17th AIAA Computational Fluid Dynamics Conference, June 6-9 2006, Toronto, Ontario, Canada

DiRTlib: A Library to Add an Overset Capability to Your Flow Solver

Ralph W. Noack*

University of Alabama at Birmingham, Birmingham, AL, 35294, USA

The modifications to a flow solver to add an overset capability can be an impediment to developers wishing such a capability. The Donor interpolation Receptor Transaction library (DiRTlib) is a solver neutral library that simplifies the addition of an overset capability to a flow solver by encapsulating the required operations. This paper discusses the design issues of such a general capability and details the use of DiRTlib in a solver.

I. Introduction

The overset or chimera composite grid methodology has been in use for more than 20 years to simplify grid generation of complex geometries or to enable relative motion between geometry components.[1,2] Relatively recent activities have applied the approach to unstructured grids to accommodate relative motion and to eliminate the need to regrid for geometry changes.[3,4] There is significant interest in adding an overset capability to other flow solvers.

The overset composite grid utilizes a set of overlapping grids to discretize the domain and uses interpolation at appropriate points to couple the solution on the different grids. Any points that lie outside the domain of interest, for example inside of a body or behind a symmetry plane, are marked for exclusion from the computations and are termed hole or out points. Points that surround the out points become new inter-grid boundary points, which are called fringe or receptor points, and require boundary values to be applied. The boundary values required by a receptor point are provided by interpolating from a donor grid that overlaps the region. The definition of which points are out points, receptor points along with their corresponding donor members is termed the domain connectivity information and is provided by an overset grid assembly code such as the PEGASUS family for structured grids,[5,6] the SUGGAR code[7,8] for structured, unstructured and/or general polyhedral grids, or a capability that is integrated into the overset capable flow solver.[3,9]

An overset capable flow solver requires the following set of operations to be executed to produce a composite solution on the overlapping grids.

- The domain connectivity information for the composite grid system must be obtained. This may require reading a file generated by an external program such as SUGGAR or running the integral overset grid assembly process in a code such as BEGGAR.[9]
- The solution on appropriate locations in the donor grids is extracted and multiplied by interpolation weights to produce the values required for the receptor locations.

*Senior Member AIAA. CFD On-site Lead for the PET program at the Army Research Laboratory Major Shared Resource Center, Aberdeen, MD.

Distribution A: Approved for public release; distribution is unlimited.

- The interpolated values are transmitted to the appropriate location for use at the receptor locations. This may require parallel communication for distributed memory parallel programs.
- The interpolated values are applied at the receptor locations as a Dirichlet boundary value.
- The system of equations being solved should be modified to reflect the change of status of appropriate points. The out points should be decoupled from the solution equations at field points since the out points are outside the domain of interest. The equations solved at the fringe points should be changed from a regular field point to a Dirichlet boundary condition.

The set of operations just described are not large or very complex but can still be an impediment to adding to a non-overset capable flow solver.

A library that encapsulates the required operations would have several benefits.

- The library will simplify the addition of an overset capability by reducing the amount of code the flow solver developer must write.
- The time required to interface to the library will be significantly less than that required to duplicate the capability.
- The library can be used in many flow solvers and thus the library development effort can be amortized over many solvers.
- Issues that may arise, such as parallel decomposition and communication, can be solved once and shared among all solvers using the library.
- The low cost of adding and maintaining an overset capability via a library will enable the inclusion of the capability in many solvers that could not develop a native capability.

This paper describes the Donor interpolation Receptor Transaction library (DiRTlib), which is a solver neutral library that simplifies the addition of an overset capability to a flow solver by encapsulating the required operations. The design issues and details of its use are discussed.

II. Goals for the Library

The development of DiRTlib was driven with the primary goals of support for as wide a range of flow solvers as possible and to minimize the effort required to add the capability to the flow solver.

The range of formulations used in CFD flow solvers is significant. Just some of the formulation details that have an impact on the overset capability are:

- Node centered or cell centered.
- Structured or unstructured grid topology
- Coupled or segregated equation solution
- Serial or parallel execution
- Message passing library used by the solver
- Point or global equation solution

Some of these issues, such as node- or cell-centered formulations, must be accounted for within DiRTlib but have a greater impact on the grid assembly codes such as PEGASUS or SUGGAR. A tight integration of the library with flow solver data structures should be avoided to allow the library to work with any particular storage approach. Multiple ways of achieving the same operation, such as modifying the system of equations should be included to provide the solver developer the choice that simplifies the task of using the library.

III. Nomenclature

A brief description of a few terms in use in the overset community and in this paper in particular will provide a common starting point for all readers. The primary entities in an overset grid assembly are:

Hole or Out Point A hole or out point is a solution node or cell that has been defined to be outside the domain of interest and should be excluded from advancement as a typical field point. The out points typically arise from overlapping grids with some portion of the grid inside a physical body or behind a symmetry plane

Fringe or Receptor Point The points adjacent to the out points are fringe or receptor points and form a new boundary interior to the mesh and hence are also called inter-grid boundary points. Fringe points can also be found on outer boundaries of a mesh that is completely embedded in another mesh.

Donor and Donor Members A donor is the interpolation source for providing the value to be applied at the receptor/fringe points. The donor will be composed of multiple donor members that provide the data from one or more locations in the donor grid or grids and will be multiplied by interpolation weights to produce the interpolated value.

Orphan Point An orphan point is a receptor/fringe point for which no valid donor is available. The orphans can arise because of an error in marking of out points or because the grid system does not have sufficient overlap and the donor would contain donor members that are either fringe or out points, which should be avoided.

Domain Connectivity Information The domain connectivity information composed of the specification of which nodes or cells are out, orphan, or fringe points along with the corresponding donor information specified by the donor members and the interpolation weights.

IV. Issues

This section will present a number of issues that have impacted the design of DiRTlib. Details on how they are handled within the library will be provided in the next section.

A. Single Unstructured Grid

The typical unstructured grid solver has the capability to handle only a single grid while an overset composite grid will be composed of two or more grid components. Maintaining a single grid for the unstructured solver minimizes the modifications to the solver. DiRTlib must accommodate the renumbering required by working with domain connectivity referencing multiple grids while providing the solver with an interface to a single grid.

