] along with an implementation of the interface that enables seamless run-time selection of linear solvers implemented within SLAT or those implemented within SPARSKIT. SLAT's implementation of the linear solver interface provides symbolic GMRES, Flexible GMRES with the choice of orthonormalization by Modified Gram-Schmidt or Householder Transforms methods and facilitates the evaluation of ongoing research in applications. The ipar and dpar arrays are used to specify options (e.g., Krylov dimension) and return information from reordering operations (e.g., residual norm). SLAT's implementation of the linear solver interface requires ipar_length >= 16 and dpar_length >= 16 and specifies ipar[0] := starting node. Part of the design of the tinf_ interfaces allows other implementations to define larger arrays if necessary. The tinf_linear_solver_rci_create function creates a linear solver context for a particular reordering method (e.g., GMRES) that is not tied to a particular linear system. The tinf_linear_solver_rci function can be used to progress through the reverse communication pattern. An introduction to the use of the linear solver interface is included in Section VI. A full description and exemplar uses of the linear solver interface will be included in the documentation when SLAT is released.

Listing 5   Reordering interface.

```
int32_t tinf_reordering_create( void **reord_context,
        const char *method_name, const int32_t method_name_length,
        const int32_t select_tinf_data_type, const int32_t select_tinf_matrix_type,
        int64_t *const ipar, const int32_t ipar_length,
        void *const dpar, const int32_t dpar_length,
        const int32_t global_id )

int32_t tinf_reordering_destroy( void **reord_context )

int32_t tinf_reordering_generate_symbolic( void *const reord_context, const int64_t num_rows,
        const int32_t block_size, const int64_t a_num_blocks,
        int64_t *const a_index1, int64_t *const a_index2, int64_t *const a_diag,
        int64_t *const ipar, void *const dpar, int64_t **old_to_new,  int64_t **new_to_old )

int32_t tinf_reordering_apply( void *const reord_context, const int64_t num_rows,
        const int32_t block_size, const int64_t a_num_blocks, int64_t *const ipar,
        void *const dpar, const int64_t *const old_to_new, const int64_t *const new_to_old,
        const void *const a_val, const int64_t *const a_index1, const int64_t *const a_index2,
        const int64_t *const a_diag, void **ra_val, int64_t **ra_index1, int64_t **ra_index2,
        int64_t **ra_diag )
```

reord_context	reordering context
method_name	reordering method name, eg. cmk
method_name_length	length of the method_name (including null terminator)
select_tinf_data_type	identify the data type of the dpar array
select_tinf_matrix_type	identify the matrix storage type
ipar	integer parameter array for the reordering operation
ipar_length	length of the ipar array
dpar	data parameter array storing inputs and outputs from the reordering operation
dpar_length	length of the dpar array
global_id	partition id
num_rows	number of block rows in linear system
block_size	number of scalar rows in each block
a_num_blocks	number of blocks in linear system
a_index1	stores the locations in the a_val array that start a row
a_index2	stores the column indices of the blocks in the a_val array
a_diag	stores the indices of the main diagonal blocks in the a_val array
a_val	contiguous array of A matrix values
old_to_new	permutation array of row indices that maps from the original ordering to the new ordering
new_to_old	permutation array of row indices that maps from the new ordering to the new original
ra_index1	stores the locations in the reordered a_val array that start a row
ra_index2	stores the column indices of the blocks in the reordered a_val array
ra_diag	stores the indices of the main diagonal blocks in the reordered a_val array
ra_val	contiguous array of reordered A matrix values
returns	error code, TINF_SUCCESS or TINF_FAILURE

Listing 6 Preconditioner interface.

```
int32_t tinf_preconditioner_create( void **precond_context, const char *method_name,
        const int32_t method_name_length, const int32_t select_tinf_data_type,
        const int32_t select_tinf_concurrency_type, const int32_t select_tinf_matrix_type,
        int64_t *const ipar, const int32_t ipar_length, void *const dpar,
        const int32_t dpar_length, const void *const iris_context,
        const void *const iris_sync_pattern, const int32_t global_id )

int32_t tinf_preconditioner_destroy( void **precond_context, const int32_t block_size )

int32_t tinf_preconditioner_generate_symbolic( void *const precond_context, int64_t *const ipar,
        void *const dpar, const int64_t num_rows, const int64_t num_cols,const int32_t block_size,
        const int64_t a_num_blocks, const void *const a_val, const int64_t *const a_index1, const
            int64_t *const a_index2 )

int32_t tinf_preconditioner_initialize_numeric( void *const precond_context, int64_t *const ipar,
        void *dpar, const void *const a_val )

int32_t tinf_preconditioner_update_numeric( void *const precond_context, int64_t *const ipar,
        void *const dpar, const void *const a_val )

int32_t tinf_preconditioner_apply( void *const precond_context, int64_t *const ipar,
        void *const dpar, const int64_t num_rows, const void *const x, void *const y )
```

precond_context	preconditioner context
method_name	preconditioner method name, eg. iluk
method_name_length	length of the method_name (including null terminator)
select_tinf_data_type	identify the data type of the dpar and a_val arrays
select_tinf_concurrency_type	identify the concurrency execution type
select_tinf_matrix_type	identify the matrix storage type
ipar	integer parameter array for the preconditioner
ipar_length	length of the ipar array
dpar	data parameter array storing inputs and outputs from preconditioner
dpar_length	length of the dpar array
iris_context	IRIS communications context
iris_sync_pattern	IRIS communications synchronization pattern
global_id	partition id
num_rows	number of block rows in A matrix
num_cols	number of block cols in A matrix
natural_block_size	number of scalar rows in each block of the original A matrix
a_num_blocks	number of blocks in A matrix
a_index1	stores the locations in the a_val array that start a row
a_index2	stores the column indices of the blocks in the a_val array
a_diag	stores the indices of the main diagonal blocks in the a_val array
a_val	contiguous array of A matrix values
x	dense input array of size num_rows∗natural_block_size
y	dense output array of size num_rows∗natural_block_size
returns	error code, TINF_SUCCESS or TINF_FAILURE

Listing 7 Linear solver interface.

```
int32_t tinf_linear_solver_rci_create ( void **linear_solver_context,
        const char *const method_name, const int32_t method_name_length,
        const int32_t select_tinf_data_type,
        const int64_t num_rows, const int32_t block_size,
        int64_t *const ipar, const int32_t ipar_length,
        void *const dpar, const int32_t dpar_length,
        void *const iris_context, void *const iris_sync_pattern,
        void **work_space )

int32_t tinf_linear_solver_rci_destroy ( void **linear_solver_context, void **work_space )

int32_t tinf_linear_solver_rci ( void *const linear_solver_context, const int64_t num_rows,
        const int32_t block_size, void *const rhs, void *const sol, int64_t *const ipar,
        void *const dpar, void *const work_space )
```

linear_solver_context	linear system solver context
method_name	linear system solver method name, eg. gmres, fgmres, or jacobi
method_name_length	length of the method_name (including null terminator)
select_tinf_data_type	identify the data type of the dpar, work_space, rhs, and sol arrays
num_rows	number of block rows in linear system
block_size	number of scalar rows in each block
rhs	right-hand-side array
sol	solution array
ipar	integer parameter array for the reverse-communication protocol
ipar_length	length of the ipar array
dpar	data parameter array storing inputs and outputs from a linear solver
dpar_length	length of the dpar array
work_space	array utilized by linear solver for intermediate calculations
iris_context	IRIS communications context
iris_sync_pattern	IRIS communications synchronization pattern
returns	error code, TINF_SUCCESS or TINF_FAILURE

V. Results

The data presented in this work are from computations performed on the NASA Langley K4 and NASA HECC Electra computing clusters. Intel® 20-Core Xeon® Gold 6148 processors [63], commonly referred to as Skylake CPUs, are utilized on a 4x EDR Infiniband interconnect ($\sim 100 Gbits/s$) [64].

