Abstract

An effort is underway at Sandia National Laboratories to develop a library of algorithms to search for potential interactions between surfaces represented by analytic and discretized topological entities. This effort is also developing algorithms to determine forces due to these interactions for transient dynamics applications. This document describes the Application Programming Interface (API) for the ACME (Algorithms for Contact in a Multiphysics Environment) library.
### Table of Contents

Table of Contents .................................................... 5  
List of Figures .......................................................... 9  
List of Tables ........................................................... 11  

1. Introduction ......................................................... 13  
   1.1 Topology ......................................................... 14  
      1.1.1 Node_Blocks ............................................... 14  
      1.1.2 Face_Blocks ............................................. 15  
      1.1.3 Shells .................................................... 15  
      1.1.4 Element_Blocks ......................................... 16  
      1.1.5 Analytic_Surfaces ..................................... 17  
      1.1.6 Search_Data ............................................ 17  
   1.2 Search Algorithms ............................................. 21  
      1.2.1 Static 1-Configuration Search Algorithm ............... 22  
      1.2.2 Static 2-Configuration Search Algorithm ............... 22  
      1.2.3 Dynamic 2-Configuration Search Algorithm .............. 22  
      1.2.4 Dynamic Augmented 2-Configuration Search Algorithm ... 23  
   1.3 Interactions .................................................... 23  
      1.3.1 NodeFace_Interactions .................................. 23  
      1.3.2 NodeSurface_Interactions ............................... 25  
      1.3.3 NodeNode_Interactions ................................ 25  
      1.3.4 FaceFace_Interactions .................................. 26  
      1.3.5 FaceCoverage_Interactions .............................. 28  
      1.3.6 ElementElement_Interactions ............................ 29  
   1.4 Search Options ................................................ 29  
      1.4.1 Multiple Interactions at a Node ......................... 29  
      1.4.2 Normal Smoothing ...................................... 31  
   1.5 Gap Removal Enforcement ...................................... 33  
   1.6 Explicit Transient Dynamic Enforcement ....................... 34  
   1.7 Tied Kinematics Enforcement .................................. 35  
   1.8 Volume Transfer Enforcement .................................. 35  
   1.9 Multiple Point Constraint (MPC) Enforcement .................. 35  
   1.10 Errors .......................................................... 36  
   1.11 Plotting ...................................................... 37  
      1.11.1 Search Data Plot Variables ............................. 37  
      1.11.2 Enforcement Data Plot Variables ....................... 41  
   1.12 Restart Capabilities .......................................... 42  

2. Utility Functions ................................................... 43  
   2.1 Version Information ........................................... 43  
      2.1.1 Getting the Version ID .................................. 43  
      2.1.2 Getting the Version Date ................................. 44  
      2.1.3 Checking Compatibility with MPI ......................... 44
2.2 Errors .............................. 44
  2.2.1 Getting the Number of Errors ................. 45
  2.2.2 Extracting Error Messages ................... 45

2.3 Binary Stream Restart Functions ................. 46
  2.3.1 Getting the Binary Restart Size ............... 46
  2.3.2 Extracting the Binary Restart Data ............ 47
  2.3.3 Constructing Objects Upon Restart .......... 48

2.4 Variable-Based Restart Functions ............... 50
  2.4.1 Obtaining the Number of General Restart Variables 50
  2.4.2 Obtaining the Number of Nodal Restart Variables 51
  2.4.3 Obtaining the Number of Edge Restart Variables 52
  2.4.4 Obtaining the Number of Face Restart Variables 52
  2.4.5 Obtaining the Number of Element Restart Variables .. 53
  2.4.6 Extracting the General Restart Variables ..... 54
  2.4.7 Implanting the General Restart Variables .... 55
  2.4.8 Extracting the Nodal Restart Variables ....... 56
  2.4.9 Implanting the Nodal Restart Variables ..... 58
  2.4.10 Extracting the Edge Restart Variables ..... 60
  2.4.11 Implanting the Edge Restart Variables .... 61
  2.4.12 Extracting the Face Restart Variables ..... 63
  2.4.13 Implanting the Face Restart Variables .... 64
  2.4.14 Extracting the Element Restart Variables .. 66
  2.4.15 Implanting the Element Restart Variables .. 68
  2.4.16 Completing a Variable-Based Restart ....... 69

2.5 Creating an Exodus Plot File of the Search & Enforcement Data ...... 70

3. Search Functions .................................. 73
  3.1 Creating a ContactSearch Object ............... 73
  3.2 Updating a Search Object ...................... 77

3.3 Search_Data Array ................................ 81
  3.3.1 Checking the Search_Data Array Size .......... 81
  3.3.2 Setting Values in the Search_Data Array ... 82

3.4 Analytic_Surfaces ................................ 82
  3.4.1 Adding an Analytic_Surface .................... 82
  3.4.2 Setting the Analytic_Surface Configuration ... 84

3.5 Node_Block Data .................................. 84
  3.5.1 Setting the Node_Block Configuration ......... 84
  3.5.2 Setting the Node_Block Kinematic Constraints .. 85
  3.5.3 Setting the Node_Block Attributes ........... 86

3.6 Face_Block Data .................................. 87
  3.6.1 Setting the Face_Block Attributes ............ 87

3.7 Table Data ...................................... 87

3.8 Search Algorithms .................................. 88
  3.8.1 Setting the Search Option ................. 88
  3.8.2 Performing a Static 1-Configuration Search .. 89
  3.8.3 Performing a Static 2-Configuration Search .. 89
3.8.4 Performing a Dynamic 2-Configuration Search
3.8.5 Performing a Dynamic Augmented 2-Configuration Search
3.9 Interactions
3.9.1 Getting the Size of NodeFace_Interactions
3.9.2 Extracting NodeFace_Interactions
3.9.3 Getting the Size of NodeSurface_Interactions
3.9.4 Extracting NodeSurface_Interactions
3.9.5 Getting the Size of NodeNode_Interactions
3.9.6 Extracting NodeNode_INTERACTIONS
3.9.7 Getting the Size of FaceFace_Interactions
3.9.8 Extracting FaceFace_Interactions
3.9.9 Getting the Size of FaceCoverage_Interactions
3.9.10 Extracting FaceCoverage_Interactions
3.9.11 Getting the Size of ElementElement_Interactions
3.9.12 Extracting ElementElement_Interactions
3.9.13 Deleting Interactions
4. Gap Removal Enforcement Functions
4.1 Constructing a ContactGapRemoval Object
4.2 Computing the Gap Removal Displacements
4.3 Destroying a ContactGapRemoval Object
5. Explicit Transient Dynamic Enforcement Functions
5.1 Creating a ContactTDEnforcement Object
5.2 Defining Enforcement Models
5.3 Controlling the Algorithm
5.4 Specifying Symmetric Nodes
5.5 Computing the Contact Forces
5.6 Extracting Plot Variables
5.7 Destroying a ContactTDEnforcement Object
6. Tied Kinematics Enforcement Functions
6.1 Constructing a ContactTiedKinematics Object
6.2 Computing the ContactTiedKinematic Displacements
6.3 Destroying a ContactTiedKinematics Object
7. Volume Transfer
7.1 Constructing a Volume TransferObject
7.2 Computing the Transferred Element and Nodal Data
7.3 Destroying a ContactVolumeTransfer Object
8. MPC Enforcement
8.1 Constructing a ContactMPCs Object
8.2 Computing the Multiple Point Constraint (MPC) Equations
8.3 Getting the Number of MPC Equations
8.4 Getting the MPC Equations
8.5 Destroying a ContactVolumeTransfer Object
9. Example
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Problem Description</td>
<td>133</td>
</tr>
<tr>
<td>9.2 Constructing a ContactSearch Object</td>
<td>135</td>
</tr>
<tr>
<td>9.3 Adding an Analytic_Surface</td>
<td>136</td>
</tr>
<tr>
<td>9.4 Search Data</td>
<td>136</td>
</tr>
<tr>
<td>9.5 Setting the Search Options</td>
<td>137</td>
</tr>
<tr>
<td>9.6 Specifying Configurations</td>
<td>138</td>
</tr>
<tr>
<td>9.7 Performing the Search</td>
<td>139</td>
</tr>
<tr>
<td>9.8 Extracting Interactions</td>
<td>140</td>
</tr>
<tr>
<td>9.9 ExodusII Output</td>
<td>142</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1. Shell meshes which ACME cannot process due to the existence of a single element tab: a) all shell mesh b) mesh with both shells and hexes

Figure 2. Idealized 2D face with Search_Normal_Tolerance

Figure 3. Idealized 2D face with Search_Tangential_Tolerance

Figure 4. 3D NodeFace_Interactions

Figure 5. 3D NodeSurface_Interaction Data

Figure 6. 3D FaceFace_Interactions

Figure 7. 2D example of using PROJECTION_DIRECTION attribute to obtain user defined mortarising

Figure 8. Post-processing of FaceFace_Interactions to produce FaceCoverage_Interactions

Figure 9. Definition of Angle Between Faces

Figure 10. Interactions for Single vs. Multiple Interaction Definition

Figure 11. Normal Smoothing Across an Edge

Figure 12. Region of Normal Smoothing for a QuadFaceL4

Figure 13. Illustration of Normal Smoothing Resolution

Figure 14. Analytic Cylindrical Surfaces

Figure 15. Example impact problem (two rectangular bodies and an Analytic_Surface)

Figure 16. Face_Block Numbering for Example Problem

Figure 17. Surface Topology for Example Problem

Figure 18. ExodusII Output for Example Problem
List of Tables

Table 1. Search Data Layout For General Case ..........................................................20
Table 2. Search Data Layout without Elements or Faces. X’s indicate valid pairs. 20
Table 3. NodeFace_interaction Data for 3D ...............................................................24
Table 4. NodeSurface_interaction Data for 3D ..........................................................25
Table 5. NodeNode_interaction Data for 3D ..............................................................26
Table 6. FaceFace_interaction Data for 3D ...............................................................26
Table 7. FaceCoverage_interaction Data for 3D .........................................................29
Table 8. ElementElement_interaction Data for 3D ...................................................29
Table 9. Search Data Global Variables for ExodusII Output .....................................37
Table 10. Search Data Nodal Variables for ExodusII Output .....................................38
Table 11. Search Data Element Variables for ExodusII Output ..................................39
Table 12. Enforcement Data Nodal Variables for ExodusII Output ...........................41
Table 13. Search Data Element Variables for ExodusII Output .................................42
Table 14. C++ Data Description for Analytic_Surfaces ............................................83
Table 15. Transient Dynamic Enforcement Models and Data ....................................107
Table 16. Nodal TD Plot Variables ...........................................................................120
Table 17. Global TD variables ..................................................................................121
Table 18. Face_Blocks for Example Problem ............................................................134
Table 19. Current and Predicted Positions for Example Problem .............................139
Table 20. NodeFace_interactions for Example Problem .............................................141
Table 21. NodeSurface_interactions for Example Problem .......................................141
1. Introduction

Contact algorithms play an important role in many research and production codes that simulate various interfacial aspects of continuum solid and fluid mechanics and energy transport. Because of the difficult nature of contact in general and in order to concentrate and leverage development efforts, an effort is underway at Sandia National Laboratories to develop a library of algorithms to search for potential interactions between surfaces represented by finite element meshes and other topological entities. The requirements for such a library, along with other pertinent information, are documented at the following World Wide Web site:


This document describes the Application Programming Interface (API) for the ACME search and transient dynamics enforcement library. This introductory section gives an overview of the concepts and design of the ACME interface and outlines the building blocks that make up the data ACME needs from the host code and the data it returns to the host code. Section 2 describes various utility functions used to extract information about the package and its operation. Section 3 describes functions needed to access and utilize the search capabilities of ACME. Section 4 describes functions that can remove an initial overlap for a mesh prior to beginning a transient. Section 5 describes functions which ACME provides to enforce the results of the search in explicit transient dynamics simulations. Section 6 describes functions for a tied kinematic enforcement capability that allows nodes to be tied to faces and satisfy a no-relative-motion requirement. Section 7 describes functions that perform volume-weighted node and element variable transfers based on the element volume overlap returned by the search object. Section 8 describes functions which ACME provides to build multiple point constraint (MPC) equations which the host code may use to enforce node-face interactions returned by the search. Finally, Section 9 provides an example of how to use the API within a C++ application.

The basic philosophy of the ACME interface is to provide a separate function to support each activity. Efforts have been made to have the C++, C, and Fortran interfaces appear as similar as possible. It is important to note that all array indexes use the Fortran convention (i.e., indexes start with 1) and all floating-point data are double precision.

This release of the ACME library contains only a subset of the algorithms and functionality required to meet all the needs of the application codes. Currently, ACME supports three-dimensional (3D) topologies in serial and in parallel processing modes. No multi-state support is provided in this release (i.e., ACME has no ability to revert to previous states). ACME only supports conventional nodes (smooth particle hydrodynamics nodes are not yet supported) and a limited set of face types (a linear 4-node quadrilateral, a quadratic 8-node quadrilateral, a linear 3-node triangle, and a quadratic 6-node triangle) in this release. This release expands the support of shell contact made available in the previous release, where the shell surfaces are lofted by the shell thickness. Shell contact in this version of ACME is subject to the following restrictions: shells can not be involved with dynamic load balancing, shell edge to shell edge contact is not considered, and restarts...
Introduction

with shells are only supported through the variable restart interface. This release also only supports element-element searches using 8-node hex elements, and at least one of the elements in the search must be a Cartesian hex (a hexahedral element with edges aligned with the global x-, y-, or z-axis). Additional algorithms and functionality will be added in subsequent releases.

1.1 Topology

The topology for ACME is determined by the host code. The first step in using the library is for the host code to provide to ACME a topological description of the surfaces to be checked for interactions, or the elements to be checked for overlap. Currently, the topology consists of collections of nodes, faces, elements, and analytic surfaces. Nodes, faces, and elements are supplied to ACME in groups called blocks. A Node_Block may contain only one type of node. A Face_Block may contain only one type of face and all faces will have the same Entity_Key. (Entity_Keys are used to extract user-specified parameters from the Search_Data array for pairs of interacting topological entities, as explained in Section 1.1.6). An Element_Block may contain only one type of element, and all elements will have the same Entity_Key. Currently, a single search object should not contain both Face_Blocks and Element_Blocks. Providing the full functionality required of ACME will necessitate adding Edge_Blocks, which will be analogous to Face_Blocks (see the description in Section 1.1.2). Also, the full functionality required of ACME will necessitate adding multiple states; for this release of ACME, only a single state (with one or two configurations) is supported.

1.1.1 Node_Blocks

A Node_Block is a collection of nodes of the same type. Currently, two types of nodes are supported in ACME:

- **NODE**: A traditional node that is connected to an element or face. It has a position and an optional projection direction attribute (for face/face search; see Section 1.3.4).
- **POINT**: A node that is not connected to an element or face (i.e., a particle, a gauss point, etc.). It has a position, an optional projection direction attribute and an optional radius.

Node_Blocks can be used for three purposes. Node_Block 1 is for nodes that are connected to elements or faces and uses the entity_key (see Section 1.1.6) inherited from the elements or faces. Nodes that are connected to elements or faces but need their own entity_key (i.e., they should not inherit the data key from the faces) can be placed in additional Node_Blocks. An example would be a single node of a mesh that needs to be spot-welded to another surface. This single node would be placed in its own Node_Block and then the Search_Data and Enforcement_Data would be set up for this entity_key to define a spot-weld. Additional Node_Blocks can be used for nodes not connected to faces in the ContactSearch topology. These additional Node_Blocks can be used for SPH particles or for finding the Gauss point locations on the other side of an interface.

Each Node_Block is assigned an integer identifier (ID). This ID corresponds to the order in which the blocks were specified, using the Fortran numbering convention (i.e., the first block has an ID of 1, the second block has an ID of 2, etc.). This ID is used in specifying
configurations for Node Blocks, and for returning NodeFace Interactions, NodeSurface Interactions, and NodeNode Interactions discussed later in Section 1.3.

1.1.2 Face Blocks

A Face Block is a collection of faces of the same type that have the same Entity Key. (Entity Keys are used to extract user-specified parameters from the Search Data array, as explained in Section 1.1.6.) Currently, ACME supports a linear 4-node quadrilateral face called QUADFACEL4, a quadratic 8-node quadrilateral face called QUADFACEQ8, a linear 3-node triangular face called TRIFACEL3, and a quadratic 6-node triangular face called TRIFACEQ6. It also supports shell versions of the linear 4-node quadrilateral and 3-node triangular faces, called SHELLQUADFACEL4 and SHELLTRIFACEL3, respectively. Other face types will be added as needed. These are provided in an enumeration in the ContactSearch header file:

```c
enum ContactFace_Type {
    QUADFACEL4 = 1,
    QUADFACEQ8,
    TRIFACEL3,
    TRIFACEQ6,
    SHELLQUADFACEL4,
    SHELLTRIFACEL3
}
```

Each Face Block is assigned an ID. This ID corresponds to the order in which the blocks were specified. This ID is used in returning NodeFace Interactions.

1.1.3 Shells

In order to compute contact on shell elements, ACME creates an internal representation of the top and bottom faces in which they are separated by a distance equal to the shell thickness. Contact interactions are then computed on this lofted geometry, and any forces computed by enforcement algorithms are mapped back to the original shell nodes. This permits a more realistic representation of contact on shells, including shell-shell contact. By default, shell faces are lofted halfway from their original configuration, such that the original shell location represents the mid-surface of the shell. To simplify meshing, ACME does permit the definition of a lofting factor for each shell face. A lofting factor of 0.0 for a face causes the face to not be lofted, while a factor of 1.0 causes a face to be lofted by the full thickness. The lofting factors for the top and bottom faces of a shell should add up to 1.0. The lofting factor and shell thickness are input to ACME through the Set Face Block Attributes command.

This approach for shell contact can produce a few cases where it is difficult to determine a proper lofted topology, thus care must be taken when creating a shell mesh. For instance, ACME currently cannot process a shell mesh with a single shell “tab”-- a single shell connected to the middle of a collection of shell or hex elements. Sample cases are illustrated in Figure 1. Using more than one shell element for the tab avoids these problems.
Figure 1 Shell meshes which ACME cannot process due to the existence of a single element tab: a) all shell mesh b) mesh with both shells and hexes

Another problem occurs when two shells lie on top of each other and are connected together on one edge, thereby making it difficult to determine which shell is above the other. In this case, the lofting factors are used to resolve the ambiguity if possible. If the two faces have the same normal, then giving one face a lofting factor of 1.0 and the other a lofting factor of 0.0 permits a proper shell topology to be constructed. Note that values other than 1.0 and 0.0 would not permit proper resolution of the faces.