B. Parallel Decomposition

A critical capability for current flow solvers is the execution in a parallel computing environment, which can range from shared memory machines to distributed memory machines. This requires a means of decomposing the work into portions that can be executed on the different processors available. The type of decomposition used in the flow solver will primarily depend upon the grid topology. A structured grid code typically uses a mapping that assigns one or more grids to each processor. Another option is to use a subgrid mapping where a subset of the structured grid is assigned to a specific processor. The subsets of a grid may overlap and hence assign the same node or cell to multiple processors. An unstructured grid solver will typically assign a subset of the nodes and cells to a specific processor and renumber the data held on the processor. The mapping used by the solver to assign a node or cell to a processor must be provided to DiRTlib and is used to translate the indexing found in the domain connectivity information to the required processor with its local numbering.

C. Fragmented Donors

In most cases a donor will have all of its donor members from the same grid and on the same processor. A fragmented donor arises when the donor members are from different grids and/or were assigned to different processors. A common requirement would be for a structured block-to-block grid system where the donor members span a block-to-block boundary and hence are in different grids. The fragmented donor will more typically arise from the domain decomposition of an unstructured mesh where a single grid has been split among multiple processors.

D. Replicated Receptors

A replicated receptor results when the same receptor/fringe node or cell is assigned to more than one processor. This can occur with a subgrid splitting of a structured mesh or in some domain decompositions with node-centered unstructured solvers. The interpolated information must be provided to multiple locations on different processors.

E. Parallel Communication

The flow solver will typically use a message-passing library for communication and synchronization between multiple processors. It is desirable to support the two standard message-passing libraries, MPI[10] and PVM[11] in addition to serial execution.

F. Number of Dependent Variables

The number of dependent variables that a flow solver may use varies significantly depending upon the type of flow being simulated. A code that has the capability for multiple phases and species may require tens or hundreds of dependent variables. Dynamic specification of the number of dependent variables to be used in DiRTlib is required to maintain flexibility and efficiency.

G. Segregated Solvers

Most CFD flow solvers do not solve all the governing equations in a tightly coupled fashion and hence will update some solution variables in different portions of the code. Most compressible solvers will update the basic flow variables of density, velocity, energy in the same solver step but may update the turbulence parameters in another set of routines. The typical incompressible flow solver tends to the other extreme where each flow variable may be advanced independently. The impact is that the overset interpolation and communication may need to be applied to specific variables in different portions of the code. A flow solver with lots of species may store hundreds of dependent variables but will only update a few of them during a particular equation update. DiRTlib needs to be flexible in the number of dependent variables that it works with during an update.

H. Solver Storage

A flow solver may store the solution variables in numerous ways that are influenced by the storage word size, programming language, programmer style, vector or cache friendly, choice of single or multi-dimensional arrays, etc. Attempting to accommodate direct access to the myriad of possibilities in a library is not feasible and requires a more flexible interface for accessing solver storage.

I. Non-Monotone Interpolation

Interpolation via a set of linear weights is not guaranteed to produce a monotone value. The flow solver has no ability to apply the typical limiter to the interpolated value since it is being applied as a specified value boundary condition. Functionality to maintain a monotone interpolation is desirable so that the interpolation does not adversely affect solver stability

J. Donor Details

Solvers that perform a global matrix solution, such as an incompressible solver with a global matrix solution for pressure will require the details of the donor interpolation. The system of linear equations being solved must be modified to include the interpolation equation for the receptor locations. The solver will need not only the receptor locations but also the donor member location and interpolation weights.

K. Relative Motion

The overset approach is a primary enabler of simulations with bodies in relative motion. Support for moving body simulations raises requirements beyond the extra metric terms required to accurately simulate problems with grids that are in motion. The first requirement for an overset moving body problem is to perform the overset grid assembly process at appropriate time steps and to make the new domain connectivity information available to the solver. A second requirement is simply the identification of what portions of the grid are in motion along with the transformations required to specify the motion and new position.

V. Design Details

The design details of DiRTlib provide an understanding of how the library operates along with solutions to the issues discussed in the previous section. This is intended to be an overview of the concepts with more specific implementation information provided in the usage section.

A fundamental concept in DiRTlib is the donor and receptor transaction. The author uses this concept to emphasize the pairing between a fringe point and the interpolation that is performed to provide the value used at the fringe point. A complete transaction requires not only the specification and execution of the interpolation from the donor but also providing the interpolated value to the corresponding receptor/fringe location.

A. Solver Interface

A critical design requirement of DiRTlib was the separation of the internal storage from the solver storage for dependent variables. Support for any flow solver precludes making any assumptions regarding how the solver dependent variables are stored. A call back mechanism, similar to typical graphical user interface programming, is used to provide an interface to solver data. As an example, DiRTlib requires solver data values at specific locations for use in generating the interpolated values. This is achieved by having the solver developer provide a *GetDataValue* function that will return the required solver variable value. Likewise the interpolated data must be stored in the solver memory at the fringe locations. This is achieved by having the solver developer provide a *PutDataValue* function that will store the supplied value in the appropriate solver memory location. The *GetDataValue* and *PutDataValue* interface functions provide the primary interface to the solver memory. Additional interface functions may be provided for specific tasks described in a subsequent section. The solver interface functions can be simplified by allowing the solver to provide to DiRTlib a pointer to memory for each dependent variable that is then returned to the solver when DiRTlib calls the *GetDataValue* and *PutDataValue* functions. Additional solver interface functions can be supplied to perform appropriate actions at locations that have been marked as out or orphan and to modify the equations being solved at out, orphan, and receptor locations.

B. Index Mapping

The domain connectivity information provided by grid assembly codes such as PEGASUS5 and SUGGAR reference the individual component grids along with indices into those grids. DiRTlib uses mapping functions that translate the domain connectivity information indices into values that are appropriate for use in the solver execution. These mapping functions require as input the parallel decomposition that is used by the solver. The result of the mapping is the rank of the processor that contains the data along with the proper data index local to the processor. In addition, if the solver is unstructured a DiRTlib function must be called to indicate that the solver is working with a single composite grid.

For a structured grid solver where whole grids are assigned to processors the mapping is relatively simple since the index numbering does not change. An unstructured grid solver requires that the indices into individual grids be translated into a global composite grid index that is then translated to the index to the local processor.

The decomposition map that assigns a subset of the computational domain to a specific processor for parallel execution must be provided in the library initialization phase before the domain connectivity information is obtained. The rank and index mapping can then be used when loading the domain connectivity information to keep information that is required for a specific processor and to know where to send and receive the required information.