A. Forward CFD Analysis

The computational fluid dynamics problem presented in this work is Case 2 from the 1st AIAA CFD High Lift Prediction Workshop [65]. The case embodies a subsonic aerodynamic flowfield associated with three-dimensional, swept, medium-to-high aspect ratio multielement wings required by commercial and military transport aircraft for landing and takeoff [66]. The mesh chosen is small enough to enable expedient evaluation of the effectiveness of the preconditioners across a range of partition sizes from an average of as few as 130 mesh points on each of 5120 partitions to 328202 mesh points on two partitions. The flight conditions and mesh specifications are detailed in Table 5. Figure 3 depicts the pressure coefficient, C_p, distribution on the surface of the trapezoidal wing body geometry and Mach number field on the symmetry plane from a simulation converged until the RMS residual is less than 5.E−13. Significant flow features such as wake/boundary layer interactions and regions of separated flow are present in the simulation.

Table 5 Trapezoidal Wing "Config 1" - Slat 30°, Flap 25°.

Flight Conditions		Mesh	
Mach	0.2	Number of tet cells	3, 852, 709
Angle-of-attack [°]	13	Number of mesh points	656, 403
Reynolds number [based on MAC]	4.3E6	number of edges	4, 538, 241
Reference Temperature [R]	520	Number of boundaries	26

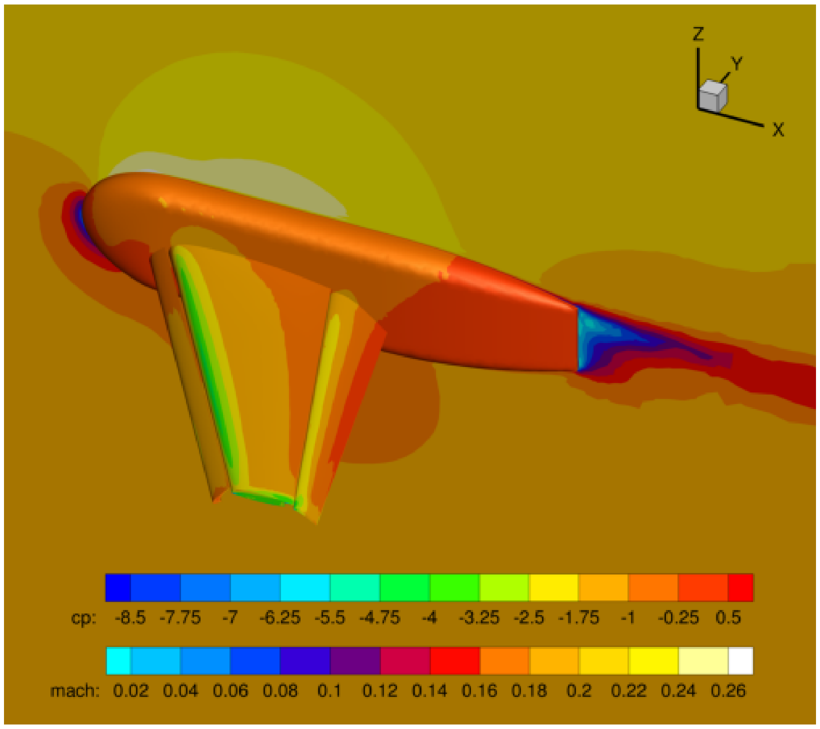

Z
Y
X
cp: -8.5 -7.75 -7 -6.25 -5.5 -4.75 -4 -3.25 -2.5 -1.75 -1 -0.25 0.5
mach: 0.02 0.04 0.06 0.08 0.1 0.12 0.14 0.16 0.18 0.2 0.22 0.24 0.26

Fig. 3 Visualization of C_p distribution on the surface of the trap wing geometry and Mach number field on the symmetry plane.

B. Strong Scaling

Two views of runtime details from the baseline simulation using unthreaded BILU preconditioning (80 MPI partitions, 1 CPU core per partition) are shown in Figure 4. The nonlinear and linear solver performance are depicted in Figure 4a as represented by the RMS residual, GMRES search directions, and Courant–Friedrichs–Lewy (CFL) number per nonlinear iteration. The RMS residual of the solution falls below 5.E−13 after 123 nonlinear iterations at a wall time of 53763 [s] (89.60 minutes). GMRES is used to solve the linear subproblem for the initial nonlinear update during each nonlinear iteration. A Krylov subspace of 300 search directions is used and at most one restart is allowed. GMRES terminates when the linear residual reaches an absolute tolerance of 1.E−15 or a relative tolerance of 1.E−8 or after a maximum of 600 search directions. GMRES fails to converge to a relative tolerance of 0.5 during nonlinear iteration 83 and the update is rejected. The CFL history depicts the global multiplier to local element time-step parameters. The CFL number is an indictor of the size of the nonlinear iteration taken toward a steady-state solution. The maximum CFL allowed during the simulation is 1.E6. The wall time required per nonlinear iteration for selected phases of FUN3D-SFE's solution process are shown in Figure 4b. Residual calculation is depicted by a red line with square markers. The assembly of the linearized coefficient matrix for the left-hand side of the equations is depicted by a grey line with right facing triangle markers. Timing for both residual calculation and the assembly of the left-hand side are stable throughout the simulation because the work load and distribution of the work load are constant. These two phases form the linear problem and comprise the temporal context to evaluate the impacts of preconditioning methods. Timing of GMRES core calculations is plotted by an orange line with left facing triangle markers. This timing encompasses the time required to orthonormalize the Krylov search space through a modified Gram-Schmidt Arnoldi process, update the Hessenberg matrix by applying Givens rotation matrices, and the final backward-substitution of the upper Hessenberg matrix to obtain the updated linear solution vector. Preconditioning of each GMRES search direction and matrix-vector products required for the linear solution are timed separately to facilitate comparison.

Preconditioning operations are parsed into three steps delineated as,

symbolic: creation of the sparsity pattern and storage for a partition's preconditioning matrix via ILU($k = 2$) fill

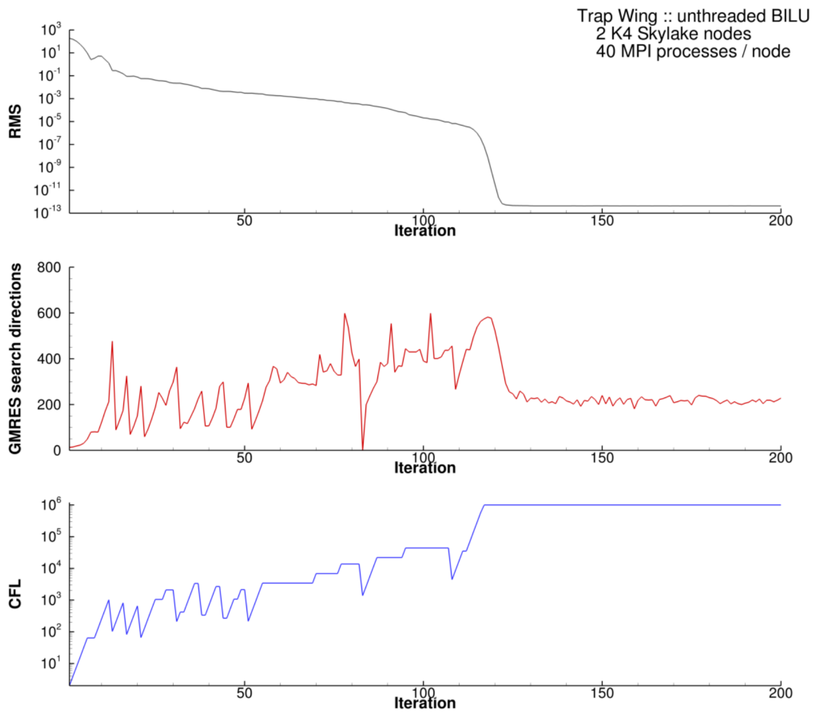
update: computation of the numeric values of the incomplete lower upper factorization

apply: application of the incomplete lower upper factorization to a vector via a forward and a backward-substitution