ACME tries to determine the location of nodes at sharp corners between shells by projecting the positions using the thicknesses and normals of the adjacent faces. However, this projection can break down when the angle between two faces falls below ten degrees. In such cases, the projection distance is limited to five times the maximum shell thickness of the adjacent faces.

All nodes which are attached only to shell faces are lofted according to the provided thickness and lofting factor. Shell nodes that are also attached to non-shell faces are not lofted, nor are shells at the corners of tab-like geometries.

Faces are identified as shell faces through the face types SHELLQUADFACE14 and SHELLTRIFACE13. Both the top and bottom faces of each shell must be included in the face blocks passed to ACME, though the two faces are not required to be in the same block.

ACME does not currently support the mixing of face blocks and element blocks in a single search object.

1.1.4 Element_Blocks

An Element_Block is a collection of elements of the same type that have the same Entity_Key. (Entity_Keys are used to extract user-specified parameters from the Search_Data array, as explained in Section 1.1.6.) Currently, two forms of an eight-node hex are supported: CARTESIANHEXELEMENTL8, which has each face aligned with a
Introduction

Cartesian plane, and HEXELEMENTL8, which is an arbitrary eight-node hex. Other element types will be added as needed. These are provided in an enumeration in the Contact-Search header file:

```c
enum ContactElement_Type {
    CARTESIANHEXELEMENTL8 = 1,
    HEXELEMENTL8
}
```

Each Element_Block is assigned an ID. This ID corresponds to the order in which the blocks were specified, using the Fortran numbering convention. This ID is used in returning ElementElement_Interactions.

ACME does not currently support the mixing of face blocks and element blocks in a single search object. Also, ACME currently requires that at least one of the element blocks involved with every element-element search be a Cartesian hex. Element-element searches between two HEXELEMENTL8 Element_Blocks is not permitted.

1.1.5 Analytic_Surfaces

In many instances, it is advantageous to search for interactions against rigid analytic surfaces (referred to as Analytic_Surfaces throughout this document) rather than mesh such a surface. Examples include a tire rolling on a flat road or dropping a shipping container on a post. Currently, ACME is designed to handle only geometric Analytic_Surfaces (e.g., planes, cylinders, etc.), and for now, only planar, spherical and cylindrical Analytic_Surfaces are supported. Other geometric Analytic_Surfaces will be added in the future as needed. Eventually, Analytic_Surfaces defined by Non-Uniform Rational B-Splines (NURBS) will be supported. The ACME API will need to be extended to support Analytic_Surfaces defined by NURBS. Analytic_Surfaces currently can interact only with nodes, not faces or elements. Analytic_Surfaces should not be specified for search objects that only contain elements.

Analytic_Surfaces, if any, are provided by the host code to ACME after the Node_Blocks and Face_Blocks have been specified. Analytic_Surfaces are given an ID that corresponds to the total number of Face_Blocks plus the order in which the Analytic_Surface was added (e.g., if three Face_Blocks exist in the topology, the ID of the first Analytic_Surface is 4, the ID of the second Analytic_Surface is 5, etc.). This ID is used in returning NodeSurface_Interactions.

1.1.6 Search_Data

The Search_Data array contains data that describe how the various topological entities are allowed to interact. The host code may specify, for example, that only nodes on surface A interact with faces on surface B, or that only nodes on surface B interact with faces on surface A, or both. The Search_Data array is the only place where such user-specified data are kept.

Currently the Search_Data array holds only three parameters for each Entity_Key pair. The first parameter is a status flag indicating what type of interactions should be defined
for this pair. Five values are currently permitted, provided in an enumeration in the ContactSearch header file:

```cpp
enum Search_Interaction_Type{
    NO_INTERACTION = 0,
    SLIDING_INTERACTION,
    TIED_INTERACTION,
    COVERAGE_INTERACTION,
    GENERIC_INTERACTION);
```

NO_INTERACTION (a value of 0) requests that no interactions be defined for this pair of entities. SLIDING_INTERACTION (a value of 1) requests that ACME search for new node/face or node/surface interactions between entities each time a search is executed. TIED_INTERACTION (a value of 2) requests that a node/face or node/surface interaction between entities persist over multiple time steps, thus allowing it to be used for mesh tying. COVERAGE_INTERACTION (a value of 3) requests that ACME search for new face/coverage interactions between entities each time a search is executed. In addition, it requires that a face/face search also be performed. GENERIC_INTERACTION (a value of 4) requests that ACME search for node/node, face/face, or element/element interactions between entities each time a search is executed.

The second parameter in the Search_Data array is the Search_Normal_Tolerance, which is used to determine whether the entity pair should interact, based on the separation between the entities (see Figure 2). Note that the Search_Normal_Tolerance is an absolute distance, so it is dependent on the units of the problem. The third parameter is the Search_Tangential_Tolerance, also used to determine whether the entity pair should interact, but taking into account distances tangential to a face, rather than normal to it. Element-Element interactions do not use either tolerance, so they should be set to zero.

Every face, element, and node is assigned an Entity_Key to allow retrieval of data from the Search_Data array. For faces, the Entity_Key corresponds to the Face_Block ID. For elements, the Entity_Key corresponds to the Element_Block ID. A node in the first node block inherits its Entity_Key from the first face that contains it. This is a limitation of the current implementation, since a node can be connected to two or more faces that are in different Face_Blocks. The Entity_Key for an analytic surface is the Analytic_Surface ID, while the Entity_Key of a node block other than the first node block is the number of Face_Blocks plus the number of Analytic_Surfaces plus the Node_Block ID minus one.

The Search_Data array is a three-dimensional Fortran array with the following size

```fortran
dimension search_data(3,num_entity_keys,num_entity_keys)
```

The first index represents one of the three parameters described previously for each entity pair, currently a node-face, node-Analytic_Surface, node-node, face-face, or element-element pair. The second index indicates the Entity_Key for the slave entity (node, face, element, or analytic_Surface) in an interaction, and the third index indicates the Entity_Key for the master entity (node, face, element, or Analytic_Surface) in an interaction. Interactions are stored on the slave entity. In the most general case, the layout of the search data is
shown in Table 1 and is supplied to ACME in row major format. If the topology does not contain any elements or faces, then node block 1 is added to the search data table. In this case, the layout of the search data is shown in Table 2. Entry (1, 1) is the upper left corner of the table and entry (num_entity_keys, num_entity_keys) is the lower right corner of the table.
Table 1: Search Data Layout For General Case

<table>
<thead>
<tr>
<th>Master Entities</th>
<th>Slave Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Element Block 1</td>
</tr>
<tr>
<td>Element Block 1</td>
<td>X X X</td>
</tr>
<tr>
<td>.</td>
<td>X X X</td>
</tr>
<tr>
<td>Element Block N</td>
<td>X X X</td>
</tr>
<tr>
<td>Face Block 1</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>.</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>Face Block N</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>Node Block 2</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>.</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>Node Block N</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>Analytic Surface 1</td>
<td>X X X</td>
</tr>
<tr>
<td>.</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>Analytic Surface N</td>
<td>X X X</td>
</tr>
</tbody>
</table>

Table 2: Search Data Layout without Elements or Faces. X’s indicate valid pairs.

<table>
<thead>
<tr>
<th>Master Entities</th>
<th>Slave Entities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Node Block 1</td>
</tr>
<tr>
<td>Node Block 1</td>
<td>X X X</td>
</tr>
<tr>
<td>.</td>
<td>X X X</td>
</tr>
<tr>
<td>Node Block N</td>
<td>X X X</td>
</tr>
<tr>
<td>Analytic Surface 1</td>
<td>X X X</td>
</tr>
<tr>
<td>.</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>Analytic Surface N</td>
<td>X X X</td>
</tr>
</tbody>
</table>
1.2 Search Algorithms

ACME provides four different algorithms for determining interactions. The data types returned in the interactions are the same for each type of search. The host code may use different types of search algorithms during an analysis (e.g., a static 1-configuration search to determine overlaps in the mesh before starting the analysis and then a dynamic search once time stepping begins in a transient dynamics code).

As an aid to understanding the differences between the search algorithms, consider the idealized 2D face of Figure 2. In this idealized example, the subtleties of what happens at the edge of a face are ignored. Any entity that is outside the face, where “outside” is defined by the outward unit normal $\mathbf{n}$, is not penetrating and has a positive Gap. Any entity that is on the face (i.e., a zero Gap) or inside the face (i.e., a negative Gap) is considered to be penetrating. The host code controls the Search_Normal_Tolerance as part of the Search_Data array (see Section 1.1.6). The Motion_Tolerance accounts for movement of the node if two configurations are used, and is computed by ACME.

A separate tolerance, Search_Tangential_Tolerance, is used to specify the behavior of the search algorithms along the edge of a face. As shown in Figure 3, a NodeFace_Interaction will be defined for any node that is outside the face tangentially but within the Search_Tangential_Tolerance. The host code controls the Search_Tangential_Tolerance as part of the Search_Data array (see Section 1.1.6).
Introduction

1.2.1 Static 1-Configuration Search Algorithm

The static 1-configuration search algorithm uses only one configuration for the topology. The interactions are determined using only a closest point projection algorithm. Interactions are defined only for entities that are within the Search_Normal_Tolerance (either negative or positive Gap) and the Search_Tangential_Tolerance. The Motion_Tolerance is implied to be zero. This search must be used for node-node, face-face and element-element interactions.

1.2.2 Static 2-Configuration Search Algorithm

The static 2-configuration search algorithm requires two configurations (Current and Predicted) for the topology. This search algorithm uses closest point projection on the predicted configuration but it has the added information of the movement of the topology. The motion tolerance implied by the two configurations is used along with the Search_Data to determine what interactions are physically realistic. Specifically, any node that has a positive Gap within the Search_Normal_Tolerance or any node that has a negative Gap within the Search_Normal_Tolerance plus the motion tolerance will result in an interaction being defined, provided that the node’s projection falls within the face boundary as extended laterally by the Search_Tangential_Tolerance. This search is not supported for face-face or element-element searches; only the static 1-configuration search will operate for these cases.

1.2.3 Dynamic 2-Configuration Search Algorithm

The dynamic 2-configuration search algorithm also requires two configurations (Current and Predicted) for the topology. A dynamic intersection algorithm based on linear interpolation of the motion is used to initiate interaction if the current and predicted Gaps are on opposing sides of the face (e.g., the current configuration has a positive Gap and the predicted configuration has a negative Gap). A closest point projection algorithm is used for subsequent interaction definition and to initiate interaction if the current and predicted Gaps are on the same side of the face. In these cases, interactions are defined by the same
Introduction

criteria as in the static 2-configuration search algorithm (see Figure 2). This search is not supported for face-face or element-element searches; only the static 1-configuration search will operate for these cases.

1.2.4 Dynamic Augmented 2-Configuration Search Algorithm

The dynamic augmented 2-configuration search algorithm is a more accurate implementation of the dynamic 2-configuration search algorithm. This search can only be used in conjunction with the ContactTDEnforcement enforcement algorithm. It uses information from the enforcement on the previous step to compute an augmented configuration that yields more accurate interactions. This search is not supported for face-face or element-element searches; only the static 1-configuration search will operate for these cases.

1.3 Interactions

The output of ACME following a search is a collection of interactions based on the topology, configuration(s), Search_Data, and search algorithm. Currently, six types of interactions are supported: NodeFace_Interactions, NodeSurface_Interactions, NodeNode_Interactions, FaceFace_Interactions, FaceCoverage_Interactions, and ElementElement_Interactions. ACME does not determine the best interaction between these types (i.e., ACME does not compete a NodeFace_Interaction against a NodeSurface_Interaction when the same node is involved; both are returned to the host code). Other interaction types (e.g., EdgeFace_Interaction) will be added in the future. The FaceFace_Interactions, FaceCoverage_Interactions, and ElementElement_Interactions are only available in the static 1-configuration search.

1.3.1 NodeFace_Interactions

A NodeFace_Interaction is returned as a set of data to the host code: a node (indicated by the Node_Block ID and the index in that Node_Block), a face (indicated by the Face_Block ID and the index in that Face_Block) and data describing the interaction. Consider the examples shown in Figure 4. The left-hand-side of Figure 4 illustrates an interaction defined using the dynamic intersection algorithm. In the figure $A_1$, $A_2$, $A_3$ and $A_T$ are the areas of the triangles 23X, 31X, 12X and 123 respectively. Here, a node, lightly shaded in its current configuration and black in its predicted configuration, intersects a TRIFACEL3 at X in an intermediate configuration denoted with white nodes. The motion of the node is represented by the vector $v_x$. Also shown are the data that are returned for this interaction. Specifically, the pushback direction is given by the vector from the penetrating node’s predicted position to the position of the contact point convected into the predicted configuration. In the second diagram, the contact point X, determined by closest point projection for a single configuration, is shown in local coordinate space for a QUADFACEL4. Table 3 gives the Fortran layout of how the data are returned. It should be noted that only two local coordinates are returned. For triangular faces, the third local coordinate is simply unity minus the sum of the other two local coordinates.

Currently, ACME can not return NodeFace_Interactions for shell faces (SHELLQUADFACEL4 and SHELLTRIFACEL3).
Local Coordinates:
(of contact point \( \mathbf{X} \))

\[
\begin{align*}
\xi_1 &= \frac{A_1}{A_T} \\
\xi_2 &= \frac{A_2}{A_T} \\
\xi_3 &= \frac{A_3}{A_T}
\end{align*}
\]

Gap: \( g \) (not returned)

Unit Pushback Vector: \( \mathbf{\dot{p}} \) (not shown)

Unit Surface Normal: \( \mathbf{\hat{n}} \) (not shown)

Algorithm: Dynamic Intersection

Local Coordinates: \( \xi_1 = \xi_1 \)
(of contact point \( \mathbf{X} \))

\[
\begin{align*}
\xi_1 &= \xi_1 \\
\xi_2 &= \eta_1
\end{align*}
\]

Gap: \( g \) (not shown)

Unit Pushback Vector: \( \mathbf{\dot{p}} \) (not shown)

Unit Surface Normal: \( \mathbf{\hat{n}} \) (not shown)

Algorithm: Closest Point Projection (1-Configuration)

---

**Figure 4 3D NodeFace_Interactions**

**Table 3 NodeFace_Interaction Data for 3D**

<table>
<thead>
<tr>
<th>Location (Fortran Indexing)</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Local Coordinate 1 (( \xi_1 ) for Q4 or Q8, ( \xi_1 ) for T3 or T6)</td>
</tr>
<tr>
<td>2</td>
<td>Local Coordinate 2 (( \eta_1 ) for Q4 or Q8, ( \xi_2 ) for T3 or T6)</td>
</tr>
<tr>
<td>3</td>
<td>Gap</td>
</tr>
<tr>
<td>4-6</td>
<td>Unit Pushback Vector (( x, y &amp; z ) components)</td>
</tr>
<tr>
<td>7-9</td>
<td>Unit Surface Normal (( x, y &amp; z ) components)</td>
</tr>
<tr>
<td>10</td>
<td>Algorithm Used to Define Interaction {1=Closest Point Projection (1 Configuration), 2=Closest Point Projection (2 Configuration), 3=Dynamic Intersection (2 Configuration)}</td>
</tr>
</tbody>
</table>
1.3.2 NodeSurface_Interaction

A NodeSurface_Interaction is returned as a set of data to the host code: a node (indicated by the Node_Block ID and the index in that Node_Block), an Analytic_Surface (indicated by its ID) and the data describing the interaction. Figure 5 shows the interaction data that are returned to the host code for each interaction. Table 4 gives the layout of the data for a NodeSurface_Interaction.

For this release of ACME, NodeSurface_Interaction are determined using a closest point projection algorithm. Therefore, only one configuration is required for the Analytic_Surfaces. The configuration used for the nodes is based on the current configuration for a 1-configuration static search and the predicted configuration for the 2-configuration static search or the dynamic searches. This limitation will be removed in a future release.

<table>
<thead>
<tr>
<th>Location (Fortran Indexing)</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>Interaction Point (x, y &amp; z coordinates)</td>
</tr>
<tr>
<td>4</td>
<td>Gap</td>
</tr>
<tr>
<td>5-7</td>
<td>Unit Surface Normal (x, y &amp; z components)</td>
</tr>
</tbody>
</table>

Figure 5 3D NodeSurface_Interaction Data

1.3.3 NodeNode_Interaction

A NodeNode_Interaction is returned as a set of data to the host code: a slave node (indicated by the Node_Block ID and the index in that Node_Block), a master node (indicated
Introduction

by the Node_Block ID and the index in that Node_Block), and data describing the interaction. Table 5 gives the layout of the data for a NodeNode_Interaction.

<table>
<thead>
<tr>
<th>Location (Fortran Indexing)</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Distance between nodes</td>
</tr>
</tbody>
</table>

### 1.3.4 FaceFace_Interaction

A FaceFace_Interaction is returned as a set of data to the host code: a slave face (indicated by the Face_Block ID and the index in that Face_Block), a master face (indicated by the Face_Block ID and the index in that Face_Block), and data describing the interaction. This interaction is only valid for faces of type QUADFACE_L4 and TRIFACE_L3. Consider the example shown in Figure 6. Here, two faces are in proximity and the FaceFace_Interaction needs to be determined. The master face is transformed into a master volume by projecting the nodes in the +/− projection direction by the Search_Normal_Tolerance. By default, the projection direction for each node on the master face is the normal at that node (with or without smoothing). Optionally, the projection direction can be user-specified as a node attribute. This permits “mortaring” to be performed under control (see Figure 7). Once the master face has been converted to a master volume, the intersection between the slave face and master volume is computed. This intersection is described with a closed polygon having N sides, E_n, and nodes, P_n. The points on the slave face that define the polygon are stored in the local coordinates of the slave face. These points are also computed as local coordinates of the master volume and projected onto the master face by setting ξ_3 = 0 and then stored in the local coordinates of the master face. The resulting convex polygon can be triangularized by the host by calculating the centroid of the polygon, P_{N+1}, and connecting it to each node. Two additional arrays (of length N) are defined that indicate with which edge, if any, of the master or slave face an edge of the polygon is coincident. Table 6 gives the Fortran layout of how the data are returned. It should be noted that only two local coordinates are returned. For triangular faces, the third local coordinate is simply unity minus the sum of the other two local coordinates.