The current decomposition approach for a structured grid requires the solver to provide DiRTlib with a map that assigns each grid to a specific processor. A subgrid splitting, where a portion of the original grid is assigned to a process, is under development. Without this subgrid splitting the user must decompose the structured grids for parallel solver execution and provide the set of decomposed grids to the overset grid assembly process.

An unstructured grid solver must provide DiRTlib with the decomposition map that assigns each node or cell to a specific processor. Unique, one-to-one, maps are produced when each node or cell is assigned to a single processor. The index local of a node or cell is assumed to be defined by the same order that they are assigned to a processor and implicitly defines a global-to-local and local-to-global map. If the solver uses a different ordering of the data on the processor an explicit global-to-local map must be provided to DiRTlib.

Some solvers assign a location to more than one processor and will produce a replicated receptor if the location is a fringe or receptor. In this case the decomposition map must specify all the processors that contain the same node or cell. The first location that is specified is the primary location and will be used for donor members if needed. The list of other processors that contain the replicated node are stored and if the location is a fringe or receptor then a replicated receptor will result as described in Section V.E. If the solver uses an ordering of the data on the processor different from the implicit assignment created by DiRTlib an explicit global-to-local map must be provided to DiRTlib.

C. Donor Interpolation

The donor interpolation part of the transaction uses the solver provided *GetDataValue* function to obtain data values from the solver for the donor members. The sum of the donor data values multiplied by the appropriate interpolation weight is stored for later transmission to the appropriate receptor.

D. Fragmented Donors

The previous section indicated that fragmented donors, where the donor members may be split across multiple grids or processors, is a common occurrence for a variety of reasons. DiRTlib handles this by splitting the donor and receptor transaction into multiple transactions with all the donor members in a transaction residing in the same grid or processor. A complete transaction requires a matching receptor and so a fragmented receptor is created to match each donor fragment. The interpolated values sent to each receptor fragment are summed to provide the complete interpolated value to be stored in solver memory for the receptor using the call to the solver provided *PutDataValue* function.

E. Replicated Receptor

As indicated in Section V.B, a parallel decomposition may assign a location to more than one processor. If the location is a fringe or receptor then special action is required by DiRTlib to provide all the locations with the same information. Such a fringe location is called a replicated receptor and is handled by simply creating duplicate donor receptor transactions for each replicated receptor. This approach will require extra interpolations and hence extra calls to the *GetDataValue* function but simplifies the code.

F. Number of Variables and Masking

The solver is required to inform DiRTlib of the number of variables that DiRTlib will store and work with at any given time. The efficiency is further improved by allowing the solver to specify a dependent variable mask. Only those variables that are set to ON in the mask will be active during the donor interpolation and receptor transactions.

The solver can also change the solver interface functions and the dependent variable pointers that are used in DiRTlib during the execution. This simplifies the writing of the solver interface functions. Functions tailored to a specific variable can be written rather than writing one that must handle all the different variables.

For example, consider an incompressible solver that segregates the solution into a pressure equation solution, followed by a solution for the three velocity components, and lastly updates a two equation turbulence model. Only three data values need to be stored in DiRTlib at any one time. The solver will call a function to set the variable mask to indicate the variables that are active for each equation solution and call functions to set the use *GetDataValue* and *PutDataValue* interface functions appropriately:

1. Set the number of dependent variables to 3 which is the maximum number that will be exchanged at a time.
2. Set the variable mask to all variables active
3. Set *GetDataValue* and *PutDataValue* to functions for velocity.
4. Solve equations for the 3 velocity components.
5. Set variable mask so only the first variable is active.
6. Set *GetDataValue* and *PutDataValue* to functions for pressure.
7. Solve pressure equation
8. Set the variable mask to two variables active
9. Set *GetDataValue* and *PutDataValue* to functions for turbulence model.
10. Solve equations for turbulence.

G. Optional IBLANK array

DiRTlib can fill a solver provided array with values that indicate the locations marked as out, orphan, or receptor/fringe points. The typical solver uses this array, which is historically called IBLANK, to provide control of execution flow and to modify the system of equations. The use of an IBLANK array is not required but can eliminate the need for some interface functions and simplify the use of DiRTlib.

Two methods are supported for setting appropriate values in the solver IBLANK array. The simplest method is for DiRTlib to set values in the array directly. This violates the principal of not touching the solver memory directly and assumes a particular storage format. Many solvers will allocate the IBLANK as an 4 byte integer array that is not padded with ghost cell storage and it is simpler for those solvers to have DiRTlib set the values directly.

A more general call back based interface is available for solvers where the storage does not match this simple definition. In this case the solver writer provides a *SetIblankValue* interface function that DiRTlib will call and the interface function is responsible for setting the value in the appropriate location.

H. Parallel Communications

DiRTlib supports using message passing libraries for parallel communications in addition to supporting serial execution. For convenience and performance the data to be transfered is copied from internal storage for the donors into a transmission buffer before sending to the appropriate location. Likewise the receive operation places data into a transmission buffer from which it is scattered into appropriate receptor storage structures. If the data destination is local to the processor then the transmission consists of a memory copy from the send buffer into the receive buffer.

The transmission of the data to another processor is made via calls to DiRTlib functions that wrap the appropriate message passing interface library function calls to abstract the DiRTlib transfer code and isolate it from dependencies on the message passing library. The solver's calls to DiRTlib are almost completely independent of the message library used. Versions of the DiRTlib parallel initialization routines are also available for serial execution so that the solver's calls to DiRTlib do not need to be changed for serial execution.

I. Interpolation Clipping

An optional clipping procedure has been implemented to address the issue of non-monotone interpolation. The procedure finds the minimum and maximum value of the dependent variable values obtained from the donor member locations and then restricts or clips the interpolated value to be within that range. The possibility of a fragmented donor precludes the application of the clipping during the donor interpolation phase since the donor fragment on a particular processor does not contain all the donor information. The present procedure finds the minimum and maximum values for the donor fragment and includes them in the data transmitted to the receptor location. The minimum and maximum of all the donor fragments for a receptor is then used to clip the final interpolated value that is stored in the receptor location.

J. Donor Details

DiRTlib provides the solver with a mechanism to obtain the details of the donor interpolation. This can be used to build the interpolation into a global matrix or for debugging output. A call back mechanism is used so that DiRTlib does not modify solver data storage. The solver will provide an interface function that DiRTlib will call with appropriate arguments. The solver interface function will in turn call other DiRTlib functions to obtain the required donor details consisting of donor member indices and weights.

