Interoperable Application Programming Interfaces for Computer Aided Engineering Applications

William T. Jones,* Stephen L. Wood,† Kevin E. Jacobson,‡ W. Kyle Anderson,§
NASA Langley Research Center, Hampton, VA 23681-2199

A suite of software interfaces has been developed to facilitate the coupling of disciplines of interest to Computer Aided Engineering. The definition of the interfaces focused on abstraction and interoperability between components of coupled disciplines. The interfaces are language independent and allow for instantiable state to be maintained by the respective implementation. However, to support the largest spectrum of programming models, C calling convention has been adopted. The use of software interfaces places the dependency on the interface and not on the specifics of its implementation. Therefore, different implementations of a given interface can be seamlessly adopted as needed. Implementation of the interfaces remains the responsibility of component developers who have total control over the details of fulfillment. However, strict adherence to the interface definition guarantees interoperability between components. The interface definition is successfully demonstrated through application to Computational Fluid Dynamics.

I. Introduction

Common development of Computer Aided Engineering (CAE) software integrates components from a number of different disciplines. This is especially true of modeling and simulation applications that typically tie together numerical representations and algorithms to produce their results.

For example, a Fluid-Structure Interaction (FSI) prediction would, at a minimum, couple the loads produced by a Computational Fluid Dynamics (CFD) simulation with the displacements obtained from a Computational Structural Mechanics (CSM) analysis. Within the computational methods, different iterative solvers may be used for varying levels of accuracy, physics, or efficiency. Furthermore, the computational methods would likely leverage a discretization based upon a numerical mesh. The numerical mesh can be considered a separate *discipline* in itself and might differ for each computational method. In a High Performance Computing (HPC) environment, the meshes may be partitioned to suit the parallel execution of the methods on each coupled partition. An HPC environment would also require some method of communication between processes to facilitate domain coupling. And given that the meshes may be different for each computational method, a data transfer scheme would be required. Other disciplines could be included to address overset domain simulation, multibody dynamics, optimization, etc..

As described, the interaction of a number of disciplines is required for such a simulation. Typically, this interaction would be hardwired for specific implementations of the required capabilities (e.g., a given CFD and CSM tool). Further still, each discipline would implement a specific set of solvers, mesh partitioners, communication schemes, etc. representing additional requirements of each discipline. The result is a very rigid system that is difficult to extend to support new and emerging implementations of required capabilities. And, unfortunately, there is a strong tendency to tie together disciplines with hidden, intertwining dependencies that inhibit extension to accommodate new methodologies.

A more manageable design would employ a well-defined Application Programming Interface (API) to control the interaction between components. Here, access to each capability is provided through an abstract software interface. As such, the *consumer* of the capability (be it a workflow framework or another component)

*Computer Engineer, Computational AeroSciences Branch, MS 128, AIAA Associate Fellow, w.t.jones@nasa.gov
†Research Scientist, Computational AeroSciences Branch, MS 128, AIAA Member
‡Research Aerospace Engineer, Aeroelasticity Branch, MS 340, AIAA Member
§Senior Research Scientist, Computational AeroSciences Branch, MS 128, AIAA Associate Fellow

interacts through the same interface regardless of how the capability is implemented. The consumer depends on the interface, not on the specifics of the underlying implementation of said interface. The interfaces are now what is rigid. Impact to consumers is limited through the adoption of the open-closed principle[1] in that the interfaces are open to extension, but closed for modification. Therefore, when a new implementation becomes available for a given capability, no change is required of the consumer to incorporate it as the interface remains unchanged. Likewise, when modifications are made to the underlying implementation of a capability, again the consumer is unaffected as the interface is constant.

In this work, value is placed on the definition of concise interfaces rather than the framework that will leverage the interfaces to accomplish a task. A large number of frameworks of varying complexity have developed into and out of existence but often fail to gain widespread acceptance. This is due in part to the nature of framework development. Framework development often reflects the organizational structure, policies, and work-flow of the framework developers. Only when those organizational influences can be adopted or forced upon others will the framework gain acceptance. Hence, we see a proliferation of frameworks as everyone is eventually forced to create their own. Historically, a large number of high profile, *universal* frameworks are relegated to "whale fall",[2,3] a collection of carcasses accumulating in the depths of the abyss.

That said, with an ever increasing number of frameworks, component developers face the never-ending challenge of supporting multiple frameworks. Each framework imposes its own requirements on the component for its inclusion. Calling mechanisms and data layouts are rarely the same. Multiple *adaptors* must be written and maintained to communicate data and actions between the framework and the component. This represents undue burden that is largely out of the scope of the underlying component development. As a result, there is reduced "buy-in" incentive on the part of the component developer, who is likely less interested in working with frameworks than developing their respective component.

Instead, the focus here is on the definition of the abstract capabilities of discipline components of interest to computational simulation. The goal is to define the interactions needed for a given component capability and leave the specific process by which they are executed up to the discretion of the consumer. A component that conforms to the interfaces has no dependencies on particular implementations of other components, including the "driver" or framework. Such a conforming component lacks code-to-code interactions with other components and is therefore self-contained and portable. This work carries on from the research and work presented by O'Connell,[4] which focused upon the software design principles that guide development of decoupled components and initiated the definition of component interfaces. This work refines the definition of component interfaces by distinguishing the common functionality of a component from the way a component is created. This work also adopts the C calling convention for component interfaces to leverage the established interoperability among programming languages. The hope is that an implementation that conforms to the interface can be used by any number of consumers (work-flow frameworks, other dependent components, etc.), thus furthering usability and increasing "buy-in" from component developers.

II. Interface Design

The design of our interfaces is independent of the programming language intended for implementation or use. The goal is to establish a *contract* for an abstract capability of interest. Included in the abstraction is the removal of any assumption of a concrete specification of language. This provides the freedom of language choice to both the interface implementer and to the consumer (i.e., caller) and increases the acceptance and use of the interface. The abstraction carries over to the specification of data types used by the interface. Arguments and return values are limited to scalars and arrays of intrinsic types to eliminate imposition of a specific data structure upon either the implementer or the consumer.

It is desirable that the interface design support thread safety and multiple instantiations. Therefore, the interfaces will communicate an opaque pointer representing implementation state. This pointer will be created by the implementation and provided to the consumer. The pointer is opaque with respect to the consumer and therefore cannot be interpreted or interrogated. Instead, the consumer must supply the pointer to subsequent interface calls of the same implementation, who can then interpret it as its state. This allows the interface implementation to store its state in any form it desires (object pointer, internal data structure, intrinsic type, or NULL) without impact on the consumer.

Although the interface definition is independent of programming language, the implementations of the interfaces will expose themselves using a standard C calling convention. This decision was made to enable the broadest interaction between computer programming languages. Practically every major programming

language knows how to issue calls with the C calling convention. Some examples include: C++ via `extern "C"`; Fortran 2003 and above via ISO C bindings; Python via ctypes, Cython, or SWIG; and Java via JNI. This does not require that the interface be implemented in C, but only that the interface adhere to the C calling convention. An example of the C calling convention is given in Listings 1 and 2 for C/C++ and Fortran, respectively. To reiterate, while we employ the C calling convention for language interoperability, an implementer or consumer is free to program in their language of choice. Listing 2 is an example of how one would implement or use the interface from Fortran. Note that, regardless of the implementing language, the function signatures are identical.

Listing 1. Interface Example in C/C++.

```
extern "C" int64_t tinf_mesh_node_count(void* mesh, int32_t* err);
```

Description

Retrieve the total number of nodes in the mesh partition (resident and neighboring/ghost nodes).