<table>
<thead>
<tr>
<th>Location (Fortran Indexing)</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>number of vertices (and edges), N</td>
</tr>
<tr>
<td>1+n</td>
<td>slave edge flag for edge n=1,...,N</td>
</tr>
<tr>
<td>(N+1)+n</td>
<td>master edge flag for edge n=1,...,N</td>
</tr>
</tbody>
</table>
Number of edges = 4
Master edge flag array = [0, 0, 0, 0]
Slave edge flag array = [0, 0, 3, 4]
Polygon nodes = P1, P2, P3, and P4
Polygon centroid = P5

Figure 6 3D FaceFace_Interactions
A FaceCoverage_Interaction is returned as a set of data to the host code: a face (indicated by the Face_Block ID and the index in that Face_Block) and data describing the interaction. Each FaceCoverage_Interaction is a closed polygon that describes an exposed (i.e., uncovered) portion of a face. The FaceCoverage_Interaction is computed by post-process-
In the FaceFace_Interaction for each face. A directed edge graph is constructed using the edges of the polygon from all the FaceFace_Interaction associated with each face and any portions of each face edge that are not part of a FaceFace_Interaction polygon. Closed polygons are then extracted from the directed edge graph to produce one or more FaceCoverage_Interaction for each face, as shown in Figure 8. Table 7 gives the Fortran layout of how the data are returned. For triangular faces, the third local coordinate is simply unity minus the sum of the other two local coordinates.

<table>
<thead>
<tr>
<th>Location (Fortran Indexing)</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of vertices (and edges), N</td>
</tr>
<tr>
<td>2*(n-1)+2</td>
<td>Local Coordinate 1 for polygon node Pn, n=1,.....,N</td>
</tr>
<tr>
<td>2*(n-1)+3</td>
<td>Local Coordinate 2 for polygon node Pn, n=1,.....,N</td>
</tr>
</tbody>
</table>

### 1.3.6 ElementElement_Interaction

An ElementElement_Interaction is returned as a set of data to the host code: a slave element (indicated by the Element_Block ID and the index in that Element_Block), a master element (indicated by the Element_Block ID and the index in that Element_Block), and data describing the interaction. This interaction is only valid for elements of type CARTEISIANHEXELEMENTL8 and HEXELEMENTL8. Furthermore, at least one of the elements in every interaction must be a CARTEISIANHEXELEMENTL8. Table 8 gives the Fortran layout of how the data are returned. Currently, the only information returned for these interactions is the volume of the overlap.

<table>
<thead>
<tr>
<th>Location (Fortran Indexing)</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Volume of overlap between elements</td>
</tr>
</tbody>
</table>

### 1.4 Search Options

#### 1.4.1 Multiple Interactions at a Node

By default, ACME defines only one interaction at a node. If potential interactions with more than one face are detected, ACME will return only one interaction (the best one, according to the algorithm used for competition between two interactions) to the host code. However, to get better behavior at a true corner of a body, multiple interactions with the faces surrounding the corner should be considered. Therefore, if desired, ACME can define multiple interactions at a node. When this feature is activated, the host code must
specify an angle (in degrees) called SHARP_NON_SHARP_ANGLE. If the angle between connected faces (computed as the angle between the normals to the faces, as shown in Figure 9) is greater than SHARP_NON_SHARP_ANGLE, then an interaction will be defined for each face, instead of competition between the two to define one interaction. If the multiple interactions feature is not active, interactions with only one of two disconnected faces will be returned (see Figure 10). Interactions with disconnected faces will be returned to the host code regardless of the angle.
1.4.2 Normal Smoothing

As previously noted, a NodeFace_Interaction consists of a contact point, a normal gap, a pushback direction, and a normal direction. The normal direction is an approximation of the normal to the surface at the contact point, which by default is simply the normal to the face. In some cases, however, it is necessary to have a continually varying normal without abrupt changes (e.g., when transitioning across an edge). The normal smoothing capability computes, if appropriate, a “smoothed” normal that varies continuously as a node transitions between faces. Smoothing occurs if the contact point is within a user-specified distance to the edge and if the included angle between the faces is less than the SHARP_NON_SHARP_ANGLE (see Figure 11). The contact point, normal gap, and pushback direction are not modified by normal smoothing.
When activating this feature, the host code must specify a SHARP_NON_SHARP_ANGLE (in degrees), a normal smoothing distance, and a RESOLUTION_METHOD for cases when a unique solution cannot be determined. If the angle between two faces is greater than the SHARP_NON_SHARP_ANGLE, then the edge is considered SHARP and no smoothing will be done to the normal. The angle specified for normal smoothing must match the angle specified for multiple interactions if that capability is active.

The normal smoothing distance (SD) specifies the region over which normal smoothing occurs (see Figure 12). This distance is in isoparametric coordinates, so its value ranges from 0 to 1 (in theory), but for practical purposes, 0.5 is an upper bound.

For the case when a unique solution does not exist for a smoothed normal, two resolution methods are provided: USE_NODE_NORMAL and USE_EDGE_BASED_NORMAL. To illustrate the differences between these two approaches, consider Figure 13. This example consists of five faces in the configuration shown, and uses a SHARP_NON_SHARP_ANGLE of 30 degrees. The angles between faces 1 and 5 and between faces 3 and 4 are greater than the SHARP_NON_SHARP_ANGLE, so the smoothing algorithm should not smooth between these faces. Smoothing is done between faces 1 and 2 and between faces 2 and 3, because the corresponding angles are less than 30 degrees. For points approaching the shared intersection of faces 1, 2, and 3, however, the two options ACME provides for determining the smoothed normal deliver different results. The USE_NODE_NORMAL option defines the normal at the intersection point to be the
node normal and thus provides a continuously smooth normal in the region near the point. The problem with this approach in this particular case is that the node normal also includes the effects of faces 4 and 5, and thus effectively provides smoothing over the boundary between faces 1 and 5. Alternatively, the USE_EDGE_BASED_NORMAL option only considers smoothing between a pair of faces. This approach ensures that no smoothing occurs between faces 1 and 5, but it unfortunately can provide a different normal if we approach the intersection point from face 1 than if we approach the point from face 3. Therefore, the smoothed normal at the intersection point can be discontinuous, which can cause numerical problems in some applications. This feature will be addressed further as host codes gain experience on what approaches provide the best behavior.

![Figure 13 Illustration of Normal Smoothing Resolution](image)

### 1.5 Gap Removal Enforcement

An optional gap removal enforcement is also included in ACME. Initial gaps often occur in meshes where curved geometries are discretized using varying mesh densities. The discretization error causes nodes from one (or more) surfaces to penetrate other surfaces. This initial gap can cause problems in explicit transient dynamic simulations (as well as other physics simulations) if the initial gap is large enough to cause interactions to be missed or if the initial gap is enforced on the first step, causing a large force. An effective method for avoiding these problems is to search for initial gaps and remove them in a strain-free manner (i.e., the initial topology is modified to remove the initial gaps). The enforcement object will compute the displacement correction needed to remove these initial gaps. Although it is not possible to have all nodes exactly on the faces of the other surface for curved geometries (it is an overconstrained problem), the gap removal enforcement seeks to satisfy the inequality that all gaps are non-negative with a minimum normal gap.

The gap removal enforcement should be used after performing a static 1-configuration search. The typical sequence for an explicit transient dynamic simulation would be:
1) Set the Search_Data array appropriate for an initial gap search.
2) Perform a static 1-configuration search.
3) Call ContactGapRemoval::Compute_Gap_Removal.
4) Apply the displacement correction from step 3 to the topology.
5) Initialization (compute volume, mass, etc. using the modified topology).

### 1.6 Explicit Transient Dynamic Enforcement

An optional explicit transient dynamic enforcement capability is included in ACME. The algorithm was written assuming that the host code is integrating the equations of motion using a central difference time integrator. The topology, interactions, and configurations are taken directly from a ContactSearch object (i.e., the enforcement is dependent on a ContactSearch object). This capability takes as input the nodal masses from the host and returns the nodal forces that need to be applied.

The explicit transient dynamic enforcement can only be used in conjunction with the dynamic 2-configuration or dynamic augmented 2-configuration search methods. Following gap removal (if desired) and initialization, the continuation of the typical sequence for an explicit transient dynamic simulation would be:

6) Set the Search_Data array appropriate for the analysis.
7) Time Step using
   a) a dynamic or dynamic augmented 2-configuration search.
   b) a ContactTDEnforcement enforcement.

Two parameters, KINEMATIC_PARTITION and FRICTION_MODEL_ID, must be supplied by the host code for each possible entity pair. The KINEMATIC_PARTITION pertains to the fraction of total momentum each contacting surface will absorb. For example, if surface 1 contacts surface 2 and the kinematic partition for surface 1 is k, then the kinematic partition for surface 2 with respect to surface 1 is 1-k. Furthermore, if k is 1, then surface 1 acts as a “slave” to surface 2. The kinematic partition can either be a constant user-specified value (between 0 and 1) or the code will compute it for each interaction using the relations

\[
\beta_1 = \frac{\rho_1 c_1}{\rho_1 c_1 + \rho_2 c_2}
\]

(1)

\[
\beta_2 = \frac{\rho_2 c_2}{\rho_1 c_1 + \rho_2 c_2}
\]

(2)

where \( \beta_i \) is the kinematic partition, \( \rho_i \) is the density, and \( c_i \) is the wavespeed of the material for surface \( i \), \( (i=1,2) \).
The FRICTION_MODEL_ID refers to the particular friction model requested by the host code. There are currently thirteen friction models available: frictionless, constant Coulomb friction, tied, pressure dependent friction, velocity dependent friction, pressure and velocity dependent friction, adhesion, spot weld, point weld, area weld, threaded, junction, and cohesive zone. For NodeFace_Interactions and NodeSurface_Interactions, tied, frictionless, and frictional conditions can be enforced at a node. If these constraints are independent (e.g., three separate contact constraints at the corner of a block), the enforcement delivers the expected result. For conflicting constraints, a least-squares methodology is employed to resolve the forces required to effect the simultaneous interacting contact conditions.

1.7 Tied Kinematics Enforcement

ACME provides an optional enforcement capability to compute the positions of nodes so that no relative motion occurs between a master face and a slave node. This enforcement only operates for a pure master-slave relationship (i.e., it does not work for symmetric cases). This enforcement was developed to serve as a remesh boundary condition for ALEGRA’s SHISM capability but may have additional uses. It should be paired with a static 1-configuration search, which only needs to be done once. The enforcement can then be called repeatedly to reposition the slave nodes using the interactions defined by the search, with the slave node position interpolated using the local coordinates and the positions of the nodes on the face.

1.8 Volume Transfer Enforcement

An optional volume transfer enforcement capability is included in ACME. This enforcement provides facilities to transfer nodal and element data between two meshes. This data transfer is performed using linear interpolation between the master (donor) to slave (receiver) mesh for nodal variables. Element variables data is mapped between the two meshes using volume fraction weighting. A call to the volume transfer routine (see Section 7.) provides both the transferred nodal and element variables along with the volume fraction on the receiving mesh filled by the donor mesh. This enforcement object requires that at least one of the meshes (either donor, receiver or both) consist of cartesian HEX8 elements.

1.9 Multiple Point Constraint (MPC) Enforcement

An optional multiple point constraint (MPC) equation capability is included in ACME. This enforcement provides facilities to build MPC equations for pure master-slave enforcement of node-on-face interactions. The topology, interactions, and configurations are taken directly from a ContactSearch object (i.e., the enforcement is dependent on a ContactSearch object). The MPC algorithm should be paired with a static 1-configuration search, which only needs to be called once. Subsequent calls to the appropriate Contact-MPC functions (see Section 8.) build the constraint equations, get the number of equations and return the MPC data to the host code.
Calls to get the MPC equations return a list of the equations in terms of slave node ID, the
ID of the master face and the IDs of the nodes attached to the face and their corresponding
coefficients (cf. Section 8.). The scheme used for numbering the node and face IDs is user
selectable being either the global ID supplied by the user when the search object was con-
structed or ACME’s internal ID (cf. Section 8.). In parallel operation ACME’s MPC en-
forcement will return to the calling processor all MPC equations for which the processor
owns the face, the slave node, or has a ghost copy of the slave node. For this reason duplicate
constraint equations may be returned when (in parallel) the calling processor owns
the face and has a ghost of the slave node.

1.10 Errors

ACME will trap internal errors whenever possible and return gracefully to the host code.
ACME will never try to recover from an error; it will simply return control to the host
code. The host code, therefore, has the final decision of how to proceed. At the moment an
internal error is detected, ACME will immediately return to the host code without attempt-
ing to finish processing or attempting to ensure its internal data are consistent. As a result,
it is essential that the host code check for errors. Interactions may not be reasonable if an
internal error was encountered.

Errors are reported in two ways. First, all public access functions that could encounter an
error return a ContactErrorCode (an enumeration in the ContactSearch header file). This
error return code will be globally synchronized (i.e., all processors will return the same
value).

The current enumeration for error codes is:

```c
enum ContactErrorCode{
    NO_ERROR = 0,
    ID_NOT_FOUND,
    UNKNOWN_TYPE,
    INVALID_ID,
    INVALID_DATA,
    UNIMPLEMENTED_FUNCTION,
    ZOLTAN_ERROR,
    EXODUS_ERROR,
    INVALID_INTERACTION,
    INTERNAL_ERROR );
```

The return value is meant as an easy check for the host code to determine if an error oc-
curred on any processor. It does not specify which processor encountered the error, nor
does it return a real description of the error or the ID (if appropriate) to determine on what
entity the error occurred (e.g., which unimplemented function was called or, possibly in
the future, which face has a negative area). ACME does not normally write any data to the
standard output or error files (stdout or stderr). Instead, ACME provides functions to ex-
tract detailed error information line by line, which the host code can then direct to its own
output files as desired. Each line is limited to 80 characters.
1.11 Plotting

ACME can be built with a compile-time option to include an ExodusII plotting capability. The host code is responsible for creating the ExodusII file, including the name and location of the plot file. It is also responsible for closing the file after ACME writes its data. Because ACME writes double precision data, this file must be created with the ExodusII parameter ICOMPWS set to 8.

If the host code desires a plot file from ACME, it must create a new file for each time step. This capability is primarily intended as a debugging tool and is not envisioned for use in production calculations. Since the host code specifies the mesh topology and has access to the interactions, it has the ability to include the interaction data in its normal plotting functionality as it sees fit.

The mesh coordinates for each plot file are always taken as those in the current configuration. The displacements are the differences between the predicted and current coordinates if the predicted coordinates have been specified; otherwise the displacements are set to zero. Each Face_Block is treated as an element block (TRI3 for TRIFACEL3, TRI6 for TRIFACEQ6, and SHELL for QUADFACEL4 and QUADFACEQ8), as are Element_Blocks. Additional element blocks, one for each edge type, are created to represent the edges (BAR for LineEdgeL2 and BAR3 for LineEdgeQ3). An additional TRI3 element block is created to represent the FaceFace_Interactions. An additional BAR element block is created to represent the FaceCoverage_Interactions. Because ExodusII does not support node blocks, all the nodes are output without their associated Node_Block. Shell faces are plotted in their lofted configuration.

1.11.1 Search Data Plot Variables

Plot output data consists of search and enforcement object variables. Global, nodal and element search data are presented in this section. The search data global output variables are listed in Table 9.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_nn_interactions</td>
<td>total number of NodeNode_Interactions</td>
</tr>
<tr>
<td>num_nf_interactions</td>
<td>total number of NodeFace_Interactions</td>
</tr>
<tr>
<td>num_ns_interactions</td>
<td>total number of NodeSurface_Interactions</td>
</tr>
<tr>
<td>num_ff_interactions</td>
<td>total number of FaceFace_Interactions</td>
</tr>
<tr>
<td>num_fc_interactions</td>
<td>total number of FaceCoverage_Interactions</td>
</tr>
<tr>
<td>num_ee_interactions</td>
<td>total number of ElementElement_Interactions</td>
</tr>
<tr>
<td>mult_interaction_status</td>
<td>flag indicating if multiple interactions is on/off</td>
</tr>
</tbody>
</table>
The search data nodal output variables include both the nodal data (displacement and node normal) and the interactions (NodeFace_Interactions and NodeSurface_Interactions). The interactions are output for their associated node, rather than with the face. Currently, up to three interactions at a node can be output, with no meaning attached to their order. If a node has no interactions, all of the interaction data for that node will be zero. If a node has one interaction, the second and third sets of interaction data will all be zero, etc. Table 10 gives a description of all the nodal data written to the ExodusII file. (UNIX-style wildcard notation is used in this and subsequent tables. For example, displ[xyz] is shorthand for displx, disply, and displz.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>displ[xyz]</td>
<td>X, Y &amp; Z components of displacement</td>
</tr>
<tr>
<td>nnorm[xyz]</td>
<td>X, Y &amp; Z components of the unit node normal</td>
</tr>
<tr>
<td>numcon</td>
<td>Number of kinematic constraints at the node</td>
</tr>
<tr>
<td>convec[xyz]</td>
<td>X, Y &amp; Z components of kinematic constraint vector (provided by host)</td>
</tr>
<tr>
<td>face_id[123]</td>
<td>The ID of the face involved in interaction 1, 2, or 3 (0 if no interaction)</td>
</tr>
<tr>
<td>alg[123]</td>
<td>Algorithm used to define interaction 1, 2, or 3 (1=closest point projection for 1-configuration search, 2=closest point projection for 2-configuration search, 3=moving_intersection)</td>
</tr>
<tr>
<td>node_ek[123]</td>
<td>The node entity key for interaction 1, 2, or 3 (0 if no interaction)</td>
</tr>
<tr>
<td>gapcur[123]</td>
<td>The Gap arising from the current time step, not including any residual Gap</td>
</tr>
<tr>
<td>gapold[123]</td>
<td>The residual Gap from the previous time step for interaction 1, 2, or 3</td>
</tr>
</tbody>
</table>
The “element” data actually consist of the element, face, and edge data (since they are output as element blocks). The FaceFace_Interaction, FaceCoverage_Interaction, and ElementElement_Interaction data are also stored as element data. Table 11 gives the names and descriptions of the element data written to the ExodusII file.