K. Relative Motion

An overset moving body problem requires that the overset grid assembly process be repeated at appropriate time steps with the bodies properly positioned. This involves communication and synchronization with the grid assembly process. The SUGGAR grid assembly code uses a simple mechanism to perform the communication and synchronization with the flow solver and DiRTlib provides some functions, which will not be detailed here, that are useful in performing the communication functions. Once the new domain connectivity information is available the flow solver must call the DiRTlib functions to load this domain connectivity information again. DiRTlib will first free the storage used by the previous domain connectivity information to prevent a memory leak.

A second requirement is the identification of what portions of the grid are in motion along with the transformations required to specify the motion and new position. The motion can be specified or controlled by rigid body dynamics that is integral to the flow solver or is an external process. The SUGGAR specific domain connectivity information file contains an association of the individual component grids to a moving body along with the matrix that transforms the grid systems from their original coordinate systems to the current location. DiRTlib provides functions to obtain the transformation matrix for a particular moving body and to assign a moving body index to each cell in the mesh. The solver can then use this information to transform the appropriate grid nodes to properly position the moving bodies. Details on these functions are omitted from this paper due to space constraints.

L. Programming Language Bindings

DiRTlib is written in the C programming language but FORTRAN bindings are available through wrapper functions. The C functions will all start with the prefix "drt_" while the FORTRAN bindings all start with the prefix "drtf_". The FORTRAN function names will follow those of the C application programmer interface (API) except when the name is longer than allowed by certain FORTRAN compilers. The usual mangling of names by FORTRAN compilers are all supported: no trailing underscore, one trailing underscore, and two trailing underscores, The compiled library file contains the C and the FORTRAN bindings with all mangled options so that the same library file can be linked into C or FORTRAN codes.

The API presented in this paper will be primarily for the C programming language. Some FORTRAN examples will be provided to illustrate specific programming requirements.

VI. DiRTlib Usage

The usage of DiRTlib along with the API will be illustrated in this section using an unstructured flow solver the primary target. Changes required for a structured solver will also be highlighted. In the following discussion we will use the nouns grids and meshes interchangeably.

The unstructured solver will treat the set of overlapping composite grids as a single grid and thus the unstructured solver will call convenience functions that operate on all the grids. The structured grid solver can also call these convenience functions when appropriate. There are some cases where the solver may need to call functions that operate on individual grids. These functions are available but will not be completely discussed in this paper.

Additional DiRTlib functions and capability are omitted from this paper due to space limitations.

A. Solver Interface Functions

The first step in integrating DiRTlib into a solver is to write the solver interface or call back functions. These function are called by DiRTlib to interact with the solver storage. An approach that minimizes the modifications to the flow solver will require five solver interface functions. An additional interface function is required if the solver uses an IBLANK array that is not allocated consistently with the DiRTlib assumptions described in Section V.G. DiRTlib will call the solver interface functions when the flow solver calls specific DiRTlib functions at appropriate times as described in subsequent sections.

The first two solver interface functions, *GetDataValue* and *PutDataValue*, GET data from the solver memory and PUT data into the solver memory. DiRTlib will call the *GetDataValue* function to retrieve solver data values to be used for interpolating values for the receptor points. The *PutDataValue* function will be used to store the interpolated values into solver memory.

The third function, *BlankLocation*, is called for any grid location that is marked as out. The function should set solver memory to something appropriate at these locations so that the solver computations at these location will not diverge. The preferred approach is to average with neighboring values so that an appropriate value will be used if the location changes from out to a field location in a moving body simulation.

The forth function, *OrphanLocation*, is called for any grid location that is marked as an orphan. The function should set solver memory to something appropriate at these locations. The usual approach is to use an average of the neighboring values.

The fifth function, *FlagBlankFringePoints*, is used to modify the solver equations at the fringe and out locations. The equations should be modified from the standard field equations to be a boundary condition with specified values. This function is not needed if the solver uses an IBLANK array to modify the equations and and calls the appropriate DiRTlib functions to fill the IBLANK array.

The C function prototypes for the calls are:

```
void PutDataValue(int *pg,int *pi,int *pk,double *pv,void *s_array);
double GetDataValue(int *pg,int *pi,int *pk, void *s_array);
void BlankLocation(int *pg,int *pi, int *pk,void *s_array);
void OrphanLocation(int *pg,int *pi, int *pk,void *s_array);
void FlagBlankFringePoints(int *pg,int *pi, int *pk,void *s_eq_array);
```

The indexing of the *pg*, *pi*, *pk* variables assume a starting index of one to match the usual FORTRAN convention. The use of pointers to the *pg*, *pi*, *pk*, and *pv* arguments was chosen to match the FORTRAN calling conventions.

The *pg* variable is the index for a grid. For an unstructured grid the solver will treat the set of overlapping grids as a single grid and the grid index is ignored.

The *pi* variable is the index of the cell (for a cell centered solver) or the grid point/node (for a node-centered solver). This is a one-dimensional array index that the solver may have to convert into multi-dimensional array indices. An example of how this is done is provided later in this section.

The *pk* variable is the index of the dependent variable.

The *pv* variable is the value of the dependent variable to be stored in solver memory.

The *GetDataValue* function should return a double/real*8 value. All computations within the library are performed using double precision values.

The *s_array* variable is a pointer to an optional solver array that can be used to simplify access to the proper solver memory. This may be required if the solver does not use globally accessible arrays for storing the dependent variable arrays. A DiRTlib function is provided to store the appropriate array starting address or pointer. This address is passed in the call to the above solver interface functions through the *s_array* variable. If the solver arrays are globally accessible then this argument can be ignored. The above interface specification types the *s_array* as void* but the type in the solver provided function must match the actual solver array type.

The *s_eq_array* variable is also a pointer to an optional solver array with the distinction that this should be a pointer to the linear equations array and is set and saved independently from the *s_array* variable.