Parameters

mesh	Opaque mesh pointer
error	Error code

Return

Total number of nodes in the partition

Listing 2. Interface Example in Fortran.

```
interface
  function tinf_mesh_node_count(mesh, error) result(cnt) bind(c)
    use, intrinsic :: iso_c_binding, only : c_ptr, c_int64_t, c_int32_t
    implicit none
    type(c_ptr),          value,          intent(in)    :: mesh
    integer(c_int32_t),                   intent(out)   :: error
    integer(c_int64_t)                                  :: cnt
  end function tinf_mesh_node_count
end interface
```

Description

Retrieve the total number of nodes in the mesh partition (resident and neighboring/ghost nodes).

Parameters

mesh	Opaque mesh pointer
error	Error code

Return

Total number of nodes in the partition

Finally, we depend on the abstraction of a capability to define commonality among various methodologies. However, the initial start-up of each methodology may be different. For example, a mesh may have one implementation based on a disk file and another leveraging an in-core byte stream. Once created, both of these mesh *objects* will provide the same actions: return number of nodes, return node coordinates, return element connectivity, etc. Therefore, we separate the instantiation of a capability from the common actions. We say that all meshing capabilities *implement* the meshing interface. A file-based mesh will instantiate itself via a "preprocessor" interface while an in-core mesh will instantiate via a "stream" interface. The "preprocessor" interface might create from a character array containing the file name, while the "stream" interface creates via

a data pointer. Both of these, however, must *implement* the meshing interface to provide the common actions once created.

III. Interface Definition

While the overall goal of this work is to define and promote the use of an API for CAE, the initial thrust has been focused on CFD. As of this writing, interfaces have been defined for: parallel communication, problem definition, mesh access, linear solver execution, nonlinear iterative solver execution, and solution access. The following subsections contain select descriptions of specific interfaces. The complete documentation for the interfaces defined to date is included in the Appendix.

A. Parallel Communication

A parallel communications interface, known as Iris, is fully described in Appendix B. It is currently implemented using the Message Passing Interface (MPI) with CUDA integration in what is commonly referred to as an MPI+X arrangement. However, Iris provides the general functionality for data migration and collection in an HPC environment and can be implemented with other protocols as needed. At a minimum, Iris hides the "+X" constituent behind the abstracted interface. Iris provides both blocking and nonblocking communication for parallel computing architectures. While the Iris interface closely mimics the MPI standard, the implementation can include other technologies, such as the aforementioned CUDA support, without impacting derivative works.

Iris, like the other interfaces in this collection, is not required for use. A CAE component developer is free to use their own communication scheme. However, that decision will restrict them to the selected communication method and impede seamless integration of other communication schemes in the future. For example, a legacy component may already be implemented with hard-coded references to MPI. It is not required to convert all communication to use the Iris interface. However, failure to do so will restrict that component to MPI and render it unusable in a framework not implemented with MPI. If instead, Iris is used, a change in the Iris implementation requires no changes to the dependent component. Integration into a framework using an alternative communications protocol poses no impact on the component developer. Therefore, use of Iris is highly encouraged.

B. Problem Definition

The definition of the problem to solve can be an arduous task. When supporting multiple disciplines, we want to avoid the proliferation of input files and formats scattered about the system. More importantly, our goal is to facilitate the extension of capabilities by allowing the seamless integration of new components. As new components are introduced, access to their input data must remain unchanged. Interaction with the evolving problem description must be concise yet flexible enough to accommodate new components as they are introduced. Such is the goal of the problem definition abstraction.

The method of abstraction for the interaction with the problem definition consists of two interface functions shown in Listing 3 and repeated in Appendix C. The first function returns a Boolean value of "true" if the definition contains a particular parameter, named by a given character array. If the problem does contain a definition for the specified parameter, the function also returns the type, matrix rank, and dimension of the parameter data. A second function is used to retrieve the data value(s) of the named parameter. The interface defines a void type that is expected to be a pointer to data of the appropriate type, rank, and dimension indicated by the first function.

Unless specifically stated in the problem value key, values are assumed to be dimensional using the International System of Units (SI).

Listing 3. Problem Interface.

```
_BOOL_ tinf_problem_defined(void* problem, const char* key, size_t keylen,
                            enum TINF_DATA_TYPE* datatype, int32_t* rank,
                            size_t dims[TINF_PROBLEM_MAX_RANK], int32_t* err)
```

Parameters

problem	Opaque problem pointer
key	Parameter name
keylen	Length of the parameter name
datatype	TINF_DATA_TYPE enumeration of the defined parameter
rank	Rank of the defined parameter
dims	Dimensions of each rank of the defined parameter
err	Error code

Return

True if *key* is defined in the problem definition

```
void tinf_problem_value(void* problem, const char* key, size_t keylen, const void* data,
                        int32_t rank, const size_t dims[], int32_t* err)
```

Parameters

problem	Opaque problem pointer
key	Parameter name
keylen	Length of the parameter name
data	Pointer to storage for the requested data
rank	Rank of the parameter
dims	Dimensions of each rank of the parameter
err	Error code

The definition of the problem to be solved can be quite complex. For example, the definition may be provided by a disk file, driven interactively by a Graphical User Interface (GUI), obtained from an input stream, etc. Furthermore, the format of the definition might differ from application to application. Formats such as XML,[5] JSON,[6] and Fortran namelists are but a few in popular use today. The issue of format can be hidden beneath the interface of Listing 3. However, the requirements for the creation of a problem definition might still be quite different (e.g., file name, byte stream). Therefore, the creation and destruction of the problem definition is defined by distinct interfaces that will additionally *implement* the common problem interface. That way, any problem *object* can be interrogated with the problem interface of Listing 3 regardless of its method of creation, provided that they leverage the same implementation.

For demonstration purposes, the interface defined to load a problem description from a file is given in Listing 4. Other forms of the create function can be defined to return a problem pointer based on other means of problem definition.

Listing 4. File-Based Problem.

```
void tinf_input_file_create(void** problem, const char* filename, size_t namelen,
                            void* comm, int32_t* err)
```

Parameters

problem	Pointer to opaque problem pointer
filename	File name
namelen	Length of the filename name
comm	Iris communications pointer
err	Error code

C. Mesh

The full mesh interface is provided in Appendix E, however, an example of the interface is given in Listing 5. Included here are methods to retrieve: the total number of nodes in the partition, number of elements of a given type in the partition, node coordinates, and element connectivity. The number of nodes in the partition includes those resident in the partition as well as neighboring halo nodes. Other interface methods can determine information about node residency. Element types are given in a defined enumeration. Reference indices for nodes are bias-0 to be consistent with the C calling convention.

Listing 5. Mesh Interface Examples.

```
int64_t tinf_mesh_node_count(void* mesh, int32_t* err)
```

Parameters

mesh	Opaque mesh pointer
err	Error code

Return

Total number of mesh nodes on the partition

```
int64_t tinf_mesh_element_type_count(void* mesh, enum TINF_ELEMENT_TYPE type,
                                     int32_t* err)
```

Parameters

mesh	Opaque mesh pointer
type	Element type enumeration
err	Error code

Return

Total number of elements of *type* on the partition

```
int32_t tinf_mesh_nodes_coordinates(void* mesh, int64_t start, int64_t cnt,
                                    double* x, double* y, double* z)
```

Parameters

mesh	Opaque mesh pointer
start	Starting node index to retrieve outputs
cnt	Number of nodes at which outputs are requested
x	X-coordinate of node
y	Y-coordinate of node
z	Z-coordinate of node

Return

Error code

```
int32_t tinf_mesh_element_nodes(void* mesh, int64_t element_id, int64_t* element_nodes)
```

Parameters

mesh	Opaque mesh pointer
element_id	Bias-0 index of an element
element_nodes	Bias-0 cell-to-node connectivity

Return

Error code

While the mesh interface defines methods for retrieval of data from a partitioned mesh, the construction and destruction of the mesh object is separated from the retrieval interface to support multiple methods of construction (file-based, data stream, etc.). Every implementation, however, will implement the mesh interface, regardless of the interface used for construction and destruction. Listing 6 presents the interface for construction of a file-based mesh object that obtains information about the file (name, format, etc.) from the problem description. Listing 7 presents an alternative interface for the construction of a mesh given the file specifics as arguments. In either case, the resulting opaque mesh pointer can be subsequently provided to the respective mesh interface implementation for interrogation of the partitioned mesh.