Table 11 Search Data Element Variables for ExodusII Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fnorm[xyz]</td>
<td>Faces Unit face normal at centroid</td>
</tr>
<tr>
<td>curvature</td>
<td>Edges 0 = Unknown 1 = Convex 2 = Concave 3 = Concave with smoothing 4 = Convex with smoothing</td>
</tr>
</tbody>
</table>
### Table 11: Search Data Element Variables for ExodusII Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>angle_bf</td>
<td>Edges</td>
<td>The angle between the two faces connected to this edge. The value is zero if the edge is connected to only one face.</td>
</tr>
<tr>
<td>FFI[0-N]_FACE_ID</td>
<td>Faces</td>
<td>The ID of the master face involved in interaction 0, 1, ..., N=num_fhi_interactions-1</td>
</tr>
<tr>
<td>FFI[0-N]_NVERTS</td>
<td>Faces</td>
<td>The number of vertexes/edges in the polygon for interaction 0, 1, ..., N=num_fhi_interactions-1</td>
</tr>
<tr>
<td>FFI[0-N]_SX[0-M]</td>
<td>Faces</td>
<td>The 1st local coordinate on the slave face for interaction 0, 1, ..., N=num_fhi_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FFI[0-N]_SY[0-M]</td>
<td>Faces</td>
<td>The 2nd local coordinate on the slave face for interaction 0, 1, ..., N=num_fhi_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FFI[0-N]_MX[0-M]</td>
<td>Faces</td>
<td>The 1st local coordinate on the master face for interaction 0, 1, ..., N=num_fhi_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FFI[0-N]_MY[0-M]</td>
<td>Faces</td>
<td>The 2nd local coordinate on the master face for interaction 0, 1, ..., N=num_fhi_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FFI[0-N]_EDGE[0-M]</td>
<td>Faces</td>
<td>The flag indicating coincidence with an edge on the slave face for interaction 0, 1, ..., N=num_fhi_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FFI[0-N]_FLAG[0-M]</td>
<td>Faces</td>
<td>The flag indicating coincidence with an edge on the master face for interaction 0, 1, ..., N=num_fhi_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FCI[0-N]_NVERTS</td>
<td>Faces</td>
<td>The number of vertexes/edges in the polygon for interaction 0, 1, ..., N=num_fci_interactions-1</td>
</tr>
<tr>
<td>FCI[0-N]_X[0-M]</td>
<td>Faces</td>
<td>The 1st local coordinate on the face for interaction 0, 1, ..., N=num_fci_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
<tr>
<td>FCI[0-N]_Y[0-M]</td>
<td>Faces</td>
<td>The 2nd local coordinate on the face for interaction 0, 1, ..., N=num_fci_interactions-1 and vertex 0, 1, ..., M=nverts-1</td>
</tr>
</tbody>
</table>
1.11.2 Enforcement Data Plot Variables

The enforcement objects which provide ExodusII plot data include Gap Removal, Explicit Transient Dynamics, Tied Kinematics and Contact Volume Transfer. Depending on the enforcement, they provide nodal and element plot data. None of the enforcement objects provides global data. Nodal data for the Gap Removal, Explicit Transient Dynamics, Tied Kinematics and Contact Volume Transfer are presented in Table 12. Element data is provided only for the Contact Volume Transfer enforcement and these are presented in Table 13. As in the search, UNIX-style wildcard notation is used in this and subsequent tables.

Table 11 Search Data Element Variables for ExodusII Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimaryOwner</td>
<td>All</td>
<td>The processor that owns the entity in the primary decomposition</td>
</tr>
<tr>
<td>PrimaryLocalID</td>
<td>All</td>
<td>The local ID for the entity on the owning processor</td>
</tr>
<tr>
<td>SecondaryOwner</td>
<td>All</td>
<td>The processor that owns the entity in the secondary decomposition</td>
</tr>
<tr>
<td>Volume</td>
<td>Elems</td>
<td>The volume of the element</td>
</tr>
<tr>
<td>NumVolVolOverlaps</td>
<td>Elems</td>
<td>The maximum number of volume-volume overlaps detected for an element, V</td>
</tr>
<tr>
<td>EEI[0-V]_ELEM_ID</td>
<td>Elems</td>
<td>The ID of an overlapping element (0 if none)</td>
</tr>
<tr>
<td>EEI[0-V]_VOLUME</td>
<td>Elems</td>
<td>The volume of the overlap between this element and the overlapping element (0 if none)</td>
</tr>
<tr>
<td>ElmEnf[1-N]</td>
<td>Elems</td>
<td>Element enforcement variables. N depends on the enforcement (see Table 13 for details)</td>
</tr>
</tbody>
</table>

Table 12 Enforcement Data Nodal Variables for ExodusII Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Enforcement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodEnf[123]</td>
<td>Gap Removal</td>
<td>Displacement increment needed to remove gap in x- y- and z-coordinates (1, 2 and 3 respectively).</td>
</tr>
<tr>
<td>NodEnf[123]</td>
<td>Explicit Transient Dynamics</td>
<td>Total contact force at the node in x- y- and z- directions (1, 2 and 3 respectively)</td>
</tr>
</tbody>
</table>
1.12 Restart Capabilities

ACME currently provides two options for restart. The first restart option is a binary data stream, where all of the data are packed into one array to be written to a restart file. This binary data stream can then be used with a special constructor to restore the objects to their original state. The second restart option allows a host code to extract node, edge and face restart variables one at a time to be output to a restart file. The variable-based restart requires the host code to call the basic constructor for the objects and then “implant” the restart variables into the object, which restores the objects to their states before the restart. Both restart methods currently require that neither the mesh topology nor the decomposition change. Eventually, the ability to restart with a different number of processors will be supported with the variable-based restart capability; it will not be supported with the binary stream restart function. Note that ACME meshes incorporating shell faces (SHELLQUADFACEL4 and SHELLTRIFACE3) are currently restricted to the variable-based restart capability.
2. Utility Functions

ACME provides various utility functions that are either independent of the search and enforcement objects or are identical for those objects. These include functions to obtain information about the current version of ACME, to extract information about errors encountered within the ACME algorithms, to extract data needed to restart ACME processing, and to create ExodusII plot files.

In each section delineating the ACME API functions (Sections 2, 3, 4, 5, and 6), the different forms for the C++, C, and Fortran syntax are presented together for each function call. The C++ API uses the full object-oriented capabilities of the language. On the other hand, the C and Fortran APIs, which in actuality have been combined into a single interface, are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The FORTRAN macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address, not by value. For Fortran, there exists no capability to pass data by value, so simply specifying the name of the variable or array will allow it to be passed appropriately.

The Search_Interface.h and Enforcement_Interface.h header files, located in the ACME search and enforcement directories respectively, include the prototypes for the C and Fortran functions described in this chapter. The ContactSearch.h and ContactEnforcement.h files include the requisite C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactSearch.h; these indicate the acceptable integral values that may be used in the C and Fortran APIs.

2.1 Version Information

Functions are provided to obtain the ACME version number and its release date and to check the compile-time compatibility of the ACME library and the host code with respect to the MPI library.

2.1.1 Getting the Version ID

The following function returns the version of ACME, which is a character string of the form x.yz, where x is an integer representing the major version, y is an integer representing the minor version, and z is a letter representing the bug fix level. The initial release of this version of ACME will be 2.2a, the first bug fix release will be 2.2b, and so on. The prototype for this function is:

\[
\text{C++} \quad \text{const char* ACME_Version();}
\]

\[
\text{C} \quad \text{void FORTRAN(acme_version)( char* vers );}
\]

\[
\text{Fortran} \quad \text{acme_version( vers )}
\]

where
Utility Functions

vers is an array of characters of length 80 (in C, 81 including the terminal ‘\n’).

2.1.2 Getting the Version Date

The following function returns the release date for ACME, which is a character string of the form ‘June 30, 2003’. The prototype for this function is:

C++
const char* ACME_VersionDate();

C
void FORTRAN(acme_versiondate)( char* vers_date );

Fortran
acme_versiondate( vers_date )

where

vers_date is an array of characters of length 80 (in C, 81 including the terminal ‘\n’).

2.1.3 Checking Compatibility with MPI

The following function returns an error if the compilations of the host code and the ACME library are incompatible with respect to the MPI library. The host code should call this function with the host_compile argument set to MPI_COMPILE, which is defined in the ContactSearch header file to be 0 if CONTACT_NO_MPI is defined at compile time, and defined as 1 otherwise. This function will check for compatibility with the value of MPI_COMPILE defined during compilation of the ACME library. The prototype for this function is:

C++
int ACME_MPI_Compatibility(int host_compile);

C
void FORTRAN(acme_mpi_compatibility)( int* host_compile, int* error );

Fortran
acme_mpi_compatibility( host_compile, error )

where

host_compile is the value of MPI_COMPILE used during compilation of the host code.
error is the return error code for the C and Fortran APIs.

2.2 Errors

As discussed in Section 1.10, ACME attempts to trap internal errors whenever possible. There are C-style character strings that can be extracted to give a detailed description of what error(s) occurred during ACME processing for the search and enforcement objects. These strings are specific to the current processor. Therefore, each processor may have a different number of error messages. The error return code is synchronized in parallel so all processors return the same error code even if a processor did not encounter an error.
2.2.1 Getting the Number of Errors

The following functions, which are public member functions in the C++ API, determine how many error messages the current processor has written. The prototypes for these functions are:

\[
\begin{align*}
\text{C++} & \quad \text{int ContactSearch::Number_of_Errors();} \\
& \quad \text{int ContactTDEnforcement::Number_of_Errors();} \\
& \quad \text{int ContactGapRemoval::Number_of_Errors();} \\
& \quad \text{int ContactTiedKinematics::Number_of_Errors();} \\
& \quad \text{int ContactVolumeTransfer::Number_of_Errors();} \\
& \quad \text{int ContatMPCs::Number_of_Errors();} \\
\text{C} & \quad \text{FORTRAN(number_of_search_errors)( int* num_errors );} \\
& \quad \text{FORTRAN(number_of_td_errors)( int* num_errors );} \\
& \quad \text{FORTRAN(number_of_gap_errors)( int* num_errors );} \\
& \quad \text{FORTRAN(number_of_tied_errors)( int* num_errors );} \\
& \quad \text{FORTRAN(number_of_voltrans_errors)( int* num_errors );} \\
& \quad \text{FORTRAN(number_of_mpcs_errors)( int* num_errors );} \\
\text{Fortran} & \quad \text{number_of_search_errors( num_errors )} \\
& \quad \text{number_of_td_errors( num_errors )} \\
& \quad \text{number_of_gap_errors( num_errors )} \\
& \quad \text{number_of_tied_errors( num_errors )} \\
& \quad \text{number_of_voltrans_errors( num_errors )} \\
& \quad \text{number_of_mpc_eqns_errors( num_errors )}
\end{align*}
\]

where

\[
\text{num_errors is the number of error messages that should be extracted by the host code.}
\]

2.2.2 Extracting Error Messages

The following functions, which are public member functions in the C++ API, can be used to extract the character strings for each error message on a processor (the number of which can be determined by the functions described in the previous section):

\[
\begin{align*}
\text{C++} & \quad \text{const char* ContactSearch::Error_Message( int i );} \\
& \quad \text{const char* ContactTDEnforcement::Error_Message( int i );} \\
& \quad \text{const char* ContactGapRemoval::Error_Message( int i );} \\
& \quad \text{const char* ContactTiedKinematics::Error_Message( int i );} \\
& \quad \text{const char* ContactVolumeTransfer::Error_Message( int i );} \\
& \quad \text{const char* ContactMPCs::Error_Message( int i );} \\
\text{C} & \quad \text{FORTRAN(get_search_error_message)( int* i, char* message );} \\
& \quad \text{FORTRAN(get_td_error_message)( int* i, char* message );} \\
& \quad \text{FORTRAN(get_gap_error_message)( int* i,}
\end{align*}
\]
Utility Functions

```c
char* message );
FORTRAN(get_tied_kin_error_message)(
    int* i,
    char* message );
FORTRAN(get_vol_tran_error_message)(
    int* i,
    char* message );
FORTRAN( get_mpc_eqns_erros)(
    int* i,
    char* message );
```

```fortran
get_search_error_message( i, message )
get_td_error_message( i, message )
get_gap_error_message( i, message )
get_tied_kin_error_message( i, message )
get_vol_tran_error_message( i, message )
get_mpc_eqns_error_message( i, message )
```

where

- `i` is the Fortran index of the error message (i.e., 1 to `Number_of_Errors()`, or `num_errors` for C and Fortran).
- `message` is an array of characters of length 80 (in C, 81 including the terminal ‘\n’).

### 2.3 Binary Stream Restart Functions

ACME provides functionality to allow restart using a single binary stream of data for each ACME search or enforcement object. The host code is responsible for allocating the array to hold the data, calling the functions, and writing the data to a restart file. Upon restart, the host code should supply the binary data stream to the special constructors described in this section, which will restore the objects to their state before restart.

Currently, ACME does not support restarting analyses which contain shell faces (SHELLQUADFACE14 and SHELLTRIFACE13) using the binary stream restart capability. Restart with shells is supported through the variable-based restart functions.

#### 2.3.1 Getting the Binary Restart Size

The following functions allow the host code to determine how much memory to allocate to store restart information for the search and enforcement objects. The return value is the number of double locations that are needed.

```c++
int ContactSearch::Restart_Size();
int ContactTDEnforcement::Restart_Size();
int ContactGapRemoval::Restart_Size();
int ContactTiedKinematics::Restart_Size();
int ContactVolumeTransfer::Restart_Size();
int ContactMPCs::Restart_Size();
```

```c
FORTRAN(search_restart_size)( int* size );
```
Utility Functions

FORTRAN(td_enf_restart_size)( int* size );
FORTRAN(gap_removal_restart_size)( int* size );
FORTRAN(tied_kin_restart_size)( int* size );
FORTRAN(vol_tran_restart_size)( int* size );
FORTRAN(mpc_eqns_restart_size( int* size );

Fortran
search_restart_size( size )
td_enf_restart_size( size )
gap_removal_restart_size( size )
tied_kin_restart_size( size )
vol_tran_restart_size( size )
mpc_eqns_restart_size( size )

where

size is the number of double locations that are needed for the restart data.

2.3.2 Extracting the Binary Restart Data

The following functions allow the host code to extract all the information needed to initialize an ACME object to its current state.

C++
ContactErrorCode
ContactSearch::Extract_Restart_Data(
    double* restart_data);
ContactErrorCode
ContactTDEnforcement::Extract_Restart_Data(
    double* restart_data);
ContactErrorCode
ContactGapRemoval::Extract_Restart_Data(
    double* restart_data);
ContactErrorCode
ContactTiedKinematics::Extract_Restart_Data(
    double* restart_data);
ContactErrorCode
ContactVolumeTransfer::Extract_Restart_Data(
    double* restart_data);
ContactErrorCode
ContactMPCs::Extract_Restart_Data(
    double* restart_data);

C
FORTRAN(search_extract_restart)(
    double* restart_data,
    int* error);
FORTRAN(td_enf_extract_restart)(
    double* restart_data,
    int* error);
FORTRAN(gap_removal_extract_restart)(
    double* restart_data,
    int* error);
FORTRAN(tied_kin_extract_restart(
    double* restart_data,
    int* error);
Utility Functions

```c
int* error);
FORTRAN(vol_tran_extract_restart(  
double* restart_data,  
int* error);
FORTRAN(mpc_eqns_extract_restart(  
double* restart_data,  
int* error);
```

```fortran
search_extract_restart( restart_data, error)
```

```fortran
td_enf_extract_restart( restart_data, error)
gap_removal_extract_restart( restart_data, error)
tied_kin_extract_restart( restart_data, error)
vol_tran_extract_restart( restart_data, error)
mpc_eqns_extract_restart( restart_data, error)
```

where

- `restart_data` is an array of type double. The length of this array is obtained from the function `Restart_Size()` (see the previous section).
- `error` is the return error code for the C and Fortran APIs.

### 2.3.3 Constructing Objects Upon Restart

Note, ACME does not enforce the requirement that host code global ids be identical across a restart. This allows the host code to renumber nodes and faces when performing a restart (due to adaptivity or element birth/death) but also requires that the host code supply the host code global id numbering to ACME when performing a restart. As noted above, a second constructor is available to allow for restarts from the binary data stream provided by the `Extract_Restart_Data` functions described in Section 2.3.2:

```cpp
ContactSearch::ContactSearch(  
    const double* restart_data,  
    const int* node_global_ids,  
    const int* face_global_ids,  
    const MPI_Comm& mpi_communicator,  
    ContactErrorCode& error );
```

```cpp
ContactTDEnforcement::ContactTDEnforcement(  
    ContactSearch* search,  
    double* restart_data,  
    int* error );
```

```cpp
ContactGapRemoval::ContactGapRemoval(  
    ContactSearch* search,  
    double* restart_data,  
    int* error );
```

```cpp
ContactTiedKinematics::ContactTiedKinematics(  
    ContactSearch* search,  
    double* restart_data,  
    int* error );
```

```cpp
ContactVolumeTransfer::ContactVolumeTransfer(  
    ContactSearch* search,
```
double* restart_data,
int* error);
ContactMPCs::ContactMPCs(
    ContactSearch* search,
    double* restart_data,
    int* error);

FORTRAN(build_search_restart)(
    double* restart_data,
    int* node_global_ids,
    int* face_global_ids,
    int* mpi_communicator,
    int* error);

FORTRAN(build_td_enf_restart)(
    double* restart_data,
    int* error);

FORTRAN(build_gap_removal_restart)(
    double* restart_data,
    int* error);

FORTRAN(build_tied_kin_restart)(
    double* restart_data,
    int* error);

FORTRAN(build_vol_tran_restart)(
    double* restart_data,
    int* error);

FORTRAN(build_mpc_eqns_restart)(
    double* restart_data,
    int* error);

Fortran build_search_restart(
    restart_data,
    node_global_ids,
    face_global_ids,
    mpi_communicator,
    error)
build_td_enf_restart(
    restart_data,
    error)
build_gap_removal_restart(
    restart_data,
    error)
build_tied_kin_restart(
    restart_data,
    error)
build_vol_tran_restart(
    restart_data,
    error)
build_mpc_eqns_restart(
    restart_data,
    error)

where
Utility Functions

restart_data is an array of type double. The length of this array is obtained from the function Restart_Size() (see the previous section).

node_global_ids is an array (whose length is twice the total number of nodes in all Node_Blocks) containing the host code ID for each node. Each host code ID consists of two integers. For node N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. The IDs for the first Node_Block are listed first in the array, followed by the IDs for each of the other Node_Blocks in order (if applicable).

face_global_ids is an array (whose length is twice the total number of faces in all Face_Blocks) containing the host code ID for each face. Each host code ID consists of two integers. For face N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. The IDs for the first Face_Block are listed first in the array, followed by the IDs for each of the other Face_Blocks in order (if applicable).

mpi_communicator is an MPI communicator if ACME was built for a parallel and is simply a dummy int otherwise.

search is the associated ContactSearch object for this enforcement object. This is hidden in the C and Fortran APIs because only one search object is allowed.

error is the error return code that will reflect any errors that were detected.

2.4 Variable-Based Restart Functions

The variable-based restart functions allow a host code to extract all the restart variables from the ACME objects variable by variable. This set of functions will eventually allow restarts on different numbers of processors, although that capability is not supported in this release. There are no separate constructors for this type of restart. Instead, the traditional constructor is used and then the variable-based data are “implanted.” ACME does support restarting analyses which contain shell faces (SHELLQUADFACE1L4 and SHELLTRIFACE1L3) using the variable-based restart capability.