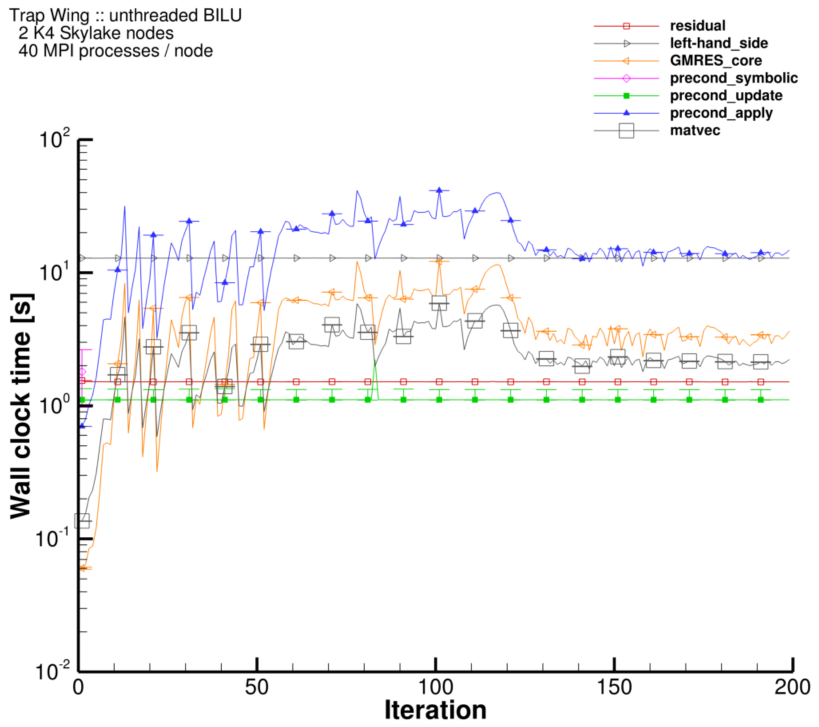
The symbolic operations to form the sparsity pattern for a preconditioning matrix are performed once at the start of the simulation, indicated by the pink circle at the first iteration. The time required to initialize preconditioning matrices at each nonlinear iteration is indicated by a red line with circle markers. The time required to update a preconditioning matrix at each nonlinear iteration is indicated by a green line with filled square markers. The time required to apply a preconditioning matrix at each nonlinear iteration is indicated by a green line with filled upward triangle markers. The time required for matrix-vector products at each nonlinear iteration is indicated by a grey line with large square markers. Error bars indicate the range of time required across the 80 partitions for each solution phase.

The time required to initialize and update a partition's preconditioning matrix once per nonlinear iteration is independent of the number of GMRES search directions. There are linear relationships between the number of GMRES search directions and the time required to perform matrix-vector products and apply the BILU preconditioner, each occurs once per search direction. The time required for GMRES core operations is related quadratically to the number of GMRES search directions during a nonlinear iteration because each search direction is orthonormalized against all prior search directions. The timings shown in Figure 4 indicate that the three largest portions of wall time are dedicated to BILU preconditioner application, left-hand side updates, and SPARSKIT's GMRES operations for the baseline parallelization strategy of 80 MPI partitions each using one CPU core.

Figure 5 presents an overview of FUN3D-SFE's performance when employing unthreaded BILU preconditioning for a range of parallelization strategies. One CPU core is assigned to each MPI partition to perform the sequential operations of BILU preconditioning. In Figure 5a, the variation of wall time, the total number of GMRES search directions, and the nonlinear iterations required to converge the solution are plotted against the number of CPU cores utilized. The trends indicate that as the number of partitions and CPU cores utilized increases (mesh points per partition decreases) the wall time to solution decreases with diminishing returns. The departure from ideal strong scaling is influenced by the cost of MPI communication during each nonlinear iteration and the increase in the total number of GMRES search directions required. The portion of time spent in solution phases relative to the wall time to solution is plotted in Figure 5b.

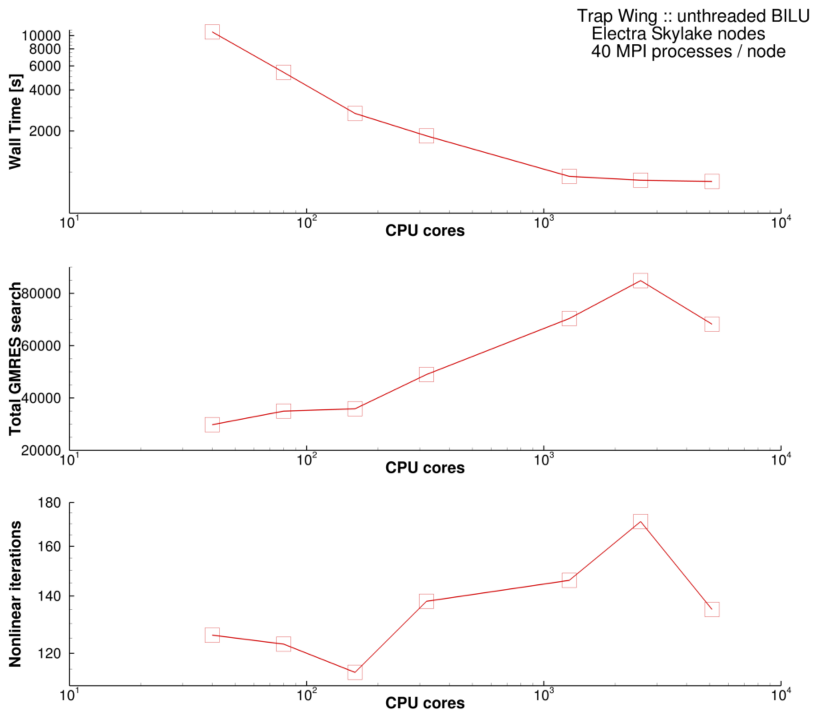


(a) Histories of the RMS nonlinear residual, number of GMRES search directions, and CFL number during 200 nonlinear iterations.