Listing 6. File-based Mesh.

```
int32_t tinf_preprocessor_create(void** mesh, void* problem, void* comm)
```

Parameters

mesh	Opaque mesh pointer
problem	Problem definition pointer (containing the file name)
comm	Iris communications pointer

Return

Error code

Listing 7. Alternate File-based Mesh.

```
int32_t tinf_mesh_file_create(void** mesh,
                              const char* name,
                              size_t nlen,
                              const char* style,
                              size_t slen,
                              const char* form,
                              size_t flen,
                              void* problem, void* comm)
```

Parameters

mesh	Opaque mesh pointer
name	The name of the file containing the mesh
nlen	Length of *name*
style	The style of the data layout in the file
slen	Length of *style*
form	The data format (i.e., 'binary')
flen	Length of *form*
problem	A problem description object
comm	A communications object

Return

Error code

D. Linear Solver

A programming interface has been developed to abstract the solution of sparse linear systems of equations. The interface allows for multiple solution methods within a single implementation: Conjugate Gradient Method, Generalized Minimum Residual Method (GMRES), Flexible GMRES (FGMRES), etc. The interface has been successfully implemented by both SPARSKIT[7] and the Sparse Linear Algebra Toolkit for Computational Aerodynamics (SLAT).[8] Both implementations have been successfully exercised by a number of internal components demonstrating seamless exchange of each without altering the internal components. The interface includes a method of creation and destruction of the linear solver object. These are accompanied by a single method to solve the system using the reverse-communication interface, RCI.[9]

The RCI avoids having to express a matrix-vector product through a subroutine with a defined calling sequence. As such, the caller is free to choose the representation of the matrix. No fixed data structure is imposed. In fact, the RCI allows the linear solver to function even if the matrix is not available explicitly. The user is free to express the action of the matrix on a vector by the best means that meets their needs.

The three linear solver interface functions are defined in Appendix G.

E. Solution

While the methods of the execution and the creation/destruction of state may vary across different solvers, access to the system solution vector is uniform. Therefore, we separate the interfaces for solution access from those defined to interact with the solver. Solvers are required to *implement* the solution interface.

A method of access similar to that of the Problem Definition in Subsection B above is defined. The ability to set input values and retrieve outputs both follow a similar procedure and two interfaces are defined for each. The first function is used to determine a list of inputs/outputs that the solution expects/provides. The list is comprised of character arrays that communicate the names of the variables to be exchanged. The second function is given a subset of this list representing the variable names to be set/retrieved as well as the values/storage for the data. As with the list of variables to set/retrieve, data can be set/retrieved at a subset of nodes in the computational mesh. Listing 8 represents the interface to exchange output variables.

Solution input and output differ with respect to who is in control when invoking the interface. Interface invocation is controlled by the consumer. A problem exists when the implementation requires multiple input

elements be received before processing can begin (i.e., an entire state vector is required for nondimensionalization). But the implementation is not in control of how it might be called by the consumer. The consumer may set multiple elements by a single invocation or by multiple invocations of the input interface. The implementation needs to know when the input is complete such that it can begin processing the input elements. Therefore, the input interface includes the data matrix rank and dimensions for this purpose. If an entire state is required for processing a particular variable, the input is communicated per invocation in the form of a vector (matrix, etc.). However, as the output interface is also controlled by the consumer, it can determine when enough data has been retrieved for processing to begin. As such simple scalar data retrieval is sufficient for output processing.

Similar to the problem interface, unless specifically stated in the solution input/output name, values are assumed to be dimensional using SI units.

Listing 8. Solution Interface Examples.

```
void tinf_solution_get_nodal_output_names(void* solution, int64_t *n_outputs,
                                          const char** names[], int32_t* err)
```

Parameters

solution	Opaque solution pointer
n_outputs	Number of outputs available
names	List of character arrays containing output names
err	Error code

```
void tinf_solution_get_outputs_at_nodes(void* solution, int64_t start,
                                        int64_t node_count, int64_t value_count,
                                        const char* names[], double* values,
                                        int32_t* err)
```

Parameters

solution	Opaque solution pointer
start	Starting node index to retrieve outputs
node_count	Number of nodes at which outputs are requested
value_count	Number of outputs requested
names	List of character arrays containing output names
values	Vector of output values at nodes (*value_count* stride)
err	Error code

F. Nonlinear Solver

The iterative solution of nonlinear systems has been abstracted to include the creation of the solver and its destruction. Again, the complete interface is included in Appendix H, however, the following functionality is documented here to relate the primary work-flow. The iterative solution is broken into three phases. The principal action is the advancement of the time-step, however, additional granularity is provided to execute operations before and after the time advancement. These steps allow a solver to perform any additional operations prior, or subsequent, to the time-step (e.g., coupling such as mesh update or force calculation). While the interface defines these two additional operations, an implementer may simply provide empty placeholders if the added granularity is not required.

Listing 9. Nonlinear Solver Interface Examples.

```
int32_t tinf_solver_step_pre(void* solver)
```

Parameters

solver	Opaque nonlinear solver pointer

Return

Error code

```
int32_t tinf_solver_step(void* solver, double seconds_to_advance)
```

Parameters

solver	Opaque nonlinear solver pointer
seconds_to_advance	Seconds to advance the solution for a time dependent system

Return

Error code

```
void tinf_solver_step_post(void* solver)
```

Parameters

solver	Opaque nonlinear solver pointer

Return

Error code

IV. Results and Discussion

The described interfaces have been implemented and tested to gauge interoperability and, frankly, as part of the interface design process. In addition, multiple implementations have been developed for the problem definition, mesh, linear solver, nonlinear solver, and solution interfaces using a collection of languages including standard C, C/C++, and Fortran.

An implementation of the problem, mesh, solution, and solver interfaces has also been created using the internal data structures and methodologies of the NASA FUN3D CFD solver,[10] written primarily in Fortran. This implementation is being used to leverage the wealth of capability already existing in FUN3D for testing new components. For example, the problem and mesh interfaces are implemented by FUN3D using its own internal input methods and partitioner. However, this implementation is currently limited to static linking of components.

To facilitate static linking, an simple implementation of the interfaces is provided by FUN3D, which switches among the actual predefined implementations of the components based on user input. Symbol clashes are avoided by mangling the symbol names of the actual implementation consistent with the method defined by the libltdl package of GNU libtool.[11] In summary, symbols are prefixed with the module name followed by "_LTX_". This prefix is automatically trimmed away by libltdl and can be linked/loaded directly by other methods. Note that this is only necessary for static linking of multiple component implementations simultaneously and is not required for interface implementation. In fact, the symbols of existing, binary components can be mangled for static linking (e.g., objcopy[12]) as needed.