2.4.1 Obtaining the Number of General Restart Variables

These functions supply the number of general variables from each search and enforcement object that need to be written to (or read from) a restart file. A general variable is a variable that is true for an entire search or enforcement, and is not related to any specific node, edge, or face.

C++

```cpp
int ContactSearch::Number_General_Restart_Variables();
int ContactTDEnforcement::Number_General_Restart_Variables();
int ContactGapRemoval::Number_General_Restart_Variables();
int ContactTiedKinematics::Number_General_Restart_Variables();
int ContactVolumeTransfer::Number_General_Restart_Variables();
int ContactMPCs::Number_General_Restart_Variables();
```

C

```fortran
FORTRAN( search_num_general_rsvars )( int* num_gvars );
FORTRAN( td_enf_num_general_rsvars )( int* num_gvars );
FORTRAN( gap_removal_num_general_rsvars )
( int* num_gvars );
```
Utility Functions

```fortran
FORTRAN( tied_kin_num_general_rsvars  ) ( int* num_gvars );
FORTRAN( vol_tran_num_general_rsvars ) ( int* num_gvars );
FORTRAN( mpc_eqns_num_general_rsvars ) ( int* num_gvars );
```

where

num_gvars is the number of general restart variables

### 2.4.2 Obtaining the Number of Nodal Restart Variables

These functions supply the number of nodal variables from each search and enforcement object that need to be written to (or read from) a restart file.

```cpp
int ContactSearch::Number_Nodal_Restart_Variables();
int ContactTDEnforcement::Number_Nodal_Restart_Variables();
int ContactGapRemoval::Number_Nodal_Restart_Variables();
int ContactTiedKinematics::Number_Nodal_Restart_Variables();
int ContactVolumeTransfer::Number_Nodal_Restart_Variables();
int ContactMPCs::Number_Nodal_Restart_Variables();
```

```c
FORTRAN( search_num_node_rsvars ) ( int* num_nvars );
FORTRAN( td_enf_num_node_rsvars ) ( int* num_nvars );
FORTRAN( gap_removal_num_node_rsvars ) ( int* num_nvars );
FORTRAN( tied_kin_num_node_rsvars ) ( int* num_nvars );
FORTRAN( vol_tran_num_node_rsvars ) ( int* num_nvars );
FORTRAN( mpc_eqns_num_node_rsvars ) ( int* num_nvars );
```

where

num_nvars is the number of nodal restart variables
Utility Functions

2.4.3 Obtaining the Number of Edge Restart Variables

These functions supply the number of edge variables from each search and enforcement object that need to be written to (or read from) a restart file. Currently, there are no edge-based restart variables, so the ambiguity of how to handle the issue that edges are internally generated, not supplied by the host code, is deferred until edge-based restart variables are required. These functions are included here to complete the API.

```cpp
int ContactSearch::Number_Edge_Restart_Variables();
int ContactTDEnforcement::Number_Edge_Restart_Variables();
int ContactGapRemoval::Number_Edge_Restart_Variables();
int ContactTiedKinematics::Number_Edge_Restart_Variables();
int ContactVolumeTransfer::Number_Edge_Restart_Variables();
int ContactMPCs::Number_Edge_Restart_Variables();
```

```fortran
search_num_edge_rsvars( num_evars )
td_enf_num_edge_rsvars( num_evars )
gap_removal_num_edge_rsvars( num_evars )
tied_kin_num_edge_rsvars( num_evars )
vol_tran_num_edge_rsvars( num_evars )
mpc_eqns_num_edge_rsvars( num_evars )
```

where

```
num_evars is the number of edge restart variables
```

2.4.4 Obtaining the Number of Face Restart Variables

These functions supply the number of face variables from each search and enforcement object that need to be written to (or read from) a restart file.

```cpp
int ContactSearch::Number_Face_Restart_Variables();
int ContactTDEnforcement::Number_Face_Restart_Variables();
int ContactGapRemoval::Number_Face_Restart_Variables();
```
Utility Functions

ContactTiedKinematics::Number_Face_Restart_Variables();
int ContactVolumeTransfer::Number_Face_Restart_Variables();
int ContactMPCs::Number_Face_Restart_Variables();

C
FORTRAN( search_num_face_rsvars ) ( int* num_fvars );
FORTRAN( td_enf_num_face_rsvars ) ( int* num_fvars );
FORTRAN( gap_num_face_rsvars ) ( int* num_fvars );
FORTRAN( tied_kin_num_face_rsvars ) ( int* num_fvars );
FORTRAN( vol_tran_num_face_rsvars ) ( int* num_fvars );
FORTRAN( mpc_eqns_num_face_rsvars ) ( int* num_fvars );

Fortran
search_num_face_rsvars( num_fvars )
td_enf_num_face_rsvars( num_fvars )
gap_num_face_rsvars( num_fvars )
tied_kin_num_face_rsvars( num_fvars )
vol_tran_num_face_rsvars( num_fvars )
mpc_eqns_num_face_rsvars( num_fvars )

where

num_fvars is the number of face restart variables

2.4.5 Obtaining the Number of Element Restart Variables

These functions supply the number of element variables from each search and enforcement object that need to be written to (or read from) a restart file.

C++
int ContactSearch::Number_Element_Restart_Variables();
int ContactTDEnforcement::Number_Element_Restart_Variables();
int ContactGapRemoval::Number_Element_Restart_Variables();
int ContactTiedKinematics::Number_Element_Restart_Variables();
int ContactVolumeTransfer::Number_Element_Restart_Variables();
int ContactMPCs::Number_Element_Restart_Variables();

C
FORTRAN( search_num_element_rsvars ) ( int* num_elvars );
FORTRAN( td_enf_num_element_rsvars ) ( int* num_elvars );
FORTRAN( gap_num_element_rsvars ) ( int* num_elvars );
FORTRAN( tied_kin_num_element_rsvars ) ( int* num_elvars );
FORTRAN( vol_tran_num_element_rsvars ) ( int* num_elvars );
FORTRAN( mpc_eqns_num_element_rsvars ) ( int* num_elvars );

Fortran
search_num_element_rsvars( num_elvars )
td_enf_num_element_rsvars( num_elvars )
gap_num_element_rsvars( num_elvars )
tied_kin_num_element_rsvars( num_elvars )
Utility Functions

```cpp
vol_tran_num_element_rsvars( num_elvars )
mpc_eqns_num_element_rsvars( num_elvars )
```

where

```
num_elvars is the number of element restart variables
```

### 2.4.6 Extracting the General Restart Variables

These functions extract the general variables that are required for restart for the search and enforcement objects.

**C++**

```cpp
ContactErrorCode
ContactSearch::Extract_General_Restart_Variable( 
    double* variable_data );
ContactErrorCode
ContactTDEnforcement::Extract_General_Restart_Variable( 
    double* variable_data );
ContactErrorCode
ContactGapRemoval::Extract_General_Restart_Variable( 
    double* variable_data );
ContactErrorCode
ContactTiedKinematics::Extract_General_Restart_Variable( 
    double* variable_data );
ContactErrorCode
ContactVolumeTransfer::Extract_General_Restart_Variable( 
    double* variable_data );
ContactErrorCode
ContactMPCs::Extract_General_Restart_Variable( 
    double* variable_data );
```

**C**

```c
FORTRAN(search_extract_general_rsvars)( 
    double* variable_data, 
    int* error );
FORTRAN(td_extract_general_rsvars)( 
    double* variable_data, 
    int* error );
FORTRAN(gap_extract_general_rsvars)( 
    double* variable_data, 
    int* error );
FORTRAN(tied_kin_extract_general_rsvars)( 
    double* variable_data, 
    int* error );
FORTRAN(vol_tran_extract_general_rsvars)( 
    double* variable_data, 
    int* error );
FORTRAN(mpc_eqns_extract_general_rsvars)( 
    double* variable_data, 
    int* error );
```

**Fortran**

```fortran
search_extract_general_rsvars( 
    variable_data, 
    int* error );
```
Utility Functions

```c
error )
td_extract_general_rsvars(
    variable_data,
    error )
gap_extract_general_rsvars(
    variable_data,
    error )
tied_kin_extract_general_rsvars)(
    variable_data,
    error )
vol_tran_extract_general_rsvars(
    variable_data,
    error )
mpc_eqns_extract_general_rsvars(
    variable_data,
    error )
```

where

variable_data is an array of type double. The length of the array is the number of global variables as given by the functions described above.

error is the return error code for the C and Fortran APIs.

### 2.4.7 Implanting the General Restart Variables

These functions implant the general variables that are required for restart for the search and enforcement objects.

```c++
ContactErrorCode
ContactSearch::Implant_General_Restart_Variable(
    double* variable_data );
ContactErrorCode
ContactTDEnforcement::Implant_General_Restart_Variable(
    double* variable_data );
ContactErrorCode
ContactGapRemoval::Implant_General_Restart_Variable(
    double* variable_data );
ContactErrorCode
ContactTiedKinematics::Implant_General_Restart_Variable(
    double* variable_data );
ContactErrorCode
ContactVolumeTransfer::Implant_General_Restart_Variable(
    double* variable_data );
ContactErrorCode
ContactMPCs::Implant_General_Restart_Variable(
    double* variable_data );
```

```fortran
FORTRAN(search_implant_general_rsvrs)
    ( double* variable_data,
    int* error );
FORTRAN(td_implant_general_rsvrs)
    ( double* variable_data,
    int* error );
```
Utility Functions

FORTRAN(gap_implant_general_rsvars)(
    double* variable_data,
    int* error );
FORTRAN(tied_kin_implant_general_rsvars)(
    double* variable_data,
    int* error );
FORTRAN(voltrans_implant_general_rsvars)(
    double* variable_data,
    int* error );
FORTRAN(mpc_eqns_implant_general_rsvars)(
    double* variable_data,
    int* error );

Fortran
search_implant_general_rsvars(
    variable_data,
    error )
td_implant_general_rsvars(
    variable_data,
    error )
gap_implant_general_rsvars(
    variable_data,
    error )
tied_kin_implant_general_rsvars(
    variable_data,
    error )
voltrans_implant_general_rsvars(
    variable_data,
    error )
mpc_eqns_implant_general_rsvars(
    variable_data,
    error )

where

variable_data is an array of type double. The length of the array is the number of global variables as given by the functions described above.
error is the return error code for the C and Fortran APIs.

2.4.8 Extracting the Nodal Restart Variables

These functions extract the nodal variables, one by one, that are required for restart.

C++
ContactErrorCode
ContactSearch::Extract_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactTDEnforcement::Extract_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactGapRemoval::Extract_Nodal_Restart_Variable(
Utility Functions

```c
int variable_number,
double* variable_data );
ContactErrorCode
ContactTiedKinematics::Extract_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactVolumeTransfer::Extract_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactMPCs::Extract_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data );
```

```fortran
C
FORTRAN(search_extract_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(td_extract_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(gap_extract_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(tied_kin_extract_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(vol_tran_extract_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(mpc_eqns_extract_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
```

```fortran
Fortran search_extract_node_rsvars(
    variable_number,
    variable_data,
    error )
td_extract_node_rsvars(
    variable_number,
    variable_data,
    error )
gap_extract_node_rsvars(
    variable_number,
    variable_data,
    error )
tied_kin_extract_node_rsvars)(
    int variable_number,
```
Utility Functions

```c
double* variable_data,
int* error);
vol_tran_extract_node_rsvars(
    int variable_number,
    double* variable_data,
    int* error);
mpc_eqns_extract_node_rsvars(
    int variable_number,
    double* variable_data,
    int* error);
```

where

- `variable_number` is the variable number (using Fortran indexing; i.e., from 1 to N).
- `variable_data` is an array of type double. The length of the array is given by the number of nodes in the surface topology for this processor (as supplied in the constructor).
- `error` is the return error code for the C and Fortran APIs.

### 2.4.9 Implanting the Nodal Restart Variables

These functions implant the nodal variables, one by one, that are required for restart.

```c
C++
ContactErrorCode
ContactSearch::Implant_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactTDEnforcement::Implant_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactGapRemoval::Implant_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactTiedKinematics::Implant_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactVolumeTransfer::Implant_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactMPCs::Implant_Nodal_Restart_Variable(
    int variable_number,
    double* variable_data);
```

```c
C
FORTRAN(search_implant_node_rsvsrs)(
    int variable_number,
    double* variable_data,
    int* error);
FORTRAN(td_implant_node_rsvsrs)(
    int variable_number,
```
Utility Functions

```c
    double* variable_data,
    int* error );
FORTRAN(gap_implant_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(tied_kin_implant_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(vol_tran_implant_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(mpc_eqns_implant_node_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
```

```fortran
  Fortran  search_implant_node_rsvars(
    variable_number,
    variable_data,
    error )
  td_implant_node_rsvars(
    variable_number,
    variable_data,
    error )
  gap_implant_node_rsvars(
    variable_number,
    variable_data,
    error )
  tied_kin_implant_node_rsvars(
    variable_number,
    variable_data,
    error )
  vol_tran_implant_node_rsvars(
    variable_number,
    variable_data,
    error )
  mpc_eqns_implant_node_rsvars(
    variable_number,
    variable_data,
    error )
```

where

- `variable_number` is the variable number (using Fortran indexing; i.e., from 1 to N).
- `variable_data` is an array of type double. The length of the array is given by the number of nodes in the surface topology for this processor (as supplied in the constructor).
2.4.10 Extracting the Edge Restart Variables

These functions extract the edge variables, one by one, that are required for restart. As previously mentioned, there are currently no edge-based restart variables, so these functions will not be used in this version of ACME.

**C++**

```cpp
ContactErrorCode
ContactSearch::Extract_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactTDEnforcement::Extract_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactGapRemoval::Extract_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactTiedKinematics::Extract_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactVolumeTransfer::Extract_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactMPCs::Extract_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
```

**C**

```c
FORTRAN(search_extract_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(td_extract_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(gap_extract_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(tied_kin_extract_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(vol_tran_extract_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(mpc_eqns_extract_edge_rsvars)(
    int variable_number,
```
Utility Functions

```c
double* variable_data,
int* error);
```

**Fortran**
```fortran
search_extract_edge_rsvars(
    variable_number,
    variable_data,
    error
)
dt_extract_edge_rsvars(
    variable_number,
    variable_data,
    error
)gap_extract_edge_rsvars(
    variable_number,
    variable_data,
    error
)tied_kin_extract_edge_rsvars(
    variable_number,
    variable_data,
    error
)vol_tran_extract_edge_rsvars(
    variable_number,
    variable_data,
    error
)mpc_eqns_extract_edge_rsvars(
    variable_number,
    variable_data,
    error
)
```

where

- `variable_number` is the variable number (using Fortran indexing; i.e., from 1 to N).
- `variable_data` is an array of type double. The length of the array is given by the number of edges in the surface topology for this processor.

### 2.4.11 Implanting the Edge Restart Variables

These functions implant the edge variables, one by one, that are required for restart. As previously mentioned, there are currently no edge-based restart variables so these functions will not be used in this version of ACME.

```c++
ContactErrorCode
ContactSearch::Implant_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactTDEnforcement::Implant_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactGapRemoval::Implant_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
```
Utility Functions

ContactErrorCode
ContactTiedKinematics::Implant_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactVolumeTransfer::Implant_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactMPCs::Implant_Edge_Restart_Variable(
    int variable_number,
    double* variable_data );

C
FORTRAN(search_implant_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(td_implant_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(gap_implant_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(tied_kin_implant_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(vol_tran_implant_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(mpc_eqns_implant_edge_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );

Fortran
search_implant_edge_rsvars(
    variable_number,
    variable_data,
    error )
td_implant_edge_rsvars(
    variable_number,
    variable_data,
    error )
gap_implant_edge_rsvars(
    variable_number,
    variable_data,
    error )
tied_kin_implant_edge_rsvars(
    variable_number,
    variable_data,
    error )
Utility Functions

vol_tran_implant_edge_rsvars(
    variable_number,
    variable_data,
    error )
mpc_eqns_implant_edge_rsvars(
    variable_number,
    variable_data,
    error )

where

variable_number is the variable number (using Fortran indexing; i.e, from 1 to N).
variable_data is an array of type double. The length of the array is given by the number of edges in
the surface topology for this processor.

2.4.12 Extracting the Face Restart Variables

These functions extract the face variables, one by one, that are required for restart.

C++
ContactErrorCode
ContactSearch::Extract_Face_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactTDEnforcement::Extract_Face_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactGapRemoval::Extract_Face_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactTiedKinematics::Extract_Face_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactVolumeTransfer::Extract_Face_Restart_Variable(
    int variable_number,
    double* variable_data );
ContactErrorCode
ContactMPCs::Extract_Face_Restart_Variable(
    int variable_number,
    double* variable_data );

C
FORTRAN(search_extract_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
FORTRAN(td_extract_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );
Utility Functions

FORTRAN(gap_extract_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(tied_kin_extract_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(vol_tran_extract_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(mpc_eqns_extract_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error );

Fortran search_extract_face_rsvars(
    variable_number,
    variable_data,
    error )

td_extract_face_rsvars(
    variable_number,
    variable_data,
    error )

gap_extract_face_rsvars(
    variable_number,
    variable_data,
    error )

tied_kin_extract_face_rsvars(
    variable_number,
    variable_data,
    error )

vol_tran_extract_face_rsvars(
    variable_number,
    variable_data,
    error )

mpc_eqns_extract_face_rsvars(
    variable_number,
    variable_data,
    error )

where

variable_number is the variable number (using Fortran indexing; i.e, from 1 to N).
variable_data is an array of type double. The length of the array is given by the number of faces in
the surface topology for this processor (as supplied in the constructor).

2.4.13 Implanting the Face Restart Variables

These functions implant the face variables, one by one, that are required for restart.
Utility Functions

C++

```cpp
ContactErrorCode
ContactSearch::Implant_Face_Have_Data(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactTDEnforcement::Implant_Face_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactGapRemoval::Implant_Face_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactTiedKinematics::Implant_Face_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactVolumeTransfer::Implant_Face_Restart_Variable(
    int variable_number,
    double* variable_data);
ContactErrorCode
ContactMPCs::Implant_Face_Restart_Variable(
    int variable_number,
    double* variable_data);
```

C

```c
FORTRAN(search_implant_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);
FORTRAN(td_implant_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);
FORTRAN(gap_implant_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);
FORTRAN(tied_kin_implant_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);
FORTRAN(vol_tran_implant_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);
FORTRAN(mpc_eqns_implant_face_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);
```

Fortran

```fortran
search_implant_face_rsvars(
    variable_number,
    variable_data,
    error)
```
Utility Functions

td_implant_face_rsvars(
    variable_number,
    variable_data,
    error )
gap_implant_face_rsvars(
    variable_number,
    variable_data,
    error )
tied_kin_implant_face_rsvars(
    variable_number,
    variable_data,
    error )
vol_tran_implant_face_rsvars(
    variable_number,
    variable_data,
    error )
mpc_eqns_implant_face_rsvars(
    variable_number,
    variable_data,
    error )

where

variable_number is the variable number (using Fortran indexing; i.e, from 1 to N).
variable_data is an array of type double. The length of the array is given by the number of faces in
the surface topology for this processor (as supplied in the constructor).