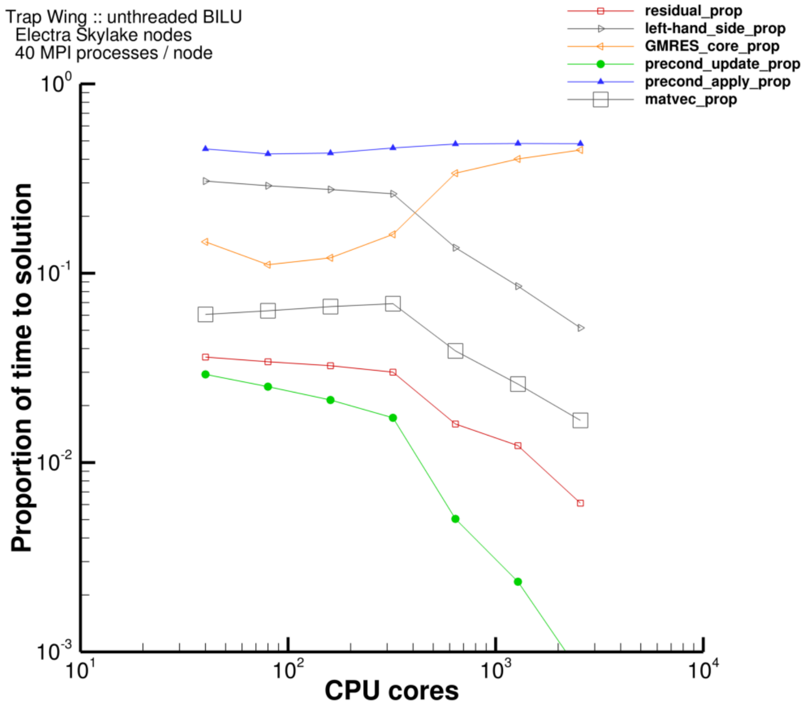


(b) Wall time of selected solution phases per nonlinear iteration. Symbols mark the mean value and bars mark the minimum and maximum among the 80 partitions.

Fig. 4 Visualization of FUN3D-SFE performance characteristics from a steady state simulation of the trapeziodal wing configuration 1 case. The solutions were obtained using SPARSKIT's GMRES and BILU preconditioning (See Algorithm 1) on 80 MPI partitions each employing one CPU core. Progress toward convergence is depicted in (a) and the wall time required by solution phases is plotted in (b).



(a) Variation of wall time, nonlinear iterations, and total GMRES search directions required to converge the solution with the number of CPU cores.



(b) Relative proportion of the wall time to solution spent in each phase of a nonlinear iteration.

Fig. 5 Two perspectives summarizing how FUN3D-SFE calling SPARKIT's GMRES strong scales when employing unthreaded BILU preconditioning (Algorithm 1): (a), the total GMRES search directions and nonlinear iterations depict the variation in the workload as partition size decreases, and (b), the nonlinear workload is parsed into solution phases to identify performance bottlenecks as partition size decreases.

Figure 6 presents an overall comparison of strong scaling when using the implementations of GMRES in SPARSKIT and SLAT. FUN3D-SFE solves the 16-degree angle of attack trap wing case from the first high lift workshop 49% faster on 40 CPU cores when utilizing the GMRES implementation within SLAT than when utilizing SPARSKIT's GMRES. On 1280 CPU cores, FUN3D-SFE converges 56.6% faster when calling GMRES from SLAT than from SPARSKIT.

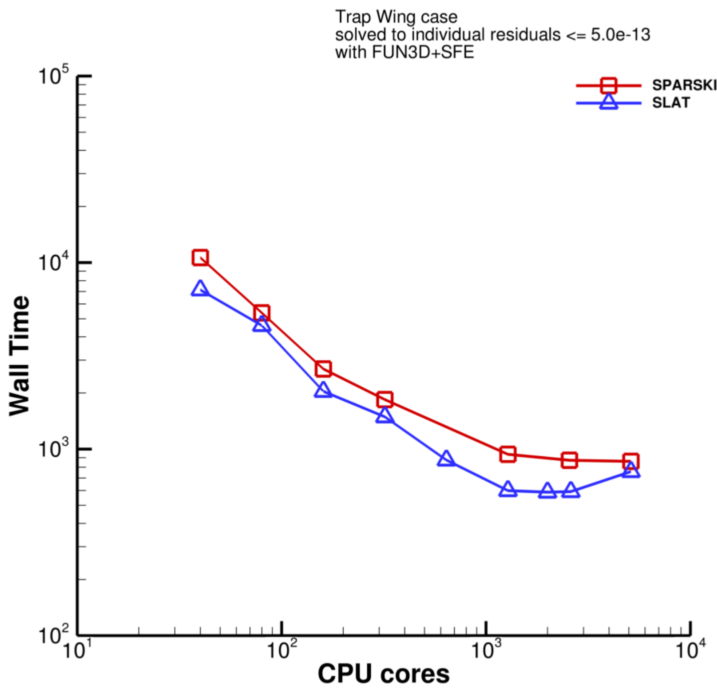


Fig. 6 Comparison of wall time to solution using unthreaded BILU preconditioning and GMRES implementations from SPARSKIT and SLAT within FUN3D-SFE.

C. Adjoint, Complex-step, & Surreal verification

A brief summary of an adjoint problem is provided in this work to discuss the requirements placed upon a linear solver. Readers interested in a deeper discussion of adjoint CFD applications are directed to Refs. [26, 67]. The adjoint solution, λ, for a particular output functional I, is obtained by solving the linear system of equations

$$\left[\frac{\partial R_h}{\partial q_h}\right]^T \lambda = -\left(\frac{\partial I}{\partial q_h}\right)^T \tag{13}$$

where q_h is the set of solution state variables, and R_h is the discrete residual of the partial differential equations under consideration. The implementations of complex-step and surreal operations within SLAT's preconditioners and linear solvers are verified with adjoint solutions produced by solving Eq. 13 for the sensitivity of the z force component to the state variables in four simulations. Directional derivatives of the z force component along a random direction of the states are calculated with complex-step, surreal, and real-valued finite-difference methods. The same random directions are used for all three mothods. A complex source term, h =1.0E-30, is used to perturb the flow equations in the complex-step method. A derivative source term, h =1.0E-30, is used to perturb the flow equations in the surreal method. A real-valued source term, x =1.0E-8, is used to perturb the flow equations in the real-valued finite-difference method. The results presented in Table 6 cover inviscid, laminar, and turbulent flow regimes with geometry of moderate and high complexity from three test cases, and exercise both strong and weak boundary conditions. The first test case is VERIF/3DB, a 3D bump in channel [68], the second test case is Onera M6 [69], and the third test case is the JAXA Standard Model case 2c [70]. It is noteworthy that the linear solver and preconditioner operations are also stress-tested by adjoint systems. These adjoint systems are exceptionally stiff due to an effective CFL of ∞ which negates the boost to diagonal dominance provided by the pseudotime term. The results in Table 6 show a minimum of 8 significant digits of agreement between adjoint, complex-step, and surreal solutions, along with the expected limited agreement with finite-difference approximation of the sensitivity of the z force component to the residual. Bold digits indicate agreement with adjoint results.

Table 6 Adjoint Verification.

Adjoint	Complex-step	Surreal	Finite-difference
	h =1.0E-30	h =1.0E-30	x =1.0E-8
VERIF/3DB Re=1.0E+02, Weak BC			
5.7284164824E+04	**5.7284164**542E+04	**5.72841648**01E+04	**5.728**1302000E+04
VERIF/3DB Re=1.0E+02, Strong BC			
-5.6696566536E+05	**5.6696566536**E+05	**5.6696566536**E+05	**5.6**703983900E+05
VERIF/3DB Re=3.0E+06, Weak BC			
5.5016646209E+04	**5.50166462**34E-26	**5.50166462**64E-26	**5**.4370470300E+04
VERIF/3DB Re=3.0E+06, Strong BC			
5.2945989606E+04	**5.294598960**2E-26	**5.294598960**3E-26	**5**.1965968500E+04
Onera M6 Inviscid Re=4.6E+05, Weak BC			
2.4564327703E+03	**2.4564327703**E-27	**2.4564327703**E+03	**2.456**5248712E+03
JAXA Standard Model Re=3.647E+06 Strong BC			
1.0233096937E+08	**1.023309693**4E+08	**1.023309693**8E+08	**1.0**120000087E+08

D. Linearized Frequency-domain Method

A brief summary of the Linearized Frequency-Domain (LFD) method is provided in this work to discuss the requirements placed upon a linear solver. Readers interested in LFD applications are directed to Ref. [27]. The LFD method provides accurate flutter predictions at reduced computational costs compared to time-accurate CFD simulations. The LFD method assumes small, harmonic perturbations about the equilibrium RANS solution. Based upon this assumption, an exact linearization of the governing residual is formed,

$$\left(i\omega \mathrm{M} + \left.\frac{\partial R}{\partial q}\right|_{X_{G0},q_0} \right) \hat{q} = - \left.\frac{\partial R}{\partial X_G}\right|_{X_{G0},q_0} \hat{X}_G - iw \left.\frac{\partial R}{\partial \dot{X}_G}\right|_{X_{G0},q_0} \hat{X}_G \,, \tag{14}$$

where, $\hat{X}_G$ is the harmonic mesh motion due to a perturbation of a structural mode, ω is the frequency at which structural mode is oscillated, and $\hat{q}$ is the frequency-domain flow state (Fourier coefficient), The linear system defined in Eq. 14 is complex-valued and is solved once per frequency of interest and perturbed mode pair.