Currently, there are two external CFD flow solver implementations statically linked to FUN3D. The Stabilized Finite Element (SFE) solver,[13] is written in C++. To complete the separation of SFE from FUN3D, the problem and mesh interfaces provided by FUN3D are used by SFE to obtain data for its operation. SFE implements the solver and solution interfaces and FUN3D interacts with SFE indirectly through them to execute the solver and retrieve its results. An additional node-centered, finite-volume (NCFV) solver has also been

incorporated into FUN3D via the solver and solution interfaces and it too utilizes the problem and mesh interfaces for its input. As mentioned above, a simple implementation of the solver and solution interfaces switches between SFE and NCFV based on a single user input flag. The flag is used in a conditional that subsequently invokes the name mangled symbol from the respective library. All other FUN3D code remains unaffected. Addition of a new solver in this paradigm simply requires additional logic in the switching conditional of the simple FUN3D implementation.

A mesh of the High Lift version of the Common Research Model (CRM-HL) is used to generate a simulation with FUN3D utilizing both SFE and NCFV. The results are presented in Figure 1 and Figure 2, respectively. The conditions for each simulation are identical and both employ the Spalart-Allmaras[14] turbulence model with negative turbulence variable provisions[15] (i.e., SA-NEG). SFE employs a Courant–Friedrichs–Lewy (CFL) controller as is evident by the accelerated rate of convergence. However, this, along with the Finite Element solution scheme, requires more computational time per iteration such that the cost is comparable to NCFV. Note also that for SFE, the final residual values for the turbulence model are below the values for the other variables; this is due to internal scaling withing SFE.[13]

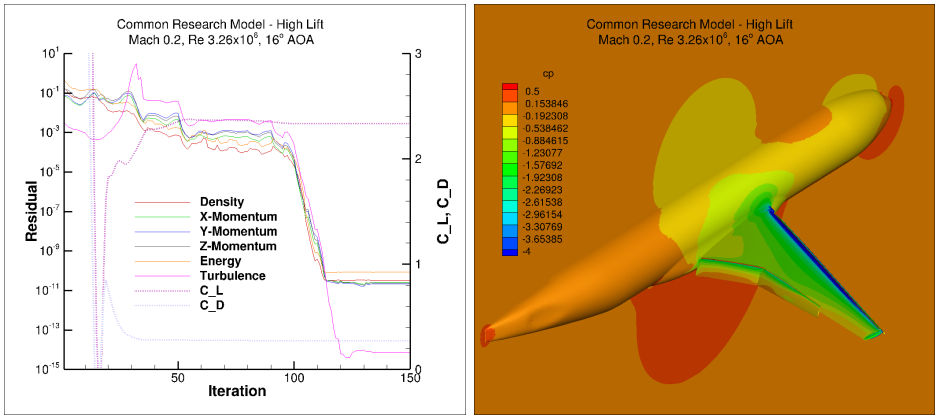

Figure 1. SFE flow solver component.

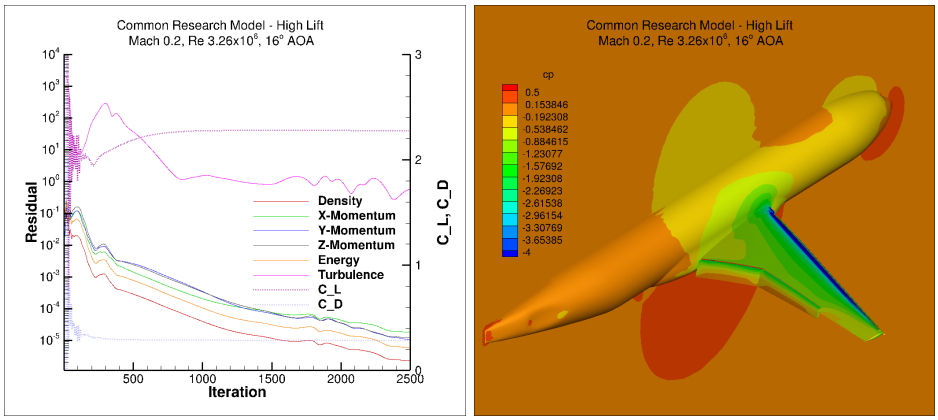

Figure 2. NCFV flow solver component.

An experimental framework is also under development that uses dynamically loadable modules (i.e., plug-ins) to load implementations. This technique provides great flexibility for configuration and execution and avoids the types of symbol collisions that accompany a statically linked program. With this framework, components can be loaded at run-time, eliminating the need to modify any source code in order to support a new component implementation. As a result, there is great potential to reduce overall maintenance and better support future extensions and capabilities. The framework, named Maestro, leverages the libltdl capabilities of libtool, and in turn Linux dlopen(3), to dynamically load symbols from a Dynamic Shared Object (DSO) representing a component. Multiple components can be loaded simultaneously with interaction and execution orchestrated by the framework.

Using Maestro, a new parallel-partitioning scheme, internally referred to as the New Front End (NFE), has been used to implement a standalone mesh interface component. Also for Maestro, a standalone FUN3D

namelist reader was developed to implement the problem definition interface. Maestro has coupled this mesh interface with the SFE solver component and the standalone problem description component to execute simulations outside of the static FUN3D environment described above. Exchange for the NCFV flow solver required no software modification of either the framework or the component implementation, as it is handled completely by loading the new NCFV DSO to replace SFE.

Results are similar to Figures 1 and 2 with minor differences in the convergence history attributed to the slightly altered partitioning provided by the NFE.

V. Conclusion

A suite of software interfaces has been presented to facilitate the coupling of disciplines of interest to Computer Aided Engineering. The interfaces presented represent an abstraction of operations required by distinct components related to CFD simulation. These included: parallel communication; problem definition; mesh partitioning; linear and nonlinear solver execution; and solution access. The interfaces are not tied to any one implementation and adhere to a standard C calling convention to provide compatibility with modern programming languages. The implementation of a given component can be written using the language of choice of the developer, provided that the interface itself is exposed via the C calling convention.

Multiple implementations of the interfaces have been developed as standalone components. As a test-bed, implementations using FUN3D internal data structures have also been developed. A stand alone implementation of the parallel communications interface based on MPI+X was used in this work. The problem definition interface was implemented with both the internal FUN3D data structures and separately as a standalone reader of the FUN3D namelist. A standalone mesh component based on the FUN3D New Front End partitioning paradigm was developed to implement the meshing interface as an alternative to the implementation based on FUN3D internal data structures. The linear solver interface was implemented with SPARSKIT and the Sparse Linear Algebra Toolkit. The Stabilized Finite Element scheme and a Node Centered Finite Volume method were used as the basis for the flow solver interface whereupon both implementations in turn implemented the solution interface.

The implementations were both statically linked to FUN3D and used to build Dynamic Shared Objects for use as software plug-ins to the developing Maestro framework. Maestro uses dynamic linking to eliminate code modification when extending support to new component implementations. CFD simulations were presented for the High Lift version of the Common Research Model using both the Stabilized Finite Element and Node Centered Finite Volume methods. The use of standard interfaces greatly simplified the support of both solution schemes.

Acknowledgments

The authors would like to acknowledge the FUN3D Development Team for the continued support and development of FUN3D as well as lengthy discussions in the development of the software interfaces. Individual acknowledgement is given to Dr. Li Wang of the team for her help in the development of NCFV. This research is sponsored by the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program (TACP) under the Aeronautics Research Mission Directorate.

References

[1]Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1988, ISBN: 0-13-629049-3.

[2]Alonso, J. J., Personal Communication, Boeing, St. Louis, MO, Oct 2018, Boeing/NASA/Stanford Techincal Interchange.

[3]Wikipedia, "Whale fall — Wikipedia, The Free Encyclopedia," http://en.wikipedia.org/w/index.php?title=Whale%20fall&oldid=921682475, 2019, [Online; accessed 25-Nov-2020].

[4]O'Connell, M. D., Druyor, C. T., Thompson, K. B., Jacobson, K. E., Anderson, W. K., Nielsen, E. J., Carlson, J. R., Park, M. A., Jones, W. T., Biedron, R. T., Kleb, B., and Zhang, X., *Application of the Dependency Inversion Principle to Multidisciplinary Software Development*, Atlanta, GA, AIAA 2018-3856.