2.4.14 Extracting the Element Restart Variables

These functions extract the element variables, one by one, that are required for restart.

C++

ContactErrorCode
ContactSearch::Extract_Element_Restart_Variable(
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactTDEnforcement::Extract_Element_Restart_Variable(
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactGapRemoval::Extract_Element_Restart_Variable(
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactTiedKinematics::Extract_Element_Restart_Variable(
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactVolumeTransfer::Extract_Element_Restart_Variable(
    int variable_number,
    double* variable_data );

ContactErrorCode
Utility Functions

ContactMPCs::Extract_Element_Restart_Variable(
    int variable_number,
    double* variable_data);

C

FORTRAN(search_extract_element_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);

FORTRAN(td_extract_element_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);

FORTRAN(gap_extract_element_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);

FORTRAN(tied_kin_extract_element_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);

FORTRAN(vol_tran_extract_element_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);

FORTRAN(mpc_eqns_extract_element_rsvars)(
    int variable_number,
    double* variable_data,
    int* error);

Fortran

search_extract_element_rsvars(
    variable_number,
    variable_data,
    error)

td_extract_element_rsvars(
    variable_number,
    variable_data,
    error)

gap_extract_element_rsvars(
    variable_number,
    variable_data,
    error)

tied_kin_extract_element_rsvars(
    variable_number,
    variable_data,
    error)

vol_tran_extract_element_rsvars(
    variable_number,
    variable_data,
    error)

mpc_eqns_extract_element_rsvars(
    variable_number,
    variable_data,
    error)
Utility Functions

where

variable_number is the variable number (using Fortran indexing; i.e., from 1 to N).
variable_data is an array of type double. The length of the array is given by the number of elements
in the surface topology for this processor (as supplied in the constructor).

2.4.15 Implanting the Element Restart Variables

These functions implant the element variables, one by one, that are required for restart.

```cpp
ContactErrorCode
ContactSearch::Implant_Element_Restart_Variable(  
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactTDEnforcement::Implant_Element_Restart_Variable(  
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactGapRemoval::Implant_Element_Restart_Variable(  
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactTiedKinematics::Implant_Element_Restart_Variable(  
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactVolumeTransfer::Implant_Element_Restart_Variable(  
    int variable_number,
    double* variable_data );

ContactErrorCode
ContactMPCs::Implant_Element_Restart_Variable(  
    int variable_number,
    double* variable_data );

C

FORTRAN(search_implant_element_rsvars)(  
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(td_implant_element_rsvars)(  
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(gap_implant_element_rsvars)(  
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(tied_kin_implant_element_rsvars)(  
    int variable_number,
    double* variable_data,
    int* error );

FORTRAN(vol_tran_implant_element_rsvars)(  
    int variable_number,
    double* variable_data,
    int* error );
```
Utility Functions

```c
int variable_number,
double* variable_data,
int* error);
```

```
FORTRAN(mpc_eqns_implant_element_rsvars)(
int variable_number,
double* variable_data,
int* error );
```

```
Fortran search_implant_element_rsvars(
  variable_number,
  variable_data,
  error )
```

```
td_implant_element_rsvars(
  variable_number,
  variable_data,
  error )
```

```
gap_implant_element_rsvars(
  variable_number,
  variable_data,
  error )
```

```
tied_kin_implant_element_rsvars(
  variable_number,
  variable_data,
  error )
```

```
vol_tran_implant_element_rsvars(
  variable_number,
  variable_data,
  error )
```

```
mpc_eqns_implant_element_rsvars(
  variable_number,
  variable_data,
  error )
```

where

- `variable_number` is the variable number (using Fortran indexing; i.e., from 1 to N).
- `variable_data` is an array of type double. The length of the array is given by the number of elements in the surface topology for this processor (as supplied in the constructor).

### 2.4.16 Completing a Variable-Based Restart

These functions must be called after constructing an ACME object and implanting the restart variables with the functions described previously. These functions restore each ACME object to its state prior to restart. After these functions have been called, normal calculations can resume.

```cpp
ContactErrorCode
ContactSearch::Complete_Restart();
ContactErrorCode
ContactTDEnforcement::Complete_Restart();
ContactErrorCode
ContactGapRemoval::Complete_Restart();
```
Utility Functions

ContactErrorCode
    ContactTiedKinematics::Complete_Restart();
ContactErrorCode
    ContactVolumeTransfer::Complete_Restart();
ContactErrorCode
    ContactMPCs::Complete_Restart();

    C
        FORTRAN(search_complete_restart)( int* error );
        FORTRAN(td_enf_complete_restart)( int* error );
        FORTRAN(gap_complete_restart)( int* error );
        FORTRAN(tied_kin_complete_restart)( int* error );
        FORTRAN(vol_tran_complete_restart)( int* error );
        FORTRAN(mpc_eqns_complete_restart)( int* error );

    Fortran
        search_complete_restart( error )
        td_enf_complete_restart( error )
        gap_complete_restart( error )
        tied_kin_complete_restart( error )
        vol_tran_complete_restart( error )
        mpc_eqns_complete_restart( error )

where

    error is the return error code for the C and Fortran APIs.

2.5 Creating an Exodus Plot File of the Search & Enforcement Data

ACME has the ability to write an ExodusII file that contains the full search topology and all of the interaction data, including enforcement results. This function can be used only if ACME was built with ExodusII support (a compile time option). See Section 1.11 for a detailed description of the data written to the ExodusII file. The host code is required to actually open and close the ExodusII file, so it must choose the name and location for the file. This file must be opened with ICOMPWS=8. The ExodusII ID is then passed to ACME, which writes the topology and the results data. Only one plot step can be written to each file. The number of variables in the database depends on the number of interaction types currently active in the search object (which can change each time step).

The prototype for this capability is:

    C++
        ContactErrorCode ContactSearch::Exodus_Output(
            int exodus_id,
            double time );
    C
        FORTRAN(exodus_output)(
            int* exodus_id,
            double* time,
            int* error );
    Fortran
        exodus_output(
            exodus_id,
            time,
            error )
where

exodus_id is the integer database ID returned by the ExodusII library from an ex_create call.
time is the time value for the “results” to be written to the ExodusII file.
error is the return error code for the C and Fortran APIs.
Utility Functions
3. Search Functions

This section describes functions that construct and operate on ContactSearch “objects.” For the C++ API, these are true objects permitted by the object-oriented capabilities of the language. There are no static variables, so an arbitrary number of objects may be simultaneously active. In the C and Fortran APIs, these functions create and operate on a ContactSearch “object,” only one of which is currently allowed. Functions are provided to allow destruction of the ContactSearch object and creation of a new object at any point. Multiple objects can be supported in the future if the need ever arises.

There are two constructors for the ContactSearch object. The first, described in this section, is intended for general use, while the second, described in Section 2, is used to construct a search object using data read in from a previously generated restart file. The ContactSearch object is neither copy-able nor assignable.

In each section delineating the ACME API functions (Sections 2, 3, 4, 5, and 6), the different forms for the C++, C, and Fortran syntax are presented together for each function call. The C++ API uses the full object-oriented capabilities of the language. On the other hand, the C and Fortran APIs, which in actuality have been combined into a single interface, are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The FORTRAN macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address, not by value. For Fortran, there exists no capability to pass data by value, so simply specifying the name of the variable or array will allow it to be passed appropriately.

The Search_Interface.h and Enforcement_Interface.h header files, located in the ACME search and enforcement directories respectively, include the prototypes for the C and Fortran functions described in this chapter. The ContactSearch.h and ContactEnforcement.h files include the requisite C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactSearch.h; these indicate the acceptable integral values that may be used in the C and Fortran APIs.

3.1 Creating a ContactSearch Object

There is one general constructor for the ContactSearch object. A second constructor for restart is described in Section 2.3.2. The prototype for the general constructor is:

```c++
ContactSearch::ContactSearch(
    int dimensionality,
    int number_of_states,
    int number_of_analytic_surfaces,
    int number_of_node_blocks,
    const ContactSearch::ContactNode_Type* node_block_types,
    const int* number_of_nodes_in_blocks,
    const int* node_exodus_ids,
    const int* node_global_ids,
```
Search Functions

const double* coordinates,
int number_of_face_blocks,
const ContactSearch::ContactFace_Type*
    face_block_types,
const int* number_of_faces_in_blocks,
const int* face_global_ids,
const int* face_connectivity,
const double* face_lofting_factors,
int number_of_element_blocks,
const ContactSearch::ContactElement_Type*
    element_block_types,
const int* number_of_elements_in_blocks,
const int* element_global_ids,
const int* element_connectivity,
int number_of_nodal_comm_partners,
const int* nodal_comm_proc_ids,
const int* number_of_nodes_to_partner,
const int* communication_nodes,
const MPI_Comm& mpi_communicator,
ContactErrorCode& error );

C FORTRAN(build_search)(
    int* dimensionality,
    int* number_of_states,
    int* number_of_analytic_surfaces,
    int* number_of_node_blocks,
    int* node_block_types,
    int* number_of_nodes_in_blocks,
    int* node_exodus_ids,
    int* node_global_ids,
    double* coordinates,
    int* number_of_face_blocks,
    int* face_block_types,
    int* number_of_faces_in_blocks,
    int* face_global_ids,
    int* face_connectivity,
    double* face_lofting_factors,
    int* number_of_element_blocks,
    int* element_block_types,
    int* number_of_elements_in_blocks,
    int* element_global_ids,
    int* element_connectivity,
    int* number_of_nodal_comm_partners,
    int* nodal_comm_proc_ids,
    int* number_of_nodes_to_partner,
    int* communication_nodes,
    int* mpi_communicator,
    int* error );

Fortran build_search(
    dimensionality,
    number_of_states,
    number_of_analytic_surfaces,
    number_of_node_blocks,
node_block_types,
number_of_nodes_in_blocks,
node_exodus_ids,
node_global_ids,
coordinates,
number_of_face_blocks,
face_block_types,
number_of_faces_in_blocks,
face_global_ids,
face_connectivity,
face_lofting_factors,
number_of_element_blocks,
element_block_types,
number_of_elements_in_blocks,
element_global_ids,
element_connectivity,
number_of_nodal_comm_partners,
nodal_comm_proc_ids,
number_of_nodes_to_partner,
communication_nodes,
mpi_communicator,
error);

where:

dimensionality is the number of spatial coordinates in the topology. Note: We are only supporting three dimensions in this release. Two-dimensional support will be added in the future.

number_of_states is the number of states the host code requests to be stored. A value of 1 implies that the ContactSearch object can not back up to an older state. A value of 2 will imply the ContactSearch object can back up to one old state, etc. For this release, this value must be 1.

number_of_analytic_surfaces is the number of analytic surfaces that will be used.

number_of_node_blocks is the number of Node_Blocks in the topology. The first Node_Block contains the nodes connected to the faces specified in the Face_Blocks; additional Node_Blocks may contain other nodes that the host code needs to search against the faces.
	node_block_types is an array (of length number_of_node_blocks) describing the type of nodes in each Node_Block. The current enumeration for this type in the C++ API is:
	enum ContactSearch::ContactNode_Type{ NODE=1, POINT=2 };

number_of_nodes_in_blocks is an array (of length number_of_node_blocks) that gives the number of nodes in each Node_Block.

node_exodus_ids is an array (whose length is the total number of nodes in all Node_Blocks) containing the Exodus ID for each node. The IDs for the first Node_Block are listed first in the array, followed by the IDs for each of the other Node_Blocks in order (if applicable).

node_global_ids is an array (whose length is twice the total number of nodes in all Node_Blocks) containing the host code ID for each node. Each host code ID consists of two integers. For node N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. The IDs for the first Node_Block are listed first in the array, followed by the IDs for each of the other Node_Blocks in order (if applicable).

coordinates is an array (whose length is dimensionality*N where N is the total number of nodes in all Node_Blocks) containing the spatial coordinates for each node. For node k, the array index dimensionality*k is the x-coordinate, the array index dimensionality*k+1 is the y-coordinate, and array index dimensionality*k+2 is the z-coordinate.

number_of_face_blocks is the number of Face_Blocks in the topology.
Search Functions

face_block_types is an array (of length number_of_face_blocks) describing the type of faces in each Face_Block. The current enumeration for this type in the C++ API is:

```cpp
enum ContactSearch::ContactFace_Type{QUADFACEL4=1,
                       QUADFACEQ8=2, TRIFACEL3=3, TRIFACEQ6=4,
                       SHELLQUADFACEL4=5, SHELLTRIFACEL3=6};
```

number_of_faces_in_blocks is an array (of length number_of_face_blocks) that gives the number of faces in each Face_Block.

face_global_ids is an array (whose length is twice the total number of faces in all Face_BLOCKS) containing the host code ID for each face. Each host code ID consists of two integers. For face N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. The IDs for the first Face_Block are listed first in the array, followed by the IDs for each of the other Face_BLOCKS in order (if applicable).

face_connectivity is a one-dimensional array that gives the connectivity (using Fortran indexing in the first Node_Block) for each face. The connectivity of each face is contiguous in memory and follows the ExodusII conventions for node order. The arrangement of this array may change when multiple Node_BLOCKS containing nodes related to faces are supported. Note that if the faces of shells are included, then both sides of the shell must be given as faces: once with a clockwise numbering and once with a counter-clockwise numbering.

face_lofting_factors is a one-dimensional array that specifies the thickness weighting factors for the faces in the block. Note that the thickness for the two-faces of a shell should sum to one.

number_of_element_blocks is the number of Element_BLOCKS in the topology.

element_block_types is an array (of length number_of_element_blocks) describing the type of elements in each Element_Block. The current enumeration for this type in the C++ API is:

```cpp
enum ContactSearch::ContactElement_Type
{CARTESIANHEXELEMENTL8=1, HEXELEMENTL8=2};
```
	number_of_elements_in_blocks is an array (of length number_of_element_blocks) that gives the number of elements in each Element_Block.

element_global_ids is an array (whose length is twice the total number of elements in all Element_BLOCKS) containing the host code ID for each element. Each host code ID consists of two integers. For element N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. The IDs for the first Element_Block are listed first in the array, followed by the IDs for each of the other Element_BLOCKS in order (if applicable).

element_connectivity is a one-dimensional array that gives the connectivity (using Fortran indexing in the first Node_Block) for each element. The connectivity of each element is contiguous in memory and follows the ExodusII conventions for node order. The arrangement of this array may change when multiple Node_BLOCKS containing nodes related to elements are supported.

number_of_nodal_comm_partners is the number of processors that share nodes with the topology supplied to ACME on the current processor.

nodal_comm_proc_ids is an array (of length number_of_nodal_comm_partners) that lists the processor IDs that share nodes with the topology supplied to ACME on the current processor.

number_of_nodes_to_partner is an array (of length number_of_nodal_comm_partners) that gives the number of nodes shared with each processor in nodal_comm_proc_ids.

communication_nodes is an array that lists the nodes in the topology supplied to ACME that are shared, grouped by processor in the order specified in nodal_comm_proc_ids.

mpi_communicator is an MPI_Communicator.

t is the error code. This reflects any errors detected during execution of this method. A non-zero result indicates an error has occurred.

If the ACME library is built in pure serial mode (i.e., CONTACT_NO_MPI is defined during compilation), then number_of_nodal_comm_partners should be set to 0 and dum-
my pointers can be supplied for nodal_comm_proc_ids, number_of_nodes_to_partner, and communication_nodes. Furthermore, any integer value can be used for mpi_communicator, which is ignored.