A simple example will illustrate these functions. The *s_array* variable is used, which allows simple indexing into the solver memory. The solver has provided pointers to the start of the array for each independent variable. Thus the s_array is case to the proper type, which in this example is a double and is simply indexed to access the proper value. A starting value of one is used for the index passed into the function and hence one is subtracted to properly index into the dependent variable array.

```
void PutDataValue(int *pg,int *pi,int *pv,double *pv,void *s_array)
{
  double *qarray = (double *)s_array;
  int index = *pi;
  qarray[index-1] = *pv;
}
double GetDataValue(int *pg,int *pi,int *pk, void *s_array)
{
  double value;
  double *qarray = (double *)s_array;
  int index = *pi;
  value = qarray[index-1];
  return value;
}
```

A more complex example is illustrated for a structured grid flow solver written in FORTRAN90. The solver array is a single precision multidimensional array and the index must be decoded into structured grid indices. The array dimensions are available in a common block and an auxiliary function is used to decode the single dimension index into the multidimensional array indices.

```
      subroutine index2ijk(index, i, j, k, idim, jdim, kdim)
c-----Decode the index argument into three-dimensional component integers
c-----The dimesions in each direction are provided by the idim, jdim, kdim arguments
      implicit none
      integer :: i, j, k, n, ij, idim, jdim, kdim, indexm1

        indexm1 = index - 1
        ij = idim * jdim
        k =  indexm1 / ij  + 1
        j = (indexm1 - (k - 1)*ij )/idim + 1
        i =  indexm1 - (k - 1)*ij - (j - 1)*idim + 1

      end subroutine

      integer function ijk2index(i, j, k, idim, jdim, kdim)
c-----convert the three-dimensional component indices
c-----into the equivalent single dimension
c-----The dimesions in each direction are provided by the idim, jdim, kdim arguments
      implicit none
      integer :: i, j, k, n, idim, jdim, kdim

        ijk2index = i + (j - 1)*idim +  (k - 1)*idim*jdim

      end function

     subroutine PutDataValue(grid, index, v, value,q)
       integer :: imax, jmax, kmax, nvar
       common /dep_variables/ imax, jmax, kmax, nvar

       integer :: grid, index, v
       real*8  :: value
       real*4  :: q(imax,jmax,kmax,nvar)

c-----Decode the index argument into three-dimensional component integers
       call index2ijk(index, i, j, k, idim, jdim, kdim)
c-----Store the interpolated value in the q array
       q(i,j,k,v) = value

     end subroutine PutDataValue
     real*8 function GetDataValue(grid, index, v,q)
       integer :: imax, jmax, kmax, nvar
       common /dep_variables/ imax, jmax, kmax, nvar

       integer :: grid, index, v
       real*8  :: value
       real*4  :: q(imax,jmax,kmax,nvar)

c-----Decode the index argument into three-dimensional component integers
       call index2ijk(index, i, j, k, idim, jdim, kdim)
c-----Store the interpolated value in the q array
       GetDataValue = q(i,j,k,v)
     end function GetDataValue
```

The routines *OrphanLocation* and *FlagBlankFringePoints* will vary significantly between solvers and no example will be provided. Instead we will illustrate the function calls to set a solver provided IBLANK array. The solver should have allocated the IBLANK array to match the C int data type. Two calls are provided to separately set the IBLANK values at fringe and out locations. These calls will be discussed in the Section VI.I.

B. Initialization

Several calls are required to initialize the library. The initialization will differ depending on if the solver is structured or unstructured, serial or parallel execution, and static or a moving body simulation.

For parallel execution the unstructured solver will also load a decomposition map into DiRTlib that assigns the cell or node to a particular rank rather than a grid to processor rank map. The discussion of this portion of the initialization is deferred until Section VI.C.

1. Flag to Treat Set of Grids as a Single Grid

The unstructured solver will treat the set of overlapping composite grids as a single grid and the first DiRTlib function that must be called sets DiRTlib into this mode. The C prototype of the function is:

```
void drt_solver_uses_single_grid(void);
```

2. Number of Grids for Structured Solver

A structured solver will maintain the individuality of each grid and DiRTlib must know the total number of grids in the system. This call will replace the *drt_solver_uses_single_grid* call. The C prototype to inform DiRTlib of the number of grids is:

```
void drt_set_number_grids(int number_grids);
```

3. Enabling Interpolation Clipping

As discussed in Section V.I DiRTlib has the ability to clip or limit the interpolated value to be between the minimum and maximum values from the donor members. This capability has been found to improve the stability of some flow solvers.

To enable the capability the flow solver must call the *drt_clip_interpolation_to_min_max* function. The call must be made made before the domain connectivity information is read and only needs to be called once. The C prototype to inform DiRTlib of the number of grids is:

```
void drt_clip_interpolation_to_min_max(int on_off);
```

A value for *on_off=1* will enable the capability and *on_off=0* will disable it.

C. Parallel Execution

The initialization for parallel execution requires informing the library of the rank of the process, the total number of processors, and details of the parallel decomposition. The unstructured solver will also load a decomposition map into DiRTlib that assigns the cell or node to a particular rank while the structured solver will provide DiRTlib with a map that assigns a grid to a particular rank.

1. Initialization for Parallel Execution

The first call to DiRTlib should set the processor rank along with the total number of processes. The rank indexing should start at 0. The C prototype for this initialization function is:

```
void drt_pll_init(int my_rank,int num_processors);
```

2. Unstructured Solver Parallel Decomposition

The parallel decomposition map for an unstructured solver can be loaded from solver memory or from a file. Two different routines can be used for loading the map from solver memory. The first saves a pointer to the solver memory and the solver should not deallocate the array. The C prototype for this option is:

```
void drt_load_decomposition_map(int *map,int len_map,int min_rank);
```

The map argument is the array containing the decomposition map. The *len_map* argument is the length of the map array. The *min_rank* argument is the minimum rank value used in the map. This will usually be 0.

The second option is call a routine that creates a copy of the solver provided array and allows the solver to deallocate the array:

```
void drt_copy_decomposition_map(int *map,int len_map,int min_rank)
```

The final option loads the decomposition map from a file:

```
void drt_load_decomposition_map_from_file(char *filename)
```

The file is a text file with a line in the file containing the processor rank assignment for each node or element in the grid.

3. Structured Solver Parallel Decomposition

For parallel execution the structured solver will also load a grid-to-processor map into DiRTlib that assigns a grid to a particular rank, which should follow the call to set the number of grids. The map should be of length equal to the number of grids and contain the rank assigned to the grid index. The C function prototype is:

```
void drt_set_grid_to_processor_rank_map(int *grid2pe_map);
```

D. Set the Number of Data Values

The next step is to set the maximum number of data values or dependent variables that DiRTlib will be managing at any one time. For a typical three-dimensional Euler solver with the usual dependent variables: density, three velocity components, and energy, the number of data values will be 5.

A segregated solver will break the solution process into different equation sets that are solved independently. In this case DiRTlib will be used to process a subset of the dependent variables at any given time.