[5]Sperberg-McQueen, M., Bray, T., Maler, E., Paoli, J., and Yergeau, F., "Extensible Markup Language (XML) 1.0 (Fifth Edition)," http://www.w3.org/TR/2008/REC-xml-20081126, Nov 2008, [Online; accessed 25-Nov-2020].

[6]ECMA, "ECMA-404: The JSON Data Interchange Format," http://www.ecma-international.org/publications/standards/Ecma-404.htm, 2013, [Online; accessed 25-Nov-2020].

[7]Saad, Y., "SPARSKIT: a basic tool kit for sparse matrix computations - Version 2," 1994.

[8]Wood, S. L., Jacobson, K. E., Jones, W. T., and Anderson, W. K., *Sparse Linear Algebra Toolkit for Computational*

Aerodynamics, AIAA, Orlando, FL, AIAA 2020-0317, Jan 2020.

[9]Dongarra, J., Eijkhout, V., and Kalhan, A., "Reverse Communication Interface for Linear Algebra Templates for Iterative Methods," UT-CS-95-291, University of Tennessee, May 1995.

[10]Biedron, R. T., Carlson, J. R., Derlaga, J. M., Gnoffo, P. A., Hammond, D. P., Jones, W. T., Kleb, B., Lee-Rausch, E. M., Nielsen, E. J., Park, M. A., Rumsey, C. L., Thomas, J. L., Thompson, K. B., and Wood, W. A., "FUN3D Manual: 13.5," NASA TM-2019-220271, NASA, Langley Research Center, Hampton, VA, Apr 2019.

[11]"GNU Libtool," https://www.gnu.org/software/libtool/manual/libtool.html#Modules-for-libltdl, 2015, [Online; accessed 12-Nov-2020].

[12]"GNU Binary Utilities," https://sourceware.org/binutils/docs/binutils/objcopy.html, 2020, [Online; accessed 12-Nov-2020].

[13]Anderson, W. K., Newman, J. C., and Karman, S. L., "Stabilized Finite Elements in FUN3D," *Journal of Aircraft*, Vol. 55, No. 2, Oct 2017, pp. 696–714.

[14]Spalart, P. R. and Allmaras, S. R., "A One-Equation Turbulence Model for Aerodynamic Flows," AIAA 92-0439, Jan 1991.

[15]Allmaras, S. R., Johnson, F. T., and Spalart, P. R., "Modifications and Clarifications for the Implementation of the Spalart-Allmaras Turbulence Model," *7th International Conference on Computational Fluid Dynamics (ICCFD7)*, Paper ICCFD7-1902, 2012.

Appendix

A. Enumeration Types

```
enum TINF_MAX_RANKS {
    TINF_PROBLEM_MAX_RANK = 5,
    TINF_DATA_MAX_RANK = 2
}
```

Description

Max array ranks.

```
enum TINF_ERROR_CODES {
    TINF_SUCCESS = 1,
    TINF_FAILURE = 2
}
```

Description

Function return codes.

```
enum TINF_ELEMENT_TYPE {  /*--- Following the CGNS element winding convention */
    TINF_NODE = 2,
    TINF_BAR_2 = 3,
    TINF_BAR_3 = 4,
    TINF_BAR_4 = 24,
    TINF_BAR_5 = 40,
    TINF_TRI_3 = 5,
    TINF_TRI_6 = 6,
    TINF_TRI_9 = 25,
    TINF_TRI_10 = 26,
    TINF_TRI_12 = 41,
    TINF_TRI_15 = 42,
    TINF_QUAD_4 = 7,
    TINF_QUAD_8 = 8,
    TINF_QUAD_9 = 9,
    TINF_QUAD_12 = 27,
    TINF_QUAD_16 = 28,
    TINF_QUAD_P4_16 = 43,
    TINF_QUAD_25 = 44,
    TINF_TETRA_4 = 10,
    TINF_TETRA_10 = 11,
    TINF_TETRA_16 = 29,
    TINF_TETRA_20 = 30,
    TINF_TETRA_22 = 45,
    TINF_TETRA_34 = 46,
    TINF_TETRA_35 = 47,
    TINF_PYRA_5 = 12,
    TINF_PYRA_14 = 13,
    TINF_PYRA_13 = 21,
    TINF_PYRA_21 = 31,
    TINF_PYRA_29 = 32,
    TINF_PYRA_30 = 33,
    TINF_PYRA_P4_29 = 48,
    TINF_PYRA_50 = 49,
    TINF_PYRA_55 = 50,
    TINF_PENTA_6 = 14,
    TINF_PENTA_15 = 15,
    TINF_PENTA_18 = 16,
    TINF_PENTA_24 = 34,
    TINF_PENTA_38 = 35,
    TINF_PENTA_40 = 36,
    TINF_PENTA_33 = 51,
    TINF_PENTA_66 = 52,
```

```
    TINF_PENTA_75 = 53,
    TINF_HEXA_8 = 17,
    TINF_HEXA_20 = 18,
    TINF_HEXA_27 = 19,
    TINF_HEXA_32 = 37,
    TINF_HEXA_56 = 38,
    TINF_HEXA_64 = 39,
    TINF_HEXA_44 = 54,
    TINF_HEXA_98 = 55,
    TINF_HEXA_125 = 56
}
```

Description

Element type definitions. These follow the CGNS element winding convention.

```
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
}
```

Description

Data types

```
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
}
```

Description

Matrix storage types

B. Iris Parallel Communications Interface

```
int32_t tinf_iris_create(void** comm, int32_t communicatorr)
```

Description

Create an Iris object based on an MPI communicator.

Parameters

comm	Opaque Iris pointer
communicator	MPI Fortran communicator

Return

Error code

```
int32_t tinf_iris_get_mpi_fcomm(const void* comm, int32_t* err)
```

Description

Return the Fortran MPI communicator from an opaque Iris pointer.

Parameters

comm	Opaque Iris pointer
err	Error code

Return

MPI Fortran communicator

```
int32_t tinf_iris_fcomm_free(const void* comm)
```

Description

Release MPI Fortran communicator associated to an opaque Iris pointer.

Parameters

comm	Opaque Iris pointer

Return

Error code

```
int32_t tinf_iris_rank(const void* comm, int32_t* err)
```

Description

Retrieve the rank of the calling process in an opaque Iris pointer.

Parameters

comm	Opaque Iris pointer
err	Error code

Return

Processor rank (bias-0)

```
int32_t tinf_iris_number_of_processes(const void* comm, int32_t* err)
```

Description

Retrieve the size of the group associated with an opaque Iris pointer.

Parameters

comm	Opaque Iris pointer
err	Error code

Return

Total nuber of processors

```
int32_t tinf_iris_stop(const void* comm, int32_t flag, const char* msg,
                       size_t len)
```

Description

Stop the processes associated with an opaque Iris pointer based on a nonzero reduction of processor flags. If any process triggers a stop, then all processes will be stopped.

Parameters

comm	Opaque Iris pointer
flag	Stop flag (nonzero indicates stop requested by process)
msg	Common message to indicate what has triggered the stop
len	Length of *msg* character array

Return

Error code

```
int32_t tinf_iris_abort(const void* comm)
```

Description

Terminate execution environment.

Parameters

comm	Opaque Iris pointer

Return

Error code

```
int32_t tinf_iris_destroy(const void* comm)
```

Description

Destroy the Opaque Iris pointer and release its state.

Parameters

comm	Opaque Iris pointer

Return

Error code

```
int32_t tinf_iris_barrier(const void* comm)
```

Description

Create a synchronization barrier for the processes.