Fourier transformations of forced motion time-domain simulations of an NACA 64A010 airfoil are used to verify the LFD implementation as shown in Figure 7 and Table 7. The forced motion amplitude is small, modal displacements of 0.001, in order to approximate the linear response with the nonlinear time-domain solver. The Fourier coefficients of the flow field and generalized aerodynamic forces both match at least to the precision of the amplitude of the force motion. This verifies the implementation and linear solver for the complex-valued frequency-domain problem.

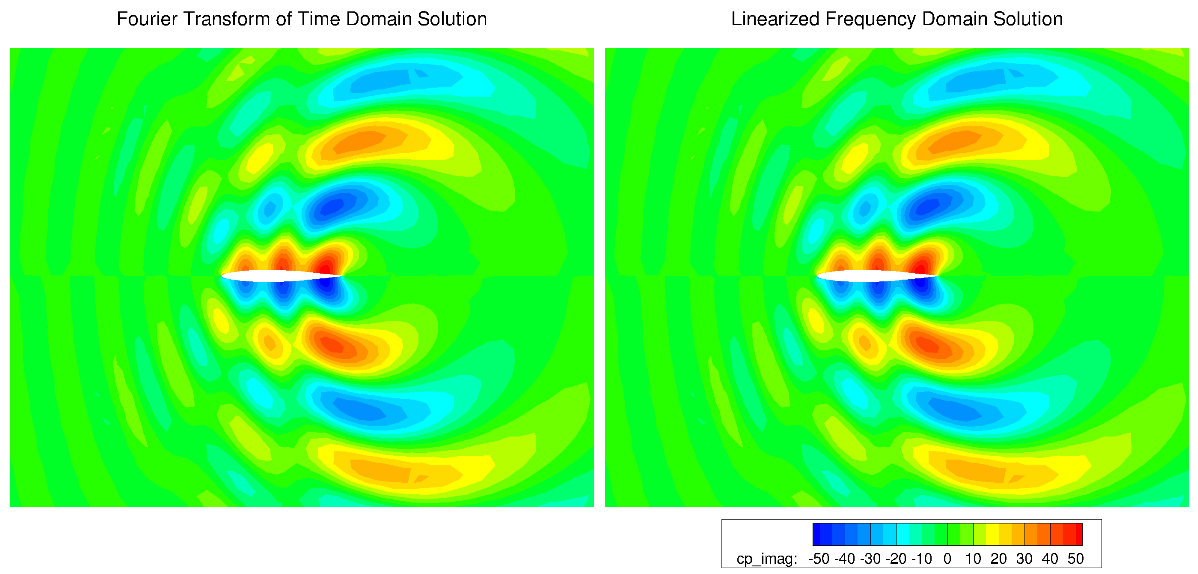


Fig. 7 Visualization of imaginary part of the Fourier coefficients of pressure perturbations around an NACA 64A010 airfoil predicted by time-domain and LFD methods.

Table 7 Comparison of generalized aerodynamic forces from LFD and time-domain simulations for an NACA 64A010 airfoil with pitch and plunge degrees of freedom.

A11	GAF value	Amplitude	Phase[°]
Time-domain least-squares fit	-0.062034714+0.009486982j	0.062755945	171.305110883
Linearized frequency-domain	-0.061998716+0.009479661j	0.062719253	171.306750664
A12	GAF value	Amplitude	Phase[°]
Time-domain least-squares fit	0.309573196-0.236955257j	0.389850430	-37.431369012
Linearized frequency-domain	0.309443029-0.236805496j	0.389656042	-37.425516008
A21	GAF value	Amplitude	Phase[°]
Time-domain least-squares fit	-0.003903398-0.020899123j	0.021260523	-100.579426188
Linearized frequency-domain	-0.003914324-0.020910025j	0.021273248	-100.602961521
A22	GAF value	Amplitude	Phase[°]
Time-domain least-squares fit	0.099903089+0.096984878j	0.139236108	44.150842391
Linearized frequency-domain	0.099990495+0.096988284j	0.139301207	44.126806007

VI. Sample code

The error checking macro in Listing 8 can be used to capture the return values from interface functions and produce feed back that aids debugging.

Listing 8 Error Checking Macro.

```
#ifdef _DEBUG
#define SLAT_CHECK(err)                                                         \
do                                                                              \
{                                                                               \
  slat_int_t e_ = (err);                                                        \
  if (e_ != 0)                                                                  \
  {                                                                             \
    fprintf(stderr, "SLAT_CHECK ERROR %d : (%s, line %d)\n", e_, __FILE__,     \
            __LINE__);                                                          \
    exit(-1);                                                                   \
  }                                                                             \
} while (0)
#else
#define SLAT_CHECK(err)
#endif
```

The code snippet in Listing 9 uses the preconditioner, matrix-vector product, $L2$ norm, and linear solver interfaces to solve a linear system. Sample screen output from Listing 9 is shown in Listing 10.