The C prototype of the function to set the maximum number of data values that DiRTlib will manage at a time is:

```
void drt_set_num_data_values_all_grids(int num_data_values);
```

E. Load Domain Connectivity Information

The domain connectivity information (DCI) specifies the donors (interpolation sources), receptors (fringe locations needing interpolated data), and other locations that are marked as out or orphan. DiRTlib provides routines to load this data from the file formats produced by the PEGASUS5 and SUGGAR grid assembly codes.

For an unstructured grid code the format that is currently supported is the flex file produced by SUGGAR. The C prototype for the function to load a flex file is:

```
void drt_load_flex_grid_drt_file(char *dci_file_name);
```

The FORTRAN interface requires that the filename be NULL terminated, which is achieved by appending the char(0) character to the filename as shown:

```
CHARACTER(Len=80):: dci_file_name
dci_file_name = "gen_drt.dci" // char(0)
call drtf_load_flex_grid_drt_file(dci_file_name)
```

For a node-centered structured grid code the most common format it the PEGASUS5 XINTOUT. A filename and the number of meshes in the system are passed as arguments to the function calls to load the PEGASUS5 file. For the FORTRAN call the string should be NULL terminated, which is again achieved by appending CHAR(0) to the end of the file name. The C prototype to load an XINTOUT style file is: void

```
drt_load_structured_drt(char *xintout_file_name,int number_grids);
```

The corresponding FORTRAN call is:

```
integer number_grids
CHARACTER(Len=80):: xintout_file_name
xintout_file_name =  "XINTOUT" // char(0)
call drtf_load_structured_drt(xintout_file_name,number_grids)
```

F. Initialize DiRTlib

The final initialization call passes the solver interface functions to DiRTlib so that the library can call them at the appropriate times. If an IBLANK array is used to perform the required actions, the *BlankLocation*, *OrphanLocation*, and *FlagBlankFringePoints* functions are not required and a NULL value can be used in those argument locations.

The C prototype for the function to initialize the solver interface functions is:

```
void drt_Init(  void (*PutDataValue)(int *pz,int *pi,int *pk,double *pv,void *s_array),
              double (*GetDataValue)(int *pg,int *pi,int *pk,void *s_array),
               void (*BlankLocation)(int *pg,int *pi,int *pk,void *s_array),
              void (*OrphanLocation)(int *pg,int *pi,int *pk,void *s_array),
       void (*FlagBlankFringePoints)(int *pg,int *pi, int *pk,void *s_array)  );
```

The corresponding FORTRAN90 syntax requires the specification of an interface block to provide the equivalent of the C function prototype. An example is:

```
  interface
     subroutine putDataValue(pg, pi, pk, pv,s_array)
       integer :: pg, pi, pk
       real*8 :: pv
       real*4 :: s_array(*)
```

```
      integer :: parm_num

    end subroutine putDataValue

    real*8 function getDataValue(pg, pi, pk,s_array)
      integer :: pg, pi, pk
      real*4 :: s_array(*)
      integer :: parm_num
    end function getDataValue

    subroutine BlankLocation(pg, pi, pk, s_array)
      integer :: pg, pi, pk
      real*4 :: s_array(*)
    end subroutine BlankLocation

    subroutine OrphanLocation(pg, pi, pk, s_array)
      integer :: pg, pi, pk
      real*4 :: s_array(*)
    end subroutine BlankLocation

    subroutine FlagBlankFringePoints(pg, pi, pk, s_array)
      integer :: pg, pi, pk
      real*4 :: s_array(*)
    end subroutine FlagBlankFringePoints

  end interface
  call drtf_Init(PutDataValue,GetDataValue, &
                 BlankLocation,OrphanLocation,FlagBlankFringePoints)
```

G. Setting the *s_array* and *s_eq_array* Pointers

The *s_array* and *s_eq_array* pointers are passed to the solver interface functions to provide access to these arrays when they are not globally accessible and their use can simplify the interface function. DiRTlib functions are provided to pass these array addresses or pointers to DiRTlib, which will then provide them as arguments to the solver interface function calls. A separate array pointer can be stored for each dependent variable. The C prototype for the function is:

```
void drt_set_solver_data_pointer_all_grids(int *pk,void *s_array);
```

where *pk* is the index of the dependent variable for which to store the specified *s_array*.

In a similar fashion the pointers to solver arrays for the linear equations can be stored in DiRTlib for passing back to the FlagBlankFringePoints solver interface function. The C prototype for the function is:

```
void drt_set_solver_eq_data_pointer_all_grids(int k,void *s_eq_array);
```

H. Modification of Linear Equations.

The linear equations at the out, orphan, and fringe locations should be modified to reflect a specified value type boundary condition. Two options are supported in DiRTlib for assisting the flow solver in modifying the linear equations to be solved. The conventional approach is for the solver to use an IBLANK array and switch the behavior based upon the values in the array. The next section describes the DiRTlib functions to set the IBLANK values at the fringe, out, and orphan locations.

A second approach is available for the solvers that do not have an IBLANK array. The solver will call a DiRTlib function that will then call the solver interface function at each out, fringe, and orphan location. The solver interface function should modify the solution equations appropriately to treat the locations as specified value boundary conditions.

The first set of routines allows the solver to process the different locations independently. The C prototypes for these routines are:

```
void drt_flag_blank_points_all_grids(void);
void drt_flag_fringe_points_all_grids(void);
void drt_flag_orphan_points_all_grids(void);
```

The call to *drt_flag_blank_points_all_grids* will cause DiRTlib to call the *BlankLocation* interface function at all locations that have been marked as out. Similarly the *drt_flag_fringe_points_all_grids* will call *FlagBlankFringePoints* and *drt_flag_orphan_points_all_grids* will call *OrphanLocation* interface functions at fringe and orphans locations respectively.

These functions should not be called if the user is using the contents of an IBLANK array to modify the system of linear equations.

A convenience function is available that simply calls the three individual functions and the C prototype is:

```
void drt_flag_blank_fringe_orphan_points_all_grids(void);
```

I. Filling IBLANK Arrays

Most solvers currently using DiRTlib have an IBLANK array that is used to modify the system of linear equation and to identifying orphan and out locations that require solution averaging. DiRTlib provides two different methods for setting the IBLANK array values at fringe, out, and orphan locations. The solver is responsible for initializing the IBLANK array to the value appropriate for the field points.