Parameters

comm	Opaque Iris pointer

Return

Error code

```
int32_t tinf_iris_waitall(const void* comm, int32_t counts, size_t nreq,
                          int32_t array_of_requests)
```

Description

Waits for all given process requests to complete.

Parameters

comm	Opaque Iris pointer
counts	List length
nreq	Length of *array_of_requests*
array_of_requests	Array of requests

Return

Error code

```
size_t tinf_iris_max_processor_name_length(void* comm, int32_t* err)
```

Description

Retrieve the number of characters in the maximum processor name. Used to allocate processor name character array for tinf_iris_get_processor_name.

Parameters

comm	Opaque Iris pointer
err	Error code

Return

Length of maximum processor name

```
int32_t tinf_iris_get_processor_name(void* comm, const char* name, size_t nlen)
```

Description

Retrieve the processor name.

Parameters

comm	Opaque Iris pointer
name	Processor name
nlen	Number of characters in the processor name

Return

Error code

```
int32_t tinf_iris_broadcast(const void* comm, const enum TINF_DATA_TYPE data_type,
                            const int32_t data_rank,
                            const size_t data_dims[TINF_DATA_MAX_RANK],
                            const void* buf, int32_t root)
```

Description

Broadcasts a message from process rank *root* to all other processes of the group.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
root	Source rank of data

Return

Error code

```
int32_t tinf_iris_sum(const void* comm, const enum TINF_DATA_TYPE data_type,
                      const int32_t data_rank, const size_t data_dims[TINF_DATA_MAX_RANK],
                      const void* buf, void* sum)
```

Description

Sum a value(s) via reduction over all processes within a group.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
sum	Reduced summation of value(s) across all processes

Return

Error code

```
int32_t tinf_iris_min(const void* comm, const enum TINF_DATA_TYPE data_type,
                  const int32_t data_rank,
                  const size_t data_dims[TINF_DATA_MAX_RANK],
                  const void* buf, void* min)
```

Description

Find the minimum value(s) via reduction over all processes within a group.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
min	Reduced minimum value(s) across all processes

Return

Error code

```
int32_t tinf_iris_max(const void* comm, const enum TINF_DATA_TYPE data_type,
                      const int32_t data_rank,
                      const size_t data_dims[TINF_DATA_MAX_RANK],
                      void* buf, void* max)
```

Description

Find the maximum value(s) via reduction over all processes within a group.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
max	Reduced maximum value(s) across all processes

Return

Error code

```
int32_t tinf_iris_rank_of_max(const void* comm,
                              const enum TINF_DATA_TYPE data_type,
                              const int32_t data_rank,
                              const size_t data_dims[TINF_DATA_MAX_RANK],
                              void* buf, void* rank)
```

Description

Find the processor rank containing the maximum value(s) found via reduction over all processes within a group.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
rank	Rank of the maximum value(s) across all processes

Return

Error code

```
void* tinf_iris_build_sync_pattern(const void *comm,
                                   const int64_t *global_ids,
                                   const size_t global_cnt,
                                   const int64_t *have, const size_t have_cnt,
                                   const int64_t *need, const size_t need_cnt,
                                   int32_t* err)
```

Description

Build the data synchronization send/receive pairs that drive the exchange of data between processes.

Parameters

comm	Opaque Iris pointer
global_ids	Array of global identifiers for data points
global_cnt	Number of global identifiers provided in *global_ids*
have	Array of local identifiers for data points residing in the calling process
have_cnt	Number of residing local identifiers provided in *have*
need	Array of remote identifiers for data points residing in other processes
need_cnt	Number of remote identifiers provided in *need*
err	Error code

Return

Opaque pointer to synchronization pattern

```
int32_t tinf_iris_delete_sync_pattern(void *comm, void *sync_pattern)
```

Description

Destroy the data synchronization send/receive pairs created with tinf_iris_build_sync_pattern.

Parameters

comm	Opaque Iris pointer
sync_pattern	Synchronization pattern to delete

Return

Error code

```
int32_t tinf_iris_sync(const void* comm, const void* sync_pattern,
                       const enum TINF_DATA_TYPE data_type,
                       const int32_t data_rank,
                       const size_t data_dims[TINF_DATA_MAX_RANK],
                       void* buf, size_t count)
```

Description

Synchronize data across processes.

Parameters

comm	Opaque Iris pointer
sync_pattern	Synchronization pattern to delete
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
count	Number of data values to synchronize

Return

Error code

```
int32_t tinf_iris_send(const void* comm, const enum TINF_DATA_TYPE data_type,
                       const int32_t data_rank,
                       const size_t data_dims[TINF_DATA_MAX_RANK],
                       void* buf, const size_t count, const int32_t dest,
                       int32_t tag)
```

Description

Perform a blocking send.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
count	Number of data values to send
dest	Process rank of the destination
tag	Message tag

Return

Error code

```
int32_t tinf_iris_recv(const void *comm, const enum TINF_DATA_TYPE data_type,
                       const int32_t data_rank,
                       const size_t data_dims[TINF_DATA_MAX_RANK],
                       void* buf, const size_t count, const int32_t src,
                       int32_t tag, int32_t* recv_tag)
```

Description

Perform a blocking receive.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
count	Number of data values to send
src	Process rank of the data source
tag	Message tag
recv_tag	Message tag of the receive message

Return

Error code

```
int32_t tinf_iris_isend(const void* comm, const enum TINF_DATA_TYPE data_type,
                        const int32_t data_rank,
                        const size_t data_dims[TINF_DATA_MAX_RANK],
                        void* buf, const size_t count, const int32_t dest,
                        int32_t tag, void* request,
                        const size_t off[TINF_DATA_MAX_RANK],
                        void* dev_addr);
```

Description

Perform a nonblocking send.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
count	Number of data values to send
dest	Process rank of the destination
tag	Message tag
request	Communication request handle
off	Offset in *buf* of starting address to send
dev_addr	Device address pointer for buf if using accelerator

Return

Error code

```
int32_t tinf_iris_irecv(const void* comm, const enum TINF_DATA_TYPE data_type,
                        const int32_t data_rank,
                        const size_t data_dims[TINF_DATA_MAX_RANK],
                        void* buf, const size_t count, const int32_t src,
                        int32_t tag, void* request,
                        const size_t off[TINF_DATA_MAX_RANK],
                        void* dev_addr)
```

Description

Perform a nonblocking receive.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
buf	Pointer to storage for *data_type* value(s)
count	Number of data values to send
src	Process rank of the data source
tag	Message tag
request	Communication request handle
off	Offset in *buf* of starting address to send
dev_addr	Device address pointer for buf if using accelerator

Return

Error code

```
int32_t tinf_iris_gather(const void *comm, const enum TINF_DATA_TYPE data_type,
                         const int32_t data_rank,
                         const size_t data_dims[TINF_DATA_MAX_RANK],
                         void *send, void *recv, int32_t rank)
```

Description

Gathers values from a group of processes.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
send	Pointer to storage for *data_type* value(s) to send
recv	Pointer to storage for *data_type* value(s) to receive
root	Source rank of receiving process

Return

Error code

```
int32_t tinf_iris_alltoall(const void *comm,
                           const enum TINF_DATA_TYPE data_type,
                           const int32_t data_rank,
                           const size_t data_dims[TINF_DATA_MAX_RANK],
                           void *send, void *recv)
```

Description

All processes send data to all processes

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
send	Pointer to storage for *data_type* value(s) to send
recv	Pointer to storage for *data_type* value(s) to receive

Return

Error code

```
int32_t tinf_iris_alltoallv(const void *comm,
                            const enum TINF_DATA_TYPE data_type,
                            const int32_t data_rank,
                            const size_t data_dims[TINF_DATA_MAX_RANK],
                            void *send, const int32_t* nsend,
                            void *recv, const int32_t* nrecv)
```

Description

Sends data from all to all processes; each process may send a different amount of data and provide displacements for the input and output data.