Listing 9 Preconditioned GMRES snippet.

```
int32_t dtype = TINF_DOUBLE;
int32_t concurtype = TINF_MPI;
int32_t mattype = TINF_BSR;
void *iris_handle{nullptr};
void *iris_sync_pattern{nullptr};
void *workPtr = nullptr;
double *workPtr_ = nullptr;

int64_t n = num_rows * block_size;
double *x{nullptr};
x = (double *) calloc(static_cast<size_t>(n), sizeof(double));
//
// Create ILU(K) preconditioner
//
SLAT_CHECK( tinf_preconditioner_create(preconditioner, "iluk", 5, dtype, concurtype, mattype,
    precond_ipar, precond_dpar, iris_handle,  iris_sync_pattern, my_rank, &precond_handle) );
SLAT_CHECK( tinf_preconditioner_generate_symbolic( precond_handle, precond_ipar, precond_dpar,
     a_num_rows, a_num_cols, a_block_size, a_num_blocks, a_val, a_row, a_col) );
SLAT_CHECK( tinf_preconditioner_initialize_numeric(precond_handle, precond_ipar, precond_dpar
    a_val) );
SLAT_CHECK( tinf_preconditioner_update_numeric(precond_handle, precond_ipar, precond_dpar,
    a_val) );
//
// Create Preconditioned GMRES linear solver
//
SLAT_CHECK(tinf_linear_solver_rci_create(&gmres_context, "gmres", 6, dtype, num_rows,
    block_size, ipar, 16, dpar, 16, iris_handle, iris_sync_pattern, &workPtr) );
workPtr_ = static_cast<double *>(workPtr);
//
// Perform Preconditioned GMRES until convergence criteria or maximum number of matvecs
//
while (ipar[RCI_INDEX] > 0)
{
  SLAT_CHECK( tinf_linear_solver_rci(gmres_context, num_rows, block_size, RHS, x, ipar, dpar,
      workPtr) );

  switch (ipar[RCI_INDEX])
  {
  case USER_MATRIX_VECTOR_PRODUCT:
```

```
  {
    //
    //  Matrix vector product
    //
    SLAT_CHECK( tinf_matvec(dtype, concurtype, mattype, a_num_rows, a_block_size, a_nnz,a_val,
             a_row, a_col, &(workPtr_[ipar[MATVEC_INPUT_INDEX]]),
             &(workPtr_[ipar[MATVEC_OUTPUT_INDEX]])) );
    break;
  }
  case USER_RIGHT_PRECONDITIONER_APPLICATION:
  {
    //
    //  Apply Right Preconditioning
    //
    SLAT_CHECK( tinf_preconditioner_apply(precond_handle, precond_ipar, precond_dpar,
        a_num_cols, &(workPtr[ipar[MATVEC_OUTPUT_INDEX]]),
        &(workPtr[ipar[MATVEC_INPUT_INDEX]])) );
    std::cout << "Search direction " << std::setw(7)
              << ipar[MATVEC_COUNT_INDEX] << " residual = " << std::scientific << std::
                  setprecision(10) << std::setw(19)
              << dpar[CURRENT_RESIDUAL_INDEX] << " rate = " << std::scientific << std::
                  setprecision(10) << std::setw(19)
              << dpar[CONVERGENCE_RATE_INDEX] << std::endl;
    break;
  }
  }
}
//
// Calculate actual residual
//
for (int64_t i = 0; i < n; ++i)
{
  workPtr_[ipar[MATVEC_OUTPUT_INDEX] + i] = 0.0;
}
SLAT_CHECK( tinf_matvec(dtype, concurtype, mattype, a_num_rows, a_block_size, a_nnz, a_val,
    a_row, a_col, x, &(workPtr_[ipar[MATVEC_OUTPUT_INDEX]])) );
for (int64_t i = 0; i < num_rows * block_size; ++i)
{
  workPtr_[ipar[MATVEC_OUTPUT_INDEX] + i] =
      RHS[i] - workPtr_[ipar[MATVEC_OUTPUT_INDEX] + i];
}
double actual_residual = 0.0;
SLAT_CHECK( tinf_norm2(dtype, concurtype, n, &(workPtr_[ipar[MATVEC_OUTPUT_INDEX]]), 1,
    iris_handle,  iris_sync_pattern, static_cast<void*>(&actual_residual)) );
std::cout << "Final Search direction " << std::setw(7)
          << ipar[MATVEC_COUNT_INDEX] << " residual = " << std::scientific
          << std::setprecision(10) << std::setw(19)
          << dpar[CURRENT_RESIDUAL_INDEX] << " rate = " << std::scientific
          << std::setprecision(10) << std::setw(19)
          << dpar[CONVERGENCE_RATE_INDEX]
          << " actual residual = " << std::scientific << std::setprecision(10)
          << std::setw(19) << actual_residual
          << " actual rate = " << std::scientific << std::setprecision(10)
          << std::setw(19) << actual_residual / dpar[INITIAL_RESIDUAL_INDEX]
          << std::endl;

SLAT_CHECK( tinf_preconditioner_destroy(&precond_handle, a_block_size) );
SLAT_CHECK( tinf_linear_solver_rci_destroy(&gmres_context, &workPtr) );
free(x);
```

Listing 10 Preconditioned GMRES output.

```
Search direction       1 residual =    2.2737634002e+01 rate =    1.0000000000e+00
Search direction       2 residual =    3.1534083154e+00 rate =    1.3868673914e-01
Search direction       3 residual =    9.0634257238e-01 rate =    3.9860900756e-02
Search direction       4 residual =    1.8851084760e-01 rate =    8.2906975979e-03
Search direction       5 residual =    7.9700723187e-02 rate =    3.5052337979e-03
Search direction       6 residual =    7.9700723187e-02 rate =    3.5052337979e-03
Search direction       7 residual =    2.6208388113e-02 rate =    1.1526435913e-03
Search direction       8 residual =    1.2166420496e-02 rate =    5.3507856159e-04
Search direction       9 residual =    5.7675401363e-03 rate =    2.5365612517e-04
Search direction      10 residual =    2.4035707720e-03 rate =    1.0570892168e-04
Search direction      11 residual =    2.4035707720e-03 rate =    1.0570892168e-04
Search direction      12 residual =    7.0467717035e-04 rate =    3.0991666516e-05
Search direction      13 residual =    3.6316902968e-04 rate =    1.5972155663e-05
Search direction      14 residual =    1.9115884718e-04 rate =    8.4071564862e-06
Search direction      15 residual =    4.7455575799e-05 rate =    2.0870938372e-06
Search direction      16 residual =    4.7455575799e-05 rate =    2.0870938372e-06
Search direction      17 residual =    1.8886082400e-05 rate =    8.3060895421e-07
Search direction      18 residual =    6.2855803720e-06 rate =    2.7643950868e-07
Search direction      19 residual =    2.7976821341e-06 rate =    1.2304191957e-07
Search direction      20 residual =    1.1897224964e-06 rate =    5.2323935564e-08
Search direction      21 residual =    1.1897224958e-06 rate =    5.2323935538e-08
Search direction      22 residual =    5.0648136309e-07 rate =    2.2275024879e-08
Search direction      23 residual =    2.0488648853e-07 rate =    9.0108974625e-09
Final Search direction       23 residual =    2.0488648853e-07 rate =    9.0108974625e-09
   actual residual =    2.0488648734e-07 actual rate =    9.0108974100e-09
```

VII. Summary

The Sparse Linear Algebra Toolkit (SLAT) for Computational Aerodynamics has been introduced. Pivotal design and implementation choices to provide a light-weight and extensible toolkit for computational research and production simulations have been highlighted and their accuracy verified. Support for the data types and operations required for forward CFD analysis, sensitivity analysis, design, adaptation, and linearized frequency-domain applcations have been verified. Functionality not presently available in five sources of prevalent linear solver routines, MKL 2019 [19], CUDA10.2.89 [20], SPARSKIT [15], MAGMA [16, 17], and PETSc [18], have been highlighted. The performance of Reynods-averaged Navier-Stokes simulations conducted with FUN3D's Stabilized Finite Element (SFE) library and SLAT have been evaluated and shown to be between 15% and 56% faster than those conducted with SPARSKIT. The linear solver interface presented in this work has been included with the FUN3D 13.6-717bd48 release [62]. The interfaces for linear algebra, reordering, and preconditioning operations introduced in this work will be included in future releases of FUN3D in concert with the broader effort to provide interfaces for computational engineering components [55, 56].

Acknowledgments

This research was sponsored by the NASA Revolutionary Computational Aerosciences (RCA) Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate.

References

[1] Hill, M. D., "What is Scalability?" *SIGARCH Comput. Archit. News*, Vol. 18, No. 4, 1990, pp. 18–21. doi:10.1145/121973.121975, URL `http://doi.acm.org/10.1145/121973.121975`.

[2] Bondi, A. B., "Characteristics of Scalability and Their Impact on Performance," *Proceedings of the 2Nd International Workshop on Software and Performance*, ACM, New York, NY, USA, 2000, pp. 195–203. doi:10.1145/350391.350432, URL `http://doi.acm.org/10.1145/350391.350432`.