The first approach allows DiRTlib to directly alter values in the IBLANK array. DiRTlib assumes that the IBLANK array is allocated as a four byte integer without any padding for ghost cells. This storage format will be the case for many unstructured solvers and some node-centered structured grid solvers. If the solver allocated the IBLANK array in this manner then the solver can call the following functions to set the value at fringe, out, and orphan locations to the value provided by the second argument. A separate call is used for each location type to allow the solver to set different values for each type. The C function prototypes are:

```
void drt_fill_iblank_for_fringe_points_all_grids(int *iblank,int value);
void drt_fill_iblank_for_out_points_all_grids(int *iblank,int value);
void drt_fill_iblank_for_orphan_points_all_grids(int *iblank,int value);
```

An approach that uses a call back function is also available for solvers where the IBLANK array does not match the DiRTlib assumptions. The solver must provide a *SetIblankValue* call back function, which is similar to the *PutDataValue* function, that DiRTlib will call at each fringe, out, and orphan location. A separate call is again used for each location type to allow the distinct values to be assigned for each type of location.

The C function prototypes for a structured grid solver that will loop over the grid indices are:

```
void drt_set_iblank_for_out_points(int grid_index,
         void (*SetIblankValue)(int *pg,int *pi,int *value,void *iblank_array),
                                    int value,void *iblank_array);
void drt_set_iblank_for_orphan_points(int grid_index,
         void (*SetIblankValue)(int *pg,int *pi,int *value,void *iblank_array),
                                      int value,void *iblank_array);
void drt_set_iblank_for_fringe_points(int grid_index,
         void (*SetIblankValue)(int *pg,int *pi,int *value,void *iblank_array),
                                      int value,void *iblank_array);
```

The unstructured solver does not loop over the individual grids and should call the convenience functions specified by the following C function prototypes:

```
void drt_set_iblank_for_out_points_all_grids(
        void (*SetIblankValue)(int *pg,int *pi,int *value,void *iblank_array),
                                             int value, void *iblank_array);
void drt_set_iblank_for_orphan_points_all_grids(
         void (*SetIblankValue)(int *pg,int *pi,int *value,void *iblank_array),
                                                int value, void *iblank_array);
void drt_set_iblank_for_fringe_points_all_grids(
         void (*SetIblankValue)(int *pg,int *pi,int *value,void *iblank_array),
                                                int value, void *iblank_array);
```

J. Overset Donor and Receptor Communication

The overset approach requires the communication of information between the interpolated values from donors in one grid and the receptors in another grid. DiRTlib uses a set of routines to provide this functionality.

This communication is decomposed into the following phases:

- Generation and Send
  - Generation of interpolated values
  - Gathering the donor values into a send buffer
  - Sending of the donor data to the appropriate receptor
- Receive and Application
  - Receiving of donor values
  - Scatter from the receive buffer
  - Application of values at receptor locations

The solver will typically perform this overset communication after the solution has been updated at a time step or Newton inner iteration. DiRTlib provides functions that match these phases and the C function prototypes for the functions are:

```
void drt_dv_generation_all_grids(void);
void drt_gather_into_xfer_send_buffer(void);
void drt_send_all(void);

void drt_recv_all(void);
void drt_scatter_from_xfer_recv_buffer(void);
void drt_dv_apply_receptor_values_all_grids(void);
```

These six functions are typically called in order as a group and a convenience function is available that allows the solver to call the following function rather than the above six functions:

```
void drt_generate_transmit_apply(void)
```

The simplest approach is to call the individual generation and send functions followed by the receive and application functions as shown above and use the *drt_generate_transmit_apply* convenience function. The parallel communication time can be hidden by allowing computations to be performed between the Send and Receive calls. This can be achieved by putting the send set of function before the solver advances the solution to the new time level allowing the parallel communications to take place while the solver is advancing the solution. The receive calls can then follow the solution update to use the transmitted data. The disadvantage of this approach is the the solution update is using data that is lagged by one iteration.

K. Set Values at Out and Orphan Locations

A function is provided to allow the solver to set dependent variable values at the out and orphan locations and it is appropriate to call this function at the same time as the function to apply receptor values is called. These functions would not be called if the solver uses an IBLANK array and performs an averaging or other operation on the out and orphan points. The C function prototypes are:

```
void drt_blank_points_all_grids(void);
void drt_orphan_points_all_grids(void);
```

VII. Current Usage of DiRTlib

The utility of DiRTlib and its API have been validated by its use in a number of flow solvers covering a wide range of grid topology and formulations. Papers on the use of DiRTlib in some of the solvers are also available at this meeting.[12,13] The list of flow solvers currently using DiRTlib will now be provided.

A. Structured Grid Codes

1. NXAIR

NXAIR[14] is a compressible structured-grid node-centered solver that has had an overset capability from many years for static and moving body problems. DiRTlib is being used to provide a more efficient overset transaction capability.

2. WIND

The WIND code[15] is a compressible block-to-block or overset structured-grid node-centered solver with chemistry with a capability for static and moving body problems. DiRTlib is being used to provide an alternative overset transaction capability.

3. Over-Rel

The Over-Rel code,[16] a derivative of the UNCLE[17] code, is an incompressible block-to-block structured grid node-centered solver. DiRTlib is being used to provide an overset grid capability for static and moving body problems.

4. UNCLE-M

The UNCLE-M code,[18] a derivative of the UNCLE[17] code, is an all-speed pressure based block-to-block structured-grid node-centered multi-phase flow solver. DiRTlib is being used to provide an overset grid capability for static and moving body problems.

5. *FDNS*

The FDNS code[19,20] is an incompressible block-to-block structured-grid node-centered solver with chemistry. DiRTlib is being used to provide an overset grid capability for static and moving body problems.

B. Unstructured-Grid Codes

1. *USM3D*

USM3D[12,21] is a tetrahedral based unstructured-grid cell-centered compressible solver. DiRTlib is being used to create an unstructured overset grid capability for static and moving body problems.

C. General Polyhedral Unstructured-grid Codes

1. *NPHASE*

NPHASE[22] is a general polyhedral unstructured-grid cell-centered, pressure based, segregated solver written in the C programming language. It uses an all speed formulation and has a multi-phase and multi-species capability. DiRTlib is being used to create an unstructured overset grid capability for static and moving body problems. The overset transactions have been incorporated into the global pressure solver to maintain convergence properties.

2. *HYB3D*

HYB3D[13,23,24] is a general polyhedral unstructured-grid cell-centered compressible solver. DiRTlib is being used to create an unstructured overset grid capability for static and moving body problems.