Parameters

comm	Opaque Iris pointer
data_type	Data type of value(s)
data_rank	Rank of the data value(s) (0 - scalar, 1 - array, etc.)
data_dims	Dimension for each *data_rank*
send	Pointer to storage for *data_type* value(s) to send
nsend	Array of length tinf_iris_number_of_processes specifying the number of elements to send to each processor
recv	Pointer to storage for *data_type* value(s) to receive
nrecv	Array of length tinf_iris_number_of_processes specifying the number of elements to send to each processor

Return

Error code

C. Problem Interface

```
_BOOL_ tinf_problem_defined(void* problem, const char* key,
                            size_t keylen, enum TINF_DATA_TYPE* datatype,
                            int32_t* rank, size_t dims[TINF_PROBLEM_MAX_RANK],
                            int32_t* err)
```

Description

Determine if the problem description defines a named parameter.

Parameters

problem	Opaque problem pointer
key	Requested parameter name (length *keylen*)
keylen	Length of the parameter name
datatype	Data type of parameter if defined. If the parameter is not defined the type is undefined.
rank	Rank of the parameter, If the parameter is not defined the rank is undefined.
dims	Array of length for each dimension (length *TINF_PROBLEM_MAX_RANK*). If the parameter is not defined this argument is undefined.
err	Error code

Return

true if problem defines the indicated parameter

```
int32_t tinf_problem_value(void* problem,
                           const char* key, size_t keylen,
                           const void* data, int32_t rank, const size_t dims[])
```

Description

Retrieve the value(s) associated with a defined named problem parameter.

Parameters

problem	Opaque problem pointer
key	Requested parameter name (length *keylen*)
keylen	Length of the parameter name
data	Storage for requested data
rank	Rank of the parameter
dims	Number of parameter requested of the given type for each dimension (length *rank* in tinf_problem_defined). This should range from 1 to the number defined as given by tinf_problem_defined

Return

Error code

D. File Based Problem

```
int32_t tinf_input_file_create(void** problem, const char* filename,
                               size_t namelen, void* comm)
```

Description

Create an problem description based on an input file and implements the problem interface.

Parameters

problem	Opaque problem pointer
filename	A file name that contains the problem description (length *namelen*)
namelen	The lenght of the filename
comm	Opaque Iris pointer

Return

Error code

```
int32_t tinf_input_file_destroy(void** problem)
```

Description

Destroy a problem description.

Parameters

problem	Opaque problem pointer

Return

Error code

E. Mesh Interface

```
int64_t tinf_mesh_node_count(void* const mesh, int32_t* error)
```

Description

Retrieve the total number of nodes in the mesh partition (resident and neighboring/ghost nodes).

Parameters

mesh	Opaque mesh pointer
error	Error code

Return

Total number of nodes in the partition

```
int64_t tinf_mesh_resident_node_count(void* const mesh, int32_t* error)
```

Description

Retrieve the number of nodes residing on (owned by) this partition.

Parameters

mesh	Opaque mesh pointer
error	Error code

Return

Number of nodes owned by this partition

```
int64_t tinf_mesh_element_count(void* const mesh, int32_t* error)
```

Description

Retrieve the number of elements in the mesh.

Parameters

mesh	Opaque mesh pointer
error	Error code

Return

Number of elements

```
int64_t tinf_mesh_element_type_count(void* const mesh, const enum TINF_ELEMENT_TYPE type,
    int32_t* error)
```

Description

Retrieve the number of elements of a given element type in the mesh.

Parameters

mesh	Opaque mesh pointer
type	Type of element (TINF_ELEMENT_TYPE) to count
error	Error code

Return

Number of elements of the given *type*

```
int32_t tinf_mesh_nodes_coordinates(void* const mesh,
                                    const enum TINF_DATA_TYPE data_type,
                                    const int64_t start, const int64_t cnt,
                                    void* x, void* y, void* z)
```

Description

Retrieve the physical coordinates of one or more mesh nodes.

Parameters

mesh	Opaque mesh pointer
data_type	identify the data type of the coordinates
start	Starting local node identifier (bias 0)
cnt	Number of local nodes to retrieve
x	X coordinate of the target node (length must be at least *cnt*)
y	Y coordinate of the target node (length must be at least *cnt*)
z	Z coordinate of the target node (length must be at least *cnt*)

Return

Error code

```
enum TINF_ELEMENT_TYPE tinf_mesh_element_type(void* const mesh,
                                              const int64_t element_id,
                                              int32_t* error)
```

Description

Retrieve the element type for a local element in the mesh.

Parameters

mesh	Opaque mesh pointer
element_id	Target local element identifier (bias 0)
error	Error code

Return

Element type from enum TINF_ELEMENT_TYPE

```
int32_t tinf_mesh_element_nodes(void* const mesh, const int64_t element_id,
                                int64_t* element_nodes)
```

Description

Retrieve the cell-to-node map for an local element in the mesh. Cell windings follow CGNS ordering.

Parameters

mesh	Opaque mesh pointer
element_id	Target local element identifier (bias 0)
element_nodes	Cell-to-node map for the element whose dimension should be large enough to hold the number of nodes for the target tinf_mesh_element_type (bias 0)

Return

Error code

```
int64_t tinf_mesh_global_node_id(void* const mesh, const int64_t local_id,
                                 int32_t* error)
```

Description

Retrieve the global node identifier for a local node in the mesh.

Parameters

mesh	Opaque mesh pointer
local_id	Target local node identifier (bias 0)
error	Error code

Return

Global node identifier (bias 0)

```
int64_t tinf_mesh_global_element_id(void* const mesh, const int64_t local_id,
                                    int32_t* error)
```

Description

Retrieve the global element identifier for a local element in the mesh.

Parameters

mesh	Opaque mesh pointer
local_id	Target local element identifier (bias 0)
error	Error code

Return

Global element identifier (bias 0)

```
int64_t tinf_mesh_element_tag(void* const mesh, const int64_t element_id,
                              int32_t* error)
```

Description

Retrieve the tag associated with an element in the mesh.

Parameters

mesh	Opaque mesh pointer
element_id	Target element identifier (bias 0)
error	Error code

Return

Associated tag

```
int64_t tinf_mesh_element_owner(void* const mesh, const int64_t element_id,
                                int32_t* error)
```

Description

Retrieve the partition identifier for the mesh.

Parameters

mesh	Opaque mesh pointer
element_id	Target element identifier (bias 0)
error	Error code

Return

Partition identifier for the element (bias 0)

```
int64_t tinf_mesh_node_owner(void* const mesh, const int64_t node_id,
                             int32_t* error)
```

Description

Retrieve the partition associated with a given node in the mesh.

Parameters

mesh	Opaque mesh pointer
node_id	Target node identifier (bias 0)
error	Error code

Return

Partition identifier for the node (bias 0)

```
int64_t tinf_mesh_partition_id(void* const mesh, int32_t* error)
```

Description

Retrieve the partition identifier for the mesh.

Parameters

mesh	Opaque mesh pointer
error	Error code

Return

Partition identifier for the opaque mesh pointer (bias 0)

F. File Based Mesh

```
int32_t tinf_preprocessor_create(void** mesh, void* problem, void* comm)
```

Description

Create a mesh preprocessor that implements the mesh interface.

Parameters

mesh	The generated mesh preprocessor object
problem	A problem description object
comm	A communications object

Return

Error code

```
int32_t tinf_preprocessor_destroy(void** mesh)
```

Description

Destroy a mesh preprocessor.