[3] Shang, Z., "Impact of Mesh Partitioning Methods in CFD for Large Scale Parallel Computing," *Computers & Fluids*, Vol. 103,

2014, pp. 1 – 5. doi:https://doi.org/10.1016/j.compfluid.2014.07.016, URL http://www.sciencedirect.com/science/article/pii/S0045793014003016.

[4] Kogge, P., and Shalf, J., “Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture,” *Computing in Science & Engineering*, Vol. 15, No. 6, 2013, pp. 16–26.

[5] Wang, Y., Zhang, L., Liu, W., Cheng, X., Zhuang, Y., and Chronopoulos, A. T., “Performance Optimizations for Scalable CFD Applications on Hybrid CPU+MIC Heterogeneous Computing System with Millions of Cores,” *CoRR*, Vol. abs/1710.09995, 2017. URL http://arxiv.org/abs/1710.09995.

[6] Reguly, I. Z., Mudalige, G. R., Bertolli, C., Giles, M. B., Betts, A., Kelly, P. H. J., and Radford, D., “Acceleration of a Full-Scale Industrial CFD Application with OP2,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 5, 2016, pp. 1265–1278. doi:10.1109/TPDS.2015.2453972.

[7] Gustafson, J. L., “Reevaluating Amdahl’s Law,” *Commun. ACM*, Vol. 31, No. 5, 1988, pp. 532–533. doi:10.1145/42411.42415, URL http://doi.acm.org/10.1145/42411.42415.

[8] Anderson, W. K., Newman, J. C., and Karman, S. L., “Stabilized Finite Elements in FUN3D,” *Journal of Aircraft*, Vol. 55, No. 2, 2018, pp. 696–714. doi:10.2514/1.C034482.

[9] Alauzet, F., and Frazza, L., *3D RANS anisotropic mesh adaptation on the high-lift version of NASA’s Common Research Model (HL-CRM)*, 2019. doi:10.2514/6.2019-2947, URL https://arc.aiaa.org/doi/abs/10.2514/6.2019-2947.

[10] Saad, Y., and Schultz, M. H., “GMRES: A Generalized Minimum Residual Algorithm for Solving Nonsymmetric Linear Systems,” *SIAM Journal of Scientific and Statistical Computing*, Vol. 7, 1986, pp. 856–869.

[11] Saad, Y., *Iterative Methods for Sparse Linear Systems*, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[12] Saad, Y., “Preconditioned Krylov subspace methods for CFD applications,” *Solution Techniques for Large-Scale CFD Problems*, 1995, pp. 139–158.

[13] Zhang, J., “Preconditioned Krylov Subspace Methods for Solving Nonsymmetric Matrices from CFD Applications,” *Computer methods in applied mechanics and engineering*, Vol. 189, No. 3, 2000, pp. 825–840.

[14] Benzi, M., “Preconditioning Techniques for Large Linear Systems: A Survey,” *Journal of computational Physics*, Vol. 182, No. 2, 2002, pp. 418–477.

[15] “SPARKSIT,” , Nov 2019. URL https://www-users.cs.umn.edu/~saad/software/SPARSKIT/.

[16] Anzt, H., Sawyer, W., Tomov, S., Luszczek, P., Yamazaki, I., and Dongarra, J., “Optimizing Krylov Subspace Solvers on Graphics Processing Units,” *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*, IEEE, IEEE, Phoenix, AZ, 2014.

[17] Yamazaki, I., Anzt, H., Tomov, S., Hoemmen, M., and Dongarra, J., “Improving the performance of CA-GMRES on multicores with multiple GPUs,” *IPDPS 2014*, IEEE, Phoenix, AZ, 2014.

[18] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H., “PETSc Web page,” https://www.mcs.anl.gov/petsc, 2019. URL https://www.mcs.anl.gov/petsc.

[19] Intel®, “Developer Reference for Intel® Math Kernel Library 2019 - C,” , Nov 2019. URL https://software.intel.com/en-us/mkl-developer-reference-c.

[20] “CUDA Toolkit Documentation v10.2.89,” , Nov 2019. URL https://docs.nvidia.com/cuda/index.html#.

[21] Brooks, A. N., and Hughes, T. J. R., “Streamline Upwind/Petrov-Galerkin Formulation for Convection Dominated Flows with Particular Emphasis on Incompressible Navier-Stokes Equations,” *Computer Methods in Applied Mechanics and Engineering*, Vol. 32, No. 1–3, 1982, pp. 199–259. doi:10.1016/0045-7825(82)90071-8.

[22] Shakib, F., Hughes, T. J. R., and Johan, Z., “A New Finite-Element Formulation for Computational Fluid Dynamics: X. The Compressible Euler and Navier-Stokes Equations,” *Computer Methods in Applied Mechanics and Engineering*, Vol. 89, No. 1–3, 1991, pp. 141–219. doi:10.1016/0045-7825(91)90041-4.

[23] Hughes, T. J. R., Franca, L. P., and Hulbert, G. M., "A New Finite-Element Formulation for Computational Fluid Dynamics: VIII. The Galerkin Least Squares Method for Advective-Diffusion Equations," *Computer Methods in Applied Mechanics and Engineering*, Vol. 73, No. 2, 1989, pp. 173–189. doi:10.1016/0045-7825(89)90111-4.

[24] Bazilevs, Y., and Akkerman, I., "Large Eddy Simulation of Turbulent Taylor–Couette Flow Using Isogeometric Analysis and the Residual-Based Variational Multiscale Method," *Journal of Computational Physics*, Vol. 229, No. 9, 2010, pp. 3402–3414. doi:http://dx.doi.org/10.1016/j.jcp.2010.01.008.

[25] Allmaras, S. R., Johnson, F. T., and Spalart, P. R., "Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model," *Seventh International Conference on Computational Fluid Dynamics (ICCFD7)*, 2012.

[26] Balan, A., Park, M. A., Wood, S. L., and Anderson, W. K., "Verification of Anisotropic Mesh Adaptation for Complex Aerospace Applications," AIAA SciTech 2020, American Institute of Aeronautics and Astronautics, Reston, VA (submitted for publication).

[27] Jacobson, K. E., Stanford, B. K., Wood, S. L., and Anderson, W. K., "Flutter Analysis with Stabilized Finite Elements based on the Linearized Frequency-domain Approach," AIAA SciTech 2020, American Institute of Aeronautics and Astronautics, Reston, VA (submitted for publication).

[28] Anderson, W., Newman, J., Whitfield, D., and Nielsen, E., *Sensitivity analysis for the Navier-Stokes equations on unstructured meshes using complex variables*, 1999. doi:10.2514/6.1999-3294, URL `https://arc.aiaa.org/doi/abs/10.2514/6.1999-3294`.

[29] Walker, H. F., "Implementation of the GMRES Method Using Householder Transformations," *SIAM Journal on Scientific Computing*, Vol. 9, No. 1, 1988, pp. 152–163.

[30] Maria, S., Layne T., W., and Rakesh K., K., "A New Adaptive GMRES Algorithm for Achieving High Accuracy," Tech. rep., Blacksburg, VA, USA, 1996.

[31] Helsing, J., "Software," , 2017. URL `http://www.maths.lth.se/na/staff/helsing/matlab.html`.

[32] Greenbaum, A., Pták, V., and Strakoš, Z., "Any Nonincreasing Convergence Curve is Possible for GMRES," *SIAM Journal on Matrix Analysis and Applications*, Vol. 17, No. 3, 1996, pp. 465–469. doi:10.1137/S0895479894275030, URL `https://doi.org/10.1137/S0895479894275030`.

[33] Intel®, "Intel® C++ Compiler 19.0 Developer Guide and Reference," , Sept 2018. URL `https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference`.