3.2 Updating a Search Object

If the topology of the search object changes (due to either element birth/death or dynamic load balancing), then the search object’s topology can be updated with the interaction information preserved across the topology change. It is assumed that the dimensionality, number_of_states, number_of_entity_keys, and the number and type of entity blocks stays the same. *This capability currently does not work if shell face types are present.* The interface to update a search object’s topology is:

```cpp
void ContactSearch::UpdateSearch(
    const int* num_node_deaths_per_blocks,
    const int* node_death_global_ids,
    const int* num_face_deaths_per_block,
    const int* face_death_global_ids,
    const int* num_element_deaths_per_blocks,
    const int* element_death_global_ids,
    const int* num_node_births_per_block,
    const int* node_births_exodus_ids,
    const int* node_births_global_ids,
    const int* num_face_births_per_block,
    const int* face_births_global_ids,
    const int* face_births_connectivity,
    const int* num_element_births_per_block,
    const int* element_births_global_ids,
    const int* element_births_connectivity,
    const int number_of_exported_nodes,
    const int* exported_nodes_global_ids,
    const int* exported_nodes_proc_ids,
    const int number_of_exported_faces,
    const int* exported_faces_global_ids,
    const int* exported_faces_proc_ids,
    const int number_of_exported_elements,
    const int* exported_elements_global_ids,
    const int* exported_elements_proc_ids,
    const int number_of_nodes,
    const int* node_global_ids,
    const int number_of_faces,
    const int* face_global_ids,
    const int number_of_elements,
    const int* element_global_ids,
    const int number_of_nodal_comm_partners,
    const int* nodal_comm_proc_ids,
    const int* number_of_nodes_to_partner,
    const int* communication_nodes,
    ContactErrorCode& error );
```

```fortran
FORTRAN(update_search)(
    int* num_node_deaths_per_blocks,
```
Search Functions

```c
int* node_death_global_ids,
int* num_face_deaths_per_block,
int* face_death_global_ids,
int* num_element_deaths_per_blocks,
int* element_death_global_ids,
int* num_node_births_per_block,
int* node_births_exodus_ids,
int* node_births_global_ids,
int* num_face_births_per_block,
int* face_births_global_ids,
int* face_births_connectivity,
int* num_element_births_per_block,
int* element_births_global_ids,
int* element_births_connectivity,
int& number_of_exported_nodes,
int* exported_nodes_global_ids,
int* exported_nodes_proc_ids,
int& number_of_exported_faces,
int* exported_faces_global_ids,
int* exported_faces_proc_ids,
int& number_of_exported_elements,
int* exported_elements_global_ids,
int* exported_elements_proc_ids,
int& number_of_nodes,
int* node_global_ids,
int& number_of_faces,
int* face_global_ids,
int& number_of_elements,
int* element_global_ids,
int* number_of_nodal_comm_partners,
int* nodal_comm_proc_ids,
int* number_of_nodes_to_partner,
int* communication_nodes,
int* error );
```

**Fortran**

```fortran
update_search(
    num_node_deaths_per_blocks,
    node_death_global_ids,
    num_face_deaths_per_block,
    face_death_global_ids,
    num_element_deaths_per_blocks,
    element_death_global_ids,
    num_node_births_per_block,
    node_births_exodus_ids,
    node_births_global_ids,
    num_face_births_per_block,
    face_births_global_ids,
    face_births_connectivity,
    num_element_births_per_block,
    element_births_global_ids,
    element_births_connectivity,
    number_of_exported_nodes,
    exported_nodes_global_ids,
    exported_nodes_proc_ids,
```

78
number_of_exported_faces,
exported_faces_global_ids,
exported_faces_proc_ids,
number_of_exported_elements,
exported_elements_global_ids,
exported_elements_proc_ids,
number_of_nodes,
nodes_global_ids,
number_of_faces,
face_global_ids,
number_of_elements,
element_global_ids,
number_of_nodal_comm_partners,
nodal_comm_proc_ids,
number_of_nodes_to_partner,
communication_nodes,
error);

where:

num_node_deaths_per_block is an array (of length number_of_node_blocks) that gives the number of nodes that died in each Node_Block. This is only for nodes of type POINT. Nodes of type NODE are automatically killed if they have to connect (faces or elements) after faces and elements are killed.
	node_death_global_ids is an array (whose length is the total number of nodes in the num_node_deaths_per_block array) containing the global ID for each node that is to be killed.

num_face_deaths_per_block is an array (of length number_of_face_blocks) that gives the number of faces that died in each Face_Block.

face_death_global_ids is an array (whose length is the total number of faces in the num_face_deaths_per_block array) containing the global ID for each face that is to be killed.

num_element_deaths_per_block is an array (of length number_of_element_blocks) that gives the number of faces that died in each Element_Block.

element_death_global_ids is an array (whose length is the total number of elements in the num_element_deaths_per_block array) containing the global ID for each element that is to be killed.

num_node_births_per_block is an array (of length number_of_node_blocks) that gives the number of nodes that are to be added in each Node_Block.

node_births_exodus_ids is an array (whose length is the total number of nodes in the num_node_births_per_block array) containing the Exodus ID for each node that is to be added.

node_births_global_ids is an array (whose length is the total number of nodes in the num_node_births_per_block array) containing the global ID for each node that is to be added.

num_face_births_per_block is an array (of length number_of_face_blocks) that gives the number of faces that are to be added in each Face_Block.

face_births_global_ids is an array (whose length is the total number of faces in the num_face_births_per_block array) containing the global ID for each face that is to be added.

face_births_connectivity is a one-dimensional array that gives the connectivity (using global IDs) for each face. The connectivity of each face is contiguous in memory and follows the ExodusII conventions for node order.
Search Functions

num_element_births_per_block is an array (of length number_of_element_blocks) that gives the number of elements that are to be added in each Element_Block.

element_births_global_ids is an array (whose length is the total number of elements in the num_element_births_per_block array) containing the global ID for each element that is to be added.

element_births_connectivity is a one-dimensional array that gives the connectivity (using global IDs) for each face. The connectivity of each face is contiguous in memory and follows the ExodusII conventions for node order.

number_of_exported_nodes is the number of nodes that are in both the old and new topology but reside on a different processor in the new topology. This is only applicable for dynamic load balancing. Note: For this release, dynamic load balancing is not yet supported.

exported_nodes_host_ids is an array (of length 2*number_of_exported_nodes) containing the host code ID for each node that is in both the old and new topology but resides on a different processor in the new topology. Each host code ID consists of two integers. For node N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. This is only applicable for dynamic load balancing.

exported_nodes_proc_ids is an array (of length number_of_exported_nodes) containing the processor ID of the processor that contains the node in the new topology. This is only applicable for dynamic load balancing.

number_of_exported_faces is the number of faces that are in both the old and new topology but reside on a different processor in the new topology. This is only applicable for dynamic load balancing. Note: For this release, dynamic load balancing is not yet supported.

exported_faces_host_ids is an array (of length 2*number_of_exported_faces) containing the host code ID for each face that is in both the old and new topology but resides on a different processor in the new topology. Each host code ID consists of two integers. For face N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. This is only applicable for dynamic load balancing.

exported_faces_proc_ids is an array (of length number_of_exported_faces) containing the processor ID of the processor that contains the face in the new topology. This is only applicable for dynamic load balancing.

number_of_exported_elements is the number of elements that are in both the old and new topology but reside on a different processor in the new topology. This is only applicable for dynamic load balancing. Note: For this release, dynamic load balancing is not yet supported.

exported_elements_host_ids is an array (of length 2*number_of_exported_elements) containing the host code ID for each element that is in both the old and new topology but resides on a different processor in the new topology. Each host code ID consists of two integers. For element N, the array index 2*N contains the most significant word and the index 2*N+1 contains the least significant word. This is only applicable for dynamic load balancing.

exported_elements_proc_ids is an array (of length number_of_exported_elements) containing the processor ID of the processor that contains the element in the new topology. This is only applicable for dynamic load balancing.

number_of_nodes is the total number of nodes in the new updated topology.

node_global_ids is an array (whose length is twice the total number of nodes in all Node_Blocks) containing the host code ID for each node in the updated topology. This array is used to provide a mapping between the host code’s ordering of nodes and ACME’s internal ordering of nodes.

number_of_faces is the total number of faces in the new updated topology.

face_global_ids is an array (whose length is twice the total number of faces in all Face_Blocks) containing the host code ID for each face in the updated topology. This array is used to provide a mapping between the host code’s ordering of faces and ACME’s internal ordering of faces.

number_of_elements is the total number of elements in the new updated topology.

element_global_ids is an array (whose length is twice the total number of elements in all Element_Blocks) containing the host code ID for each element in the updated topology.
This array is used to provide a mapping between the host code’s ordering of elements and ACME’s internal ordering of elements.

number_of_nodal_comm_partners is the number of processors that share nodes with the topology supplied to ACME on the current processor.
	nodal_comm_proc_ids is an array (of length number_of_nodal_comm_partners) that lists the processor IDs that share nodes with the topology supplied to ACME on the current processor.

number_of_nodes_to_partner is an array (of length number_of_nodal_comm_partners) that gives the number of nodes shared with each processor in nodal_comm_proc_ids.

communication_nodes is an array that lists the nodes in the topology supplied to ACME that are shared, grouped by processor in the order specified in nodal_comm_proc_ids.

error is the error code. This reflects any errors detected during execution of this method. A non-zero result indicates an error has occurred.

If the ACME library is built in pure serial mode (i.e., CONTACT_NO_MPI is defined during compilation), then number_of_nodal_comm_partners should be set to 0 and dummy pointers can be supplied for nodal_comm_proc_ids, number_of_nodes_to_partner, and communication_nodes.

3.3 Search_Data Array

As described in Section 1.1.6, Search_Data is a three-dimensional Fortran-ordered array for specifying entity pair data. The first index in the array refers to the data parameter, and the next two indexes refer to the keys for the two entities for which that parameter is applicable.

3.3.1 Checking the Search_Data Array Size

The following interface allows for checking the size of Search_Data expected by ACME. This is intended to be a check by the host code to ensure that ACME and the host code have a consistent view of the Search_Data array.

```
C++ ContactErrorCode ContactSearch::Check_Search_Data_Size(
    int size_data_per_pair,
    int number_of_entity_keys );

C FORTRAN(check_search_data_size)(
    int* size_data_per_pair,
    int* number_of_entity_keys,
    int* error );

Fortran check_search_data_size(
    size_data_per_pair,
    number_of_entity_keys,
    error )
```

where

size_data_per_pair is the number of data parameters for each entity pair (currently 3).

number_of_entity_keys is the number of entity keys.

error is the return error code for the C and Fortran APIs.
Search Functions

3.3.2 Setting Values in the Search_Data Array

The following interface allows the host code to specify the Search_Data array (see Section 1.1.6), which must be set prior to calling any of the search algorithms. This function can be called at any time to change values in the Search_Data array (e.g., to change tolerances).

```c++
void ContactSearch::Set_Search_Data(
    const double* search_data);
```

```c
FORTRAN(set_search_data)(
    double* search_data );
```

```fortran
set_search_data(
    search_data )
```

where

`search_data` is an array of double precision values for the Search_Data (see Section 1.1.6).

3.4 Analytic_Surfaces

ACME supports the determination of interactions of nodes with Analytic_Surfaces. Currently, the only supported Analytic_Surfaces are a plane, a sphere, and two types of cylinders (one for a container and one for a post). The types of Analytic_Surfaces supported will be expanded in the future. The ACME ID for an Analytic_Surface is the number of face blocks plus the order in which the surface was created.

The current enumeration for Analytic_Surface_Type in the C++ API is:

```c++
enum ContactSearch::AnalyticSurface_Type{
    PLANE=1, SPHERE=2, CYLINDER_INSIDE=3, CYLINDER_OUTSIDE=4 };
```

3.4.1 Adding an Analytic_Surface

The interface to add an Analytic_Surface is:

```c++
ContactErrorCode ContactSearch::Add_Analytic_Surface(
    ContactSearch::AnalyticSurface_Type as_type,
    const double* as_data );
```

```c
FORTRAN(add_analytic_surface)(
    int* as_type,
    double* as_data,
    int* error );
```

```fortran
add_analytic_surface(
    as_type,
    as_data,
    error )
```
where

`as_type` is the type of the analytic surface from the `ContactSearch::AnalyticSurface_Type` enum.

`as_data` is an array dependent on the type of surface being added. The `Analytic_Surface` PLANE is described by a point and a normal vector. The `Analytic_Surface` SPHERE is described by its center and a radius. Two types of cylindrical surfaces are supported: `CYLINDER_INSIDE` & `CYLINDER_OUTSIDE`. `CYLINDER_INSIDE` is intended as a cylindrical container which will define interactions to keep all nodes inside the cylinder. `CYLINDER_OUTSIDE` is intended as a post which will define interactions to keep all nodes outside the cylinder. Both types of cylindrical surfaces are described by a center point, an axial direction, and a length (see Figure 14). Table 14 gives a complete description of the array data for each `Analytic_Surface` type.

`error` is the return error code for the C and Fortran APIs.

![Figure 14 Analytic Cylindrical Surfaces](image)

### Table 14 C++ Data Description for Analytic_Surfaces

<table>
<thead>
<tr>
<th>as_data[0]</th>
<th>Plane</th>
<th>Sphere</th>
<th>Cylinder_Inside</th>
<th>Cylinder_Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>as_data[0]</code></td>
<td>X-Coordinate of Point</td>
<td>X-Coordinate of Center</td>
<td>X-Coordinate of Center</td>
<td>X-Coordinate of Center</td>
</tr>
<tr>
<td><code>as_data[1]</code></td>
<td>Y-Coordinate of Point</td>
<td>Y-Coordinate of Center</td>
<td>Y-Coordinate of Center</td>
<td>Y-Coordinate of Center</td>
</tr>
<tr>
<td><code>as_data[2]</code></td>
<td>Z-Coordinate of Point</td>
<td>Z-Coordinate of Center</td>
<td>Z-Coordinate of Center</td>
<td>Z-Coordinate of Center</td>
</tr>
</tbody>
</table>
3.4.2 Setting the Analytic_Surface Configuration

The following interface updates the configuration(s) for an Analytic_Surface.

\begin{verbatim}
C++
ContactErrorCode
 ContactSearch::Set_Analytic_Surface_Configuration(
    int as_id,
    const double* as_data );

C
FORTRAN(set_analytic_surface_configuration)(
    int* as_id,
    double* as_data,
    int* error );

Fortran
set_analytic_surface_configuration(
    as_id,
    as_data,
    error )
\end{verbatim}

where

- as_id is the ACME ID for the Analytic_Surface.
- as_data is described in Table 14.
- error is the return error code for the C and Fortran APIs.

3.5 Node_Block Data

Currently, valid Node_Block types are NODE and POINT (see Section 1.1.1).

<table>
<thead>
<tr>
<th></th>
<th>Plane</th>
<th>Sphere</th>
<th>Cylinder_Inside</th>
<th>Cylinder_Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>as_data[6]</td>
<td></td>
<td>Radius</td>
<td>Radius</td>
<td></td>
</tr>
<tr>
<td>as_data[7]</td>
<td></td>
<td>Length</td>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

Table 14 C++ Data Description for Analytic_Surfaces
3.5.1 Setting the Node_Block Configuration

The following interface allows the host code to specify the configuration(s) for the nodes by Node_Block. This function can be called at any time but must be called prior to the first search. For a one-configuration search, only the current configuration needs to be specified. For two-configuration searches, both current and predicted configurations must be specified. This function should be called every time the nodal positions in the host code are updated. The prototype for this function is:

```cpp
ContactErrorCode ContactSearch::Set_Node_Block_Configuration(
    ContactSearch::ContactNode_Configuration config_type,
    int node_block_id,
    const double* positions );
```

```c
FORTRAN(set_node_block_configuration)(
    int* config_type,
    int* node_block_id,
    double* positions,
    int* error );
```

```fortran
set_node_block_configuration(
    config_type,
    node_block_id,
    positions,
    error )
```

where:

- `config_type` is an enumeration in the C++ API for the configuration:
  ```cpp
  enum ContactSearch::ContactNode_Configuration {
      CURRENT_CONFIG=1, PREDICTED_CONFIG=2
  };
  ```
- `node_block_id` is the ACME ID for the Node_Block.
- `positions` is an array that holds the nodal positions for every node in the Node_Block. The data in this array is ordered by x, y and z locations of node 1, followed by x, y and z locations of node 2, etc.
- `error` is the return error code for the C and Fortran APIs.

3.5.2 Setting the Node_Block Kinematic Constraints

The following function informs the ContactSearch object about kinematic constraints for the nodes. If these are specified, the interactions are made consistent with the constraints. Also, the ContactTDEnforcement object computes contact forces that are consistent with these constraints.

```cpp
ContactErrorCode ContactSearch::Set_Node_Block_Kinematic_Constraints(
    int node_block_id,
    const int* constraints_per_node,
    const double* constraint_vector );
```
Search Functions

\[ \text{C} \quad \text{FORTRAN(set_node_block_kin_cons)} ( \]
\[ \text{int* node_block_id,} \]
\[ \text{int* constraints_per_node,} \]
\[ \text{double* constraint_vector,} \]
\[ \text{int* error });} \]

\[ \text{Fortran set_node_block_kin_cons(} \]
\[ \text{node_block_id,} \]
\[ \text{constraints_per_node,} \]
\[ \text{constraint_vector,} \]
\[ \text{error )} \]

where

node_block_id is the ACME ID for this Node_Block
constraints_per_node is how many degrees of freedom are constrained (i.e., 0, 1, 2 or 3).
constraint_vector is a vector for each node that describes the constraint direction. If
constraints_per_node is 0 or 3, this vector should be set to 0. If constraints_per_node is 1,
this vector should be the constrained direction. If constraints_per_node is 2, this vector
should be the unconstrained direction.
error is the return error code for the C and Fortran APIs.

3.5.3 Setting the Node_Block Attributes

The following function will be used to add Node_Block attributes, such as the projection
direction for all node types or the radius for POINT nodes.

\[ \text{C++ ContactErrorCode} \]
\[ \text{ContactSearch::Set_Node_Block_Attributes(} \]
\[ \text{ContactSearch::Node_Block_Attribute attribute,} \]
\[ \text{int node_block_id,} \]
\[ \text{const double* attributes );} \]

\[ \text{C FORTRAN(set_node_block_attributes)} ( \]
\[ \text{int* attribute,} \]
\[ \text{int* node_block_id,} \]
\[ \text{double* attributes,} \]
\[ \text{int* error });} \]

\[ \text{Fortran set_node_block_attributes(} \]
\[ \text{attribute,} \]
\[ \text{node_block_id,} \]
\[ \text{attributes,} \]
\[ \text{error )} \]

where

attribute is an enumeration in the C++ API for the attribute type:
\[ \text{enum ContactSearch::Node_Block_Attribute {} \]
\[ \text{PROJECTION_DIRECTION=0, RADIUS=1);} \]
node_block_id is the ACME ID for this Node_Block.
attributes is an array of the attribute values for this Node_Block.
error is the return error code for the C and Fortran APIs.
3.6 Face_Block Data

3.6.1 Setting the Face_Block Attributes

The following function will be used to add Face_Block attributes. Currently, the only attributes supported is the thickness and lofting factor for shell face types.

```cpp
ContactErrorCode
ContactSearch::Set_Face_Block_Attributes(
    ContactSearch::Face_Block_Attribute attribute,
    int face_block_id,
    const double* attributes);
```

```c
FORTRAN(set_face_block_attributes)(
    int* attribute,
    int* face_block_id,
    double* attributes,
    int* error);
```

```fortran
set_face_block_attributes(
    attribute,
    face_block_id,
    attributes,
    error)
```

where

- `attribute` is an enumeration in the C++ API for the attribute type:
  ```
  enum ContactSearch::Face_Block_Attribute {
    SHELL_THICKNESS=1, LOFTING_FACTOR=2);
  ```
- `face_block_id` is the ACME ID for this Face_Block.
- `attributes` is an array of the attribute values for this Face_Block.
- `error` is the return error code for the C and Fortran APIs.

3.7 Table Data

The following function will be used to add a generic table of x-y data pairs. Currently, the only use of tables is in some of the explicit transient dynamic friction models used during enforcement.

```cpp
ContactErrorCode
ContactSearch::Add_Table(
    int id,
    int number_of_points,
    double* abscissa,
    double* ordinate);
```

```c
FORTRAN(add_table)(
    int* id,
    int* number_of_points,
    double* abscissa,
    double* ordinate,
```
Search Functions

```
int* error);
```

**Fortran**
```
add_table(
id,
number_of_points,
abscissa,
ordinate
error)
```

where

- *id* is the ACME ID for the table.
- *number_of_points* is the number of entries in the table.
- *abscissa* is an array of the x-values for this table.
- *ordinate* is an array of the y-values for this table.
- *error* is the return error code for the C and Fortran APIs.

### 3.8 Search Algorithms

#### 3.8.1 Setting the Search Option

By default, both multiple interactions and normal smoothing options are inactive. The following function should be called to activate, deactivate, and control multiple interactions, normal smoothing, and node area computation.