3. *COBALT*

COBALT[25] is a general polyhedral unstructured-grid cell-centered compressible solver. DiRTlib is being used to create an unstructured overset grid capability for static and moving body problems.

VIII. Summary

A library that encapsulates the operations required of a flow solver for an overset capability has been developed and presented. The library called Donor interpolation Receptor Transaction library (DiRTlib) was developed to be solver neutral and simplify the addition of an overset capability for static and moving body problems. The design issues and details of its implementation and use were discussed.

The validity and usefulness of DiRTlib have been demonstrated by its use in nine CFD flow solvers at the present time. The capabilities of these solvers cover a wide range of the possible formulations including compressible and incompressible, node- and cell-centered, with and without chemistry, structured, unstructured, and general polyhedral unstructured grids.

Acknowledgments

This publication was made possible through support provided by the Department of Defense (DOD) High Performance Computing Modernization Program (HPCMP) Programming Environment and Training (PET) activities through Mississippi State University under the terms of Contract No. N62306-01-D-7110. The support of the U.S. Army Research Laboratory Major Shared Resource Center, Aberdeen Proving Ground, MD is also gratefully acknowledged.

The opinions expressed herein are those of the authors and do not necessarily reflect the views of the DOD or Mississippi State University.

References

[1]Benek, J. A., Steger, J., and Dougherty, F., "A Flexible Grid Embedding Technique with Applications to the Euler Equations," Paper 83-1944, AIAA, 1983.

[2]R. L. Meakin, "Unsteady Simulation of the Viscous Flow About a V-22 Rotor and Wing in Hover," *AIAA Atmospheric Flight Mechanics Conf.*, 95-3463-CP, 1995, pp. 332–344.

[3]Nakahashi, K., Togashi, F., and Sharov, D., "Intergrid-Boundary Definition Method for Overset Unstructured Grid Approach," *AIAA Journal*, Vol. 38, No. 11, 2000, pp. 2077–2084.

[4]F. Togashi, e. a., "Flow Simulation of NAL Experimental Supersonic Airplane/Booster Separation," Paper 2000-1007, AIAA, 2000.

[5]Suhs, N. and Tramel, R., "PEGSUS 4.0 User's Manual," TR 91-8, AEDC, 1991.

[6]Suhs, N., Rogers, S., and Dietz, W., "PEGASUS 5: An Automated Pre-processor for Overset-Grid CFD," Paper 2002-3186, AIAA, 2002.

[7]R. W. Noack, "Resolution Appropriate Overset Grid Assembly for Structured and Unstructured Grids," Paper 2003-4123, AIAA, 16th AIAA Computational Fluid Dynamic Conference, Orlando, FL, 2003.

[8]R. W. Noack, "SUGGAR: A General Capability for Moving Body Overset Grid Assembly," Paper 2005-5117, AIAA, 17th AIAA Computational Fluid Dynamic Conference, Toronto, Ontario, Canada, 2005.

[9]Belk, D. and Maple, R., "Automated Assembly of Structured Grids for Moving Body Problems," *Proceedings of 12th AIAA Computational Fluid Dynamics Conference*, AIAA Paper 95-1680-CP, San Diego, CA, 1995.

[10]W. Gropp and E. Lusk and A. Skjellum, *Using MPI, Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, 1999.

[11]Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, B., and Sunderam, V., *PVM: Parallel Virtual Machine– A Users' Guide and Tutorial for Network Parallel Computing*, The MIT Press, 1994.

[12]Frink, N. T., Pandya, M. J., and Noack, R., "Overset-Grid Moving Body Capability in the USM3D Unstructured Flow Solver," Paper 2005-5118, AIAA, 2005.

[13]Cheng, G., Koomullil, R., and Noack, R., "A library Based Overset Capability Development for Density- and Pressure-Based Flow Solvers," Paper 2005-5119, AIAA, 2005.

[14]Tramel, R. W. and Nichols, R. H., "A Highly Efficient Numerical Method for Overset-Mesh Moving-Body Problems," Paper 97-2040, AIAA, 1997.

[15]R. H. Bush, G. D. Power, and C.E. Towne, "WIND: The Production Flow Solver of the NPARC Alliance," Paper 98-0935, AIAA, 1998.

[16]Boger, D. A. and Dreyer, J. J., "Prediction of Hydrodynamic Forces and Moments for Underwater Vehicles Using Overset Grids," Tech. rep., In Preparation, 2006.

[17]Taylor, L. K., Arabshahi, A., and Whitfield, D. L., "Unsteady Three- Dimensional Incompressible Navier-Stokes Computations for a Prolate Spheroid Undergoing Time-Dependent Maneuvers," Paper 95-0313, AIAA, 1995.

[18]Lindau, J. W., Kunz, R. F., Boger, D. A., Stinebring, D. R., and Gibeling, H. J., "High Reynolds Number, Unsteady, Multiphase CFD Modeling of Cavitating Flows," *Journal of Fluids Engineering*, Vol. 124, 2002, pp. 607–616.

[19]Chen, Y., "Compressible and Incompressible Flow Computations with a Pressure Based Method," Paper 89-0286, AIAA, 1989.

[20]Cheng, G. and Farmer, R., "CFD Spray Combustion Model for Liquid Rocket Engine Injector Analyses," Paper 2002-0785, AIAA, 2002.

[21]Pandya, M. J., Frink, N. T., and Chung, J. J., "Recent Enhancements to USM3D Unstructured Flow Solver for Unsteady Flows," Paper 2004-5201, AIAA, 2004.

[22]Kunz, R., Siebert, B., Cope, W., Foster, N., and S. Antal, S. E., "A Coupled Phasic Exchange Algorithm for Multi-Dimensional Four-Field Analysis of Heated Flows With Mass Transfer," *Computers & Fluids*, Vol. 27, No. 7, 1998.

[23]Koomullil, R. P. and Soni, B. K., "Flow Simulation Using Generalized Static and Dynamics Grids," *AIAA Journal*, Vol. 37, No. 12, 2000, pp. 1551–1557.

[24]Koomullil, R. P., Thompson, D. S., and Soni, B. K., "Iced Airfoil Simulation Using Generalized Grids," *The Journal of Applied Numerical Mathematics*, Vol. 46, No. 3, 2003, pp. 319–330.

[25]Wurtzler, K. E. and Morton, S. A., "Accurate Drag Prediction using *Cobalt*," Paper 2004-0395, AIAA, 2004.