[34] "GCC 9.2 Manual," , Nov 2019. URL `https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc/`.

[35] Martins, J. R. R. A., Sturdza, P., and Alonso, J. J., "The Complex-step Derivative Approximation," *ACM Trans. Math. Softw.*, Vol. 29, No. 3, 2003, pp. 245–262. doi:10.1145/838250.838251, URL `http://doi.acm.org/10.1145/838250.838251`.

[36] Martins, J., Sturdza, P., and Alonso, J., *The connection between the complex-step derivative approximation and algorithmic differentiation*, 2001. doi:10.2514/6.2001-921, URL `https://arc.aiaa.org/doi/abs/10.2514/6.2001-921`.

[37] Vasta, V. N., "Computation of Sensitivity Derivatives of Navier-Stokes Equations using Complex Variables," NASA Technical Report 20040086675, 2004.

[38] Conway, J., *On Numbers and Games*, Ak Peters Series, Taylor & Francis, 2000. URL `https://books.google.com/books?id=tXiVo8qA5PQC`.

[39] Galbraith, M. C., Allmaras, S., and Darmofal, D. L., *A Verification Driven Process for Rapid Development of CFD Software*, 2015. doi:10.2514/6.2015-0818, URL `https://arc.aiaa.org/doi/abs/10.2514/6.2015-0818`.

[40] Cuthill, E., and McKee, J., "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proceedings of the 1969 24th National Conference*, ACM, New York, NY, USA, 1969, pp. 157–172. doi:10.1145/800195.805928, URL `http://doi.acm.org/10.1145/800195.805928`.

[41] Anderson, W. K., and Wood, S. L., "Node Numbering for Stabilizing Preconditioners Based on Incomplete LU Decomposition," Aviation 2020, American Institute of Aeronautics and Astronautics, Reston, VA (submitted for publication).

[42] Luby, M., "A simple parallel algorithm for the maximal independent set problem," *SIAM Journal on Computing*, 1986.

[43] Golub, G., and Van Loan, C., *Matrix Computations*, Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 2013.

[44] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, SIAM, Philadelphia, PA, 1994.

[45] Chow, E., and Patel, A., "Fine-Grained Parallel Incomplete LU Factorization," *SIAM Journal on Scientific Computing*, Vol. 37, No. 2, 2015, pp. C169–C193.

[46] "MPI Documents," , Jun 2015. URL `https://www.mpi-forum.org/docs/`.

[47] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., *Introduction to Algorithms, Third Edition*, 3$^{\text{rd}}$ ed., The MIT Press, 2009.

[48] Kepner, J., and Gilbert, J., *Graph Algorithms in the Language of Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011.

[49] "Gram–Schmidt process," , 2017. URL `https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process`.

[50] Björck, Å., "Solving linear least squares problems by Gram-Schmidt orthogonalization," *BIT Numerical Mathematics*, Vol. 7, No. 1, 1967, pp. 1–21. doi:10.1007/BF01934122, URL `https://doi.org/10.1007/BF01934122`.

[51] Kelley, C., *Iterative Methods for Linear and Nonlinear Equations*, Society for Industrial and Applied Mathematics, 1995. doi:10.1137/1.9781611970944, URL `http://epubs.siam.org/doi/abs/10.1137/1.9781611970944`.

[52] "QR DECOMPOSITION USING HOUSEHOLDER TRANSFORMATIONS," , 2017. URL `https://www.keithlantz.net/2012/05/qr-decomposition-using-householder-transformations/`.

[53] Wood, S., Burdyshaw, C. E., Erwin, J. T., Stefanski, D. L., Glasby, R. S., and Peterson, G., *Strategy for Fine-Grained Parallelism in Multi-Level Computational Engineering Solvers*, 2018. doi:10.2514/6.2018-0397, URL `https://arc.aiaa.org/doi/abs/10.2514/6.2018-0397`.

[54] Saad, Y., "A flexible inner-outer preconditioned GMRES algorithm," *SIAM Journal on Scientific Computing*, Vol. 14, No. 2, 1993, pp. 461–469.

[55] O'Connell, M., Druyor, C., Thompson, K. B., Jacobson, K., Anderson, W. K., Nielsen, E. J., Carlson, J.-R., Park, M. A., Jones, W. T., Biedron, R., Lee-Rausch, E. M., and Kleb, B., *Application of the Dependency Inversion Principle to Multidisciplinary Software Development*, ???? doi:10.2514/6.2018-3856, URL `https://arc.aiaa.org/doi/abs/10.2514/6.2018-3856`.

[56] Jones, W. T., Wood, S. L., Jacobson, K. E., and Anderson, W. K., "Interoperable Application Programming Interfaces for Computer Aided Engineering Applications," AIAA Aviation 2020, American Institute of Aeronautics and Astronautics, Reston, VA (submitted for publication).

[57] Lasseter, J., Docter, P., Stanton, A., and Ranft, J., "Toy Story," Walt Disney Pictures presents a Pixar production, 1995. Guggenheim, R. and Arnold, B. (Producers).

[58] Baboulin, M., Buttari, A., Dongarra, J. J., Kurzak, J., Langou, J., Langou, J., Luszczek, P., and Tomov, S., "Accelerating scientific computations with mixed precision algorithms," *ArXiv*, Vol. abs/0808.2794, 2009.

[59] Johansson, F., "Faster arbitrary-precision dot product and matrix multiplication," *CoRR*, Vol. abs/1901.04289, 2019. URL `http://arxiv.org/abs/1901.04289`.

[60] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D., *LAPACK Users' Guide*, 3$^{\text{rd}}$ ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[61] Kurzak, J., Wu, P., Gates, M., Yamazaki, I., Luszczek, P., Ragghianti, G., and Dongarra, J., "Designing SLATE: Software for Linear Algebra Targeting Exascale," SLATE Working Notes 3, ICL-UT-17-06, 2017-10 2017.

[62] "FUN3D Fully Unstructured Navier-Stokes," , Nov 2019. URL `https://fun3d.larc.nasa.gov/`.

[63] "Intel® Xeon® Processor E3-1240 v6," , 2017. URL `https://www.intel.com/content/www/us/en/products/processors/xeon/e3-processors/e3-1240-v6.html`.

[64] "ConnectX®-4 Single/Dual-Port Adapter supporting 100Gb/s with VPI," , 2018. URL `http://www.mellanox.com/page/products_dyn?product_family=201&mtag=connectx_4_vpi_card`.

[65] Rumsey, C., Long, M., Stuever, R., and Wayman, T., *Summary of the First AIAA CFD High Lift Prediction Workshop*, American Institute of Aeronautics and Astronautics, 2011.

[66] Slotnick, J., Hannon, J., and Chaffin, M., *Overview of the 1st AIAA CFD High Lift Prediction Workshop*, American Institute of Aeronautics and Astronautics, 2011.

[67] Anderson, W. K., and Bonhaus, D. L., "Airfoil Design on Unstructured Grids for Turbulent Flows," *AIAA Journal*, Vol. 37, No. 2, 1999, pp. 185–191. doi:10.2514/2.712, URL `https://doi.org/10.2514/2.712`.

[68] "VERIF/3DB: 3D Bump-in-channel Verification Case," , Nov 2019. URL `https://turbmodels.larc.nasa.gov/bump3d.html`.

[69] "3D ONERA M6 Wing Validation Case," , Nov 2019. URL `https://turbmodels.larc.nasa.gov/bump3d.html`.

[70] "Test Cases," , Nov 2019. URL `https://hiliftpw.larc.nasa.gov/Workshop3/testcases.html`.