Parameters

mesh	Target mesh preprocessor object

Return

Error code

G. Linear Solver Interface

```
int32_t tinf_linear_solver_rci_create(void** linear_solver,
                                      const char* const method_name,
                                      const int32_t method_name_length,
                                      const int32_t data_type,
                                      const int64_t 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 comm,
                                      void* const sync_pattern,
                                      void** work_space)
```

Description

Create a linear system solver that implements the reverse-communication interface (RCI) to solve Ax = b.

Parameters

linear_solver	Opaque pointer to linear system solver
method_name	Linear system solver method name, eg. gmres, fgmres, or jacobi
method_name_length	Length of the method_name (including null terminator)
data_type	Data type of the dpar, work_space, b, and x arrays
rows	Number of block rows in the linear system
block_size	Number of scalar rows in each block
ipar	Integer parameter array for the reverse-communication protocol
ipar_length	Length of ipar array (minimum IPAR_LENGTH)
dpar	Pointer to storage for data parameter array storing inputs and outputs
dpar_length	Length of dpar array (minimum DPAR_LENGTH)
comm	Iris communications context
sync_pattern	Iris communications syncronization pattern
work_space	Pointer to storage utilized by the linear solver for intermediate calculations

Return

Error code

```
int32_t tinf_linear_solver_rci_destroy(void** linear_solver,
                                       void** work_space)
```

Description

Destroy a linear system solver that implements the reverse-communication interface (RCI) and free the workspace.

Parameters

linear_solver	Opaque pointer to linear system solver
work_space	Pointer to storage utilized by the linear solver for intermediate calculations

Return

Error code

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

Description

Call a reverse-communication interface (RCI) linear system solver.

Parameters

linear_solver	Opaque pointer to linear system solver
rows	Number of block rows in linear system
block_size	Number of scalar rows in each block
b	Pointer to storage for the Right-Hand-Side array
x	Pointer to storage for the solution array
ipar	Integer parameter array for the reverse-communication protocol
dpar	Pointer to storage for data parameter array storing inputs and outputs
work_space	Pointer to storage utilized by the linear solver for intermediate calculations

Return

Error code

H. Solution Interface

```
void tinf_solution_get_nodal_input_names(void* const solution,
                                         int64_t* const n_inputs,
                                         const char** names[],
                                         const enum TINF_DATA_TYPE* datatype[],
                                         const int32_t* rank[],
                                         const size_t** dims[TINF_DATA_MAX_RANK])
```

Description

Return the number and list of variable names of possible inputs at nodes

Parameters

solution	Opaque solution pointer
n_inputs	Number of inputs
names	Names of the inputs
datatype	Types of the data to set at each node
rank	Ranks of the data to set at each node
dims	Data dimensions for each rank (length *rank*)

Return

Error code

```
int32_t tinf_solution_set_inputs_at_nodes(void* const solution,
                                          const enum TINF_DATA_TYPE data_type,
                                          const int64_t start_node,
                                          const int64_t node_count,
                                          const int64_t val_count,
                                          const char* const name[],
                                          const int32_t rank,
                                          const size_t dims[TINF_DATA_MAX_RANK],
                                          const void* const values)
```

Description

Set the named inputs of the solution to varying values over a range of nodes.

Parameters

solution	Opaque solution pointer
data_type	Data type (TINF_DATA_TYPE) of output values
start_node	Starting target mesh node ID
node_count	Number of target nodes to follow *start_node*
val_count	Number of target values to set at each node
name	Array of *val_count* value names (NULL terminated) to set at each node
rank	Rank of the data to set at each node (All *val_count* values have the same rank)
dims	Data dimension for each rank (length *rank*) (All *val_count* values have the same dimensions for each *rank*)
values	Pointer to storage for *data_type* input values to set at each node (length *node_count* * *dims*[0] [* *dims*[1] ... * *dims*[rank-1])

Return

Error code

```
int32_t tinf_solution_get_nodal_output_names(void* const solution,
                                             int64_t* const n_outputs,
                                             const char** names[],
                                             const enum TINF_DATA_TYPE* datatype[])
```

Description

Return the number and list of variable names provided as output from the solution.

Parameters

solution	Opaque solution pointer
n_outputs	Number of outputs
names	Array of *n_outputs* names (NULL terminated) to set at each node
datatype	Types of the data to get at each node (length *n_outputs*)

Return

Error code

```
int32_t tinf_solution_get_outputs_at_nodes(void* const solution,
                                           const enum TINF_DATA_TYPE data_type,
                                           const int64_t start_node,
                                           const int64_t node_count,
                                           const int64_t value_count,
                                           const char* names[],
                                           void* const values)
```

Description

Get the named outputs of the solution over a range of nodes.

Parameters

solution	Opaque solution pointer
data_type	Data type of output values
start_node	Starting target mesh node ID
node_count	Number of target nodes to follow *start_node*
val_count	Number of target values to get at each node
names	Array of *val_count* names (NULL terminated) to get at each node
values	Pointer to storage for *data_type* output values to get at each node (length *node_count* * *val_count*)
err	Error code

```
int32_t tinf_solution_get_outputs_in_cell(void* const solution,
                                          const enum TINF_DATA_TYPE xyz_type,
                                          const enum TINF_DATA_TYPE data_type,
                                          const void* const x, const void* const y,
                                          const void* const z, const int64_t cell_id,
                                          const int64_t value_count,
                                          const char* names[],
                                          void* const values)
```

Description

Get solution at a location in a cell. (Overset, etc.)

Parameters

solution	Opaque solution pointer
xyz_type	Data type of target location coordinates
data_type	Data type of output values
cell_id	Target cell for solution
val_count	Number of target values to get at each node
names	Array of *val_count* names (NULL terminated) to get at each node
values	Pointer to storage for *data_type* output values to get at the target location (length *val_count*)

Return

Error code

I. Nonlinear Solver Interface

```
int32_t tinf_solver_create(void** solver, void* problem_definition,
                           void* tinf_mesh, void* iris_comm)
```

Description

Constructor.

Parameters

solver	Opaque solver pointer
problem_definition	Context defining the problem
tinf_mesh	Input opaque mesh pointer
iris_comm	Iris communications opaque pointer

Return

Error code

```
int32_t tinf_solver_destroy(void** solver)
```

Description

Destructor.

Parameters

solver	Opaque solver pointer

Return

Error code

```
int32_t tinf_solver_get_version(void* solver, const char** version)
```

Description

Return the version description of the solver.

Parameters

solver	Opaque solver pointer
version	description of solver version, e.g., "13.5-<git sha>" (output)

Return

Error code

```
int32_t tinf_solver_freeze_nodes(void* solver,
                                 const int64_t* freeze_node_ids,
                                 int64_t number_of_frozen_nodes)
```

Description

Define nodes at which solution is not to be updated (blanked).

Parameters

solver	Opaque solver pointer
freeze_node_ids	List of node indices to freeze (length *number_of_frozen_nodes*)
number_of_frozen_nodes	Number of node indices to freeze

Return

Error code

```
int32_t tinf_solver_step_pre(void* solver)
```

Description

Perform nonlinear step preprocessing (shuffle backplanes, etc.) prior to tinf_solver_step

Parameters

solver	Opaque solver pointer

Return

Error code

```
int32_t tinf_solver_step(void* solver, double seconds_to_advance)
```

Description

Perform nonlinear step preprocessing (update state)

Parameters

solver	Opaque solver pointer
seconds_to_advance	Duration of physical time to advance

Return

Error code

```
int32_t tinf_solver_step_post(void* solver)
```

Description

Perform nonlinear step post-processing (compute nonstate outputs, etc.) following a call to tinf_solver_step

Parameters

solver	Opaque solver pointer

Return

Error code