**C++**
```
ContactErrorCode ContactSearch::Set_Search_Option(
   ContactSearch::Search_Option option,
   ContactSearch::Search_Option_Status status,
   double* data);
```

**C**
```
FORTRAN(set_search_option)(
   int* option,
   int* status,
   double* data,
   int* error);
```

**Fortran**
```
set_search_option(
   option,
   status,
   data,
   error)
```

where

- *option* is an enumeration in the C++ API:
  ```
  enum ContactSearch::Search_Option {
    MULTIPLE_INTERACTIONS=0,
    NORMAL_SMOOTHING=1,
    COMPUTE_NODE_AREAS=2
  };
  ```

- *status* is another enumeration:
  ```
  enum ContactSearch::Search_Option_Status {
    INACTIVE=0,
    ACTIVE=1
  };
  ```
data is an array whose first member contains the angle above which the edge between faces is considered to be sharp instead of non-sharp (rounded), and whose second and third members (valid only for the NORMAL_SMOOTHING option) contain the distance in isoparametric coordinates over which normal smoothing is calculated and the smoothing resolution algorithm, respectively. The integer specifying the smoothing resolution algorithm can take the values USE_NODE_NORMAL=0 or USE_EDGE_BASED_NORMAL=1. See the section on Normal Smoothing in the introduction for more information about the smoothing resolution algorithm.

error is the return error code for the C and Fortran APIs.

### 3.8.2 Performing a Static 1-Configuration Search

The following function performs a static 1-configuration search and can be called only after a current configuration has been specified.

```cpp
C++
ContactErrorCode
ContactSearch::Static_Search_1_Configuration();
```

```c
C
FORTRAN(static_search_1_configuration)(
    int* error );
```

```fortran
Fortran static_search_1_configuration(
    error );
```

where

```
error is the return error code for the C and Fortran APIs.
```

### 3.8.3 Performing a Static 2-Configuration Search

The following function performs a static 2-configuration search and can be called only if both current and predicted configurations have been specified.

```cpp
C++
ContactErrorCode
ContactSearch::Static_Search_2_Configuration();
```

```c
C
FORTRAN(static_search_2_configuration)(
    int* error );
```

```fortran
Fortran static_search_2_configuration(
    error )
```

where

```
error is the return error code for the C and Fortran APIs.
```

### 3.8.4 Performing a Dynamic 2-Configuration Search

The following function performs a dynamic 2-configuration search and can be called only if both the current and predicted configurations have been specified. Please note due to the dependence of this search on relative velocity, all the interactions within the normal search tolerance may not be returned. Specifically, those leaving contact may be omitted.
Search Functions

**C++**
```
ContactErrorCode
ContactSearch::Dynamic_Search_2_Configuration();
```

**C**
```
FORTRAN(dynamic_search_2_configuration)(
    int* error );
```

**Fortran**
```
dynamic_search_2_configuration(
    error )
```

where

error is the return error code for the C and Fortran APIs.

### 3.8.5 Performing a Dynamic Augmented 2-Configuration Search

The following function performs a dynamic augmented 2-configuration search and can be called only if both the current and predicted configurations have been specified and a ContactTDEnforcement object has been registered with the search. Please note due to the dependence of this search on relative velocity, all the interactions within the normal search tolerance may not be returned. Specifically, those leaving contact may be omitted.

**C++**
```
ContactErrorCode
ContactSearch::Dynamic_Search_Augmented_2_Configuration(
    double* mass,
    double dt_old,
    double dt );
```

**C**
```
FORTRAN(dynamic_search_aug_2_config)(
    double* mass,
    double* dt_old,
    double* dt,
    int* error );
```

**Fortran**
```
dynamic_search_aug_2_config(
    mass,
    dt_old,
    dt,
    error )
```

where

mass is an array that gives the mass of each node.
dt_old is the time step for the previous step.
dt is the time step for the current time step.
error is the return error code for the C and Fortran APIs.

### 3.9 Interactions

The functions in this section allow the host code to extract the interactions from the ContactSearch object. Typically, the host code should first determine how much memory is needed to hold the interactions before extracting the interactions. *ACME does not currently support the extraction of interactions involving shells.*
3.9.1 Getting the Size of NodeFace_Interactions

The following function allows the host code to determine how many NodeFace_Interactions are currently defined in a ContactSearch object and the data size for each interaction.

C++
```cpp
void ContactSearch::Size_NodeFace_Interactions(
    int& number_of_interactions,
    int& nfi_data_size);
```

C
```fortran
FORTRAN(size_nodeface_interactions)(
    int* number_of_interactions,
    int* nfi_data_size);
```

Fortran
```fortran
size_nodeface_interactions(
    number_of_interactions,
    nfi_data_size)
```

where

- `number_of_interactions` is the number of active NodeFace_Interactions that will be returned by the function Get_NodeFace_Interactions (see section 3.9.2).
- `nfi_data_size` is the number of double precision values returned for each interaction.

3.9.2 Extracting NodeFace_Interactions

The following function allows the host code to extract the active NodeFace_Interactions from the ContactSearch object.

C++
```cpp
void ContactSearch::Get_NodeFace_Interactions(
    int* node_block_ids,
    int* node_indexes_in_block,
    int* node_entity_keys,
    int* face_block_ids,
    int* face_indexes_in_block,
    int* face_procs,
    double* nfi_data);
```

C
```fortran
FORTRAN(get_nodeface_interactions)(
    int* node_block_ids,
    int* node_indexes_in_block,
    int* node_entity_keys,
    int* face_block_ids,
    int* face_indexes_in_block,
    int* face_procs,
    double* nfi_data);
```

Fortran
```fortran
get_nodeface_interactions(
    node_block_ids,
    node_indexes_in_block,
    node_entity_keys,
    face_block_ids,
```
Search Functions

\[
\text{face\_indexes\_in\_block, face\_procs, nfi\_data}
\]

where

- \(\text{node\_block\_ids}\) is an array (of length \(\text{number\_of\_interactions}\)) that contains the Node\_Block ID for the node in each interaction.
- \(\text{node\_indexes\_in\_block}\) is an array (of length \(\text{number\_of\_interactions}\)) that contains the index in the Node\_Block (using Fortran indexing conventions) for the node in each interaction.
- \(\text{node\_entity\_keys}\) is an array (of length \(\text{number\_of\_interactions}\)) that contains the node entity key for this interaction.
- \(\text{face\_block\_ids}\) is an array (of length \(\text{number\_of\_interactions}\)) that contains the Face\_Block ID for the face in each interaction.
- \(\text{face\_indexes\_in\_block}\) is an array (of length \(\text{number\_of\_interactions}\)) that contains the index in the Face\_Block (using Fortran indexing conventions) for the face in each interaction.
- \(\text{face\_procs}\) is an array (of length \(\text{number\_of\_interactions}\)) that contains the processor that owns the face in each interaction.
- \(\text{nfi\_data}\) is an array (of length \(\text{number\_of\_interactions}\*\text{nfi\_data\_size}\)) that contains the data for each interaction (see Section 1.3.1). The data for each interaction is contiguous (i.e., the first \(\text{nfi\_data\_size}\) locations contain the data for the first interaction).

### 3.9.3 Getting the Size of NodeSurface\_Interactions

The following function allows the host code to determine how many interactions are currently defined in a ContactSearch object and the data size for each interaction.

\[
\begin{align*}
\text{C}++ & \quad \text{void ContactSearch::Size\_NodeSurface\_Interactions(} \\
& \quad \quad \text{int\& number\_of\_interactions,} \\
& \quad \quad \text{int\& nsi\_data\_size);}
\end{align*}
\]

\[
\begin{align*}
\text{C} & \quad \text{FORTRAN(size\_nodesurface\_interactions)(} \\
& \quad \quad \text{int* number\_of\_interactions,} \\
& \quad \quad \text{int* nsi\_data\_size);}
\end{align*}
\]

\[
\begin{align*}
\text{Fortran} & \quad \text{size\_nodesurface\_interactions(} \\
& \quad \quad \text{number\_of\_interactions,} \\
& \quad \quad \text{nsi\_data\_size)}
\end{align*}
\]

where

- \(\text{number\_of\_interactions}\) is the number of active NodeSurface\_Interactions that will be returned by the function Get\_NodeSurface\_Interactions (see section 3.9.4).
- \(\text{nsi\_data\_size}\) is the number of double precision values returned for each interaction.

### 3.9.4 Extracting NodeSurface\_Interactions

The following function allows the host code to extract the active NodeSurface\_Interactions from the ContactSearch object.
C++    void ContactSearch::Get_NodeSurface_Interactions(
        int* node_block_ids,
        int* node_indexes_in_block,
        int* analyticsurface_ids,
        double* nsi_data );

C     FORTRAN(get_nodesurface_interactions)(
        int* node_block_ids,
        int* node_indexes_in_block,
        int* analyticsurface_ids,
        double* nsi_data );

Fortran  get_nodesurface_interactions(
        node_block_ids,
        node_indexes_in_block,
        analyticsurface_ids,
        nsi_data )

where

node_block_ids is an array (of length number_of_interactions) that contains the Node_Block ID for
the node in each interaction.
node_indexes_in_block is an array (of length number_of_interactions) that contains the index in
the Node_Block (using Fortran indexing conventions) for the node in each interaction.
analyticsurface_ids is an array (of length number_of_interactions) that contains the ID of the
Analytic_Surface for each interaction.
nsi_data is an array (of length number_of_interactions*nsi_data_size) that contains the data for
each interaction (see Section 1.3.2). The data for each interaction is contiguous (i.e., the
first nsi_data_size locations contain the data for the first interaction).

3.9.5 Getting the Size of NodeNode_Interactions

The following function allows the host code to determine how many interactions are cur-
rently defined in a ContactSearch object and the data size for each interaction.

C++    void ContactSearch::Size_NodeNode_Interactions(
        int& number_of_interactions,
        int& nni_data_size );

C     FORTRAN(size_nodenode_interactions)(
        int* number_of_interactions,
        int* nni_data_size );

Fortran  size_nodenode_interactions(
        number_of_interactions,
        nni_data_size )

where

number_of_interactions is the number of active NodeNode_Interactions that will be returned by the
function Get_NodeNode_Interactions (see section 3.9.6).
3.9.6 Extracting NodeNode_Interactions

The following function allows the host code to extract the active NodeNode_Interactions from the ContactSearch object.

**C++**
```cpp
void ContactSearch::Get_NodeNode_Interactions(
    int* slave_node_block_ids,
    int* slave_node_indexes_in_block,
    int* master_node_block_ids,
    int* master_node_indexes_in_block,
    int* master_node_procs,
    double* nni_data );
```

**C**
```c
FORTRAN(get_nodenode_interactions)(
    int* slave_node_block_ids,
    int* slave_node_indexes_in_block,
    int* master_node_block_ids,
    int* master_node_indexes_in_block,
    int* master_node_procs,
    double* nni_data );
```

**Fortran**
```fortran
get_nodenode_interactions(
    slave_node_block_ids,
    slave_node_indexes_in_block,
    master_node_block_ids,
    master_node_indexes_in_block,
    master_node_procs,
    nni_data )
```

where

- `slave_node_block_ids` is an array (of length `number_of_interactions`) that contains the Node_Block ID for the slave node in each interaction.
- `slave_node_indexes_in_block` is an array (of length `number_of_interactions`) that contains the index in the Node_Block (using Fortran indexing conventions) for the slave node in each interaction.
- `master_node_block_ids` is an array (of length `number_of_interactions`) that contains the Node_Block ID for the master node in each interaction.
- `master_node_indexes_in_block` is an array (of length `number_of_interactions`) that contains the index in the Node_Block (using Fortran indexing conventions) for the master node in each interaction.
- `master_node_procs` is an array (of length `number_of_interactions`) that contains the processor that owns the master_node in each interaction.
- `nni_data` is an array (of length `number_of_interactions*nni_data_size`) that contains the data for each interaction (see Section 1.3.3). The data for each interaction is contiguous (i.e., the first `nni_data_size` locations contain the data for the first interaction).
3.9.7 Getting the Size of FaceFace_Interactions

The following function allows the host code to determine how many FaceFace_Interactions are currently defined in a ContactSearch object and the data size for all interactions.

```cpp
void ContactSearch::Size_FaceFace_Interactions(
   int& number_of_interactions,
   int& ffi_data_size );
```

```fortran
size_faceface_interactions(
   number_of_interactions,
   ffi_data_size )
```

where

- `number_of_interactions` is the number of active FaceFace_Interactions that will be returned by the function `Get_FaceFace_Interactions` (see section 3.9.8).
- `ffi_data_size` is the number of double precision values returned for the entire set of FaceFace_Interactions.

3.9.8 Extracting FaceFace_Interactions

The following function allows the host code to extract the active FaceFace_Interactions from the ContactSearch object.

```cpp
void ContactSearch::Get_FaceFace_Interactions(
   int* slave_face_block_ids,
   int* slave_face_indexes_in_block,
   int* master_face_block_ids,
   int* master_face_indexes_in_block,
   int* master_face_procs,
   int* ffi_index,
   double* ffi_data );
```

```fortran
get_faceface_interactions(
   slave_face_block_ids,
   slave_face_indexes_in_block,
```
Search Functions

```plaintext
master_face_block_ids,
master_face_indexes_in_block,
master_face_procs,
ffi_index,
ffi_data)
```

where

slave_face_block_ids is an array (of length number_of_interactions) that contains the Face_Block ID for the slave face in each interaction.
slave_face_indexes_in_block is an array (of length number_of_interactions) that contains the index in the Face_Block (using Fortran indexing conventions) for the slave face in each interaction.
master_face_block_ids is an array (of length number_of_interactions) that contains the Face_Block ID for the master face in each interaction.
master_face_indexes_in_block is an array (of length number_of_interactions) that contains the index in the Face_Block (using Fortran indexing conventions) for the master face in each interaction.
master_face_procs is an array (of length number_of_interactions) that contains the processor that owns the master_face in each interaction.
ffi_index is an array (of length number_of_interactions) that contains the offset into the ffi_data array for the data for each interaction (i.e., the data for interaction j begins at ffi_data[ffi_index[j]]).
ffi_data is an array (of length ffi_data_size) that contains the data for each interaction (see Section 1.3.4). The data for each interaction is contiguous.

### 3.9.9 Getting the Size of FaceCoverage_Interactions

The following function allows the host code to determine how many FaceCoverage_Interactions are currently defined in a ContactSearch object and the data size for all the interactions.

```c
void ContactSearch::Size_FaceCoverage_Interactions(
    int& number_of_interactions,
    int& fci_data_size );
```

```c
FORTRAN(size_facecoverage_interactions)(
    int* number_of_interactions,
    int* fci_data_size );
```

```fortran
size_facecoverage_interactions(
    number_of_interactions,
    fci_data_size )
```

where

number_of_interactions is the number of active FaceCoverage_Interactions that will be returned by the function Get_FaceCoverage_Interactions (see section 3.9.10).
fci_data_size is the number of double precision values returned for the entire set of FaceCoverage_Interactions.
3.9.10 Extracting FaceCoverage_Interactions

The following function allows the host code to extract the active FaceCoverage_Interactions from the ContactSearch object.

\[
\begin{align*}
\text{C++} & \quad \text{void ContactSearch::Get_FaceCoverage_Interactions(} \\
& \quad \quad \quad \text{int* face_block_ids,} \\
& \quad \quad \quad \text{int* face_indexes_in_block,} \\
& \quad \quad \quad \text{int* fci_index,} \\
& \quad \quad \quad \text{double* fci_data );} \\
\text{C} & \quad \text{FORTRAN(get_facecoverage_interactions)}( \\
& \quad \quad \quad \text{int* face_block_ids,} \\
& \quad \quad \quad \text{int* face_indexes_in_block,} \\
& \quad \quad \quad \text{int* fci_index,} \\
& \quad \quad \quad \text{double* fci_data );} \\
\text{Fortran} & \quad \text{get_facecoverage_interactions(} \\
& \quad \quad \quad \text{face_block_ids,} \\
& \quad \quad \quad \text{face_indexes_in_block,} \\
& \quad \quad \quad \text{fci_index,} \\
& \quad \quad \quad \text{fci_data );}
\end{align*}
\]

where

- face_block_ids is an array (of length number_of_interactions) that contains the Face_Block ID for the face in each interaction.
- face_indexes_in_block is an array (of length number_of_interactions) that contains the index in the Face_Block (using Fortran indexing conventions) for the face in each interaction.
- fci_index is an array (of length number_of_interactions) that contains the offset into the fci_data array for the data for each interaction (i.e., the data for interaction j begins at fci_data[fci_index[j]]).
- fci_data is an array (of length fci_data_size) that contains the data for each interaction (see Section 1.3.5). The data for each interaction is contiguous.

3.9.11 Getting the Size of ElementElement_Interactions

The following function allows the host code to determine how many ElementElement_Interactions are currently defined in a ContactSearch object and the data size for all interactions.

\[
\begin{align*}
\text{C++} & \quad \text{void ContactSearch::Size_ElementElement_Interactions(} \\
& \quad \quad \quad \text{int& number_of_interactions,} \\
& \quad \quad \quad \text{int& eei_data_size );} \\
\text{C} & \quad \text{FORTRAN(size_elementelement_interactions)}( \\
& \quad \quad \quad \text{int* number_of_interactions,} \\
& \quad \quad \quad \text{int* eei_data_size );} \\
\text{Fortran} & \quad \text{size_elementelement_interactions(}
\end{align*}
\]
number_of_interactions,  
eei_data_size )

where

number_of_interactions is the number of active Element-Element_Interactions that will be returned by the function Get_ElementElement_Interactions (see section 3.9.12).
eei_data_size is the number of double precision values returned for each interaction.

### 3.9.12 Extracting Element-Element_Interactions

The following function allows the host code to extract the active Element-Element_Interactions from the ContactSearch object.

**C++**
```cpp
void ContactSearch::Get_ElementElement_Interactions(
    int* slave_element_block_ids,
    int* slave_element_indexes_in_block,
    int* master_element_block_ids,
    int* master_element_indexes_in_block,
    int* master_element_procs,
    int* eei_index,
    double* eei_data );
```

**C**
```c
FORTRAN(get_elementelement_interactions)(
    int* slave_element_block_ids,
    int* slave_element_indexes_in_block,
    int* master_element_block_ids,
    int* master_element_indexes_in_block,
    int* master_element_procs,
    int* eei_index,
    double* eei_data );
```

**Fortran**
```fortran
generatelements(elementelement_interactions(  
    slave_element_block_ids,  
    slave_element_indexes_in_block,  
    master_element_block_ids,  
    master_element_indexes_in_block,  
    master_element_procs,  
    eei_index,  
    eei_data)
```

where

slave_element_block_ids is an array (of length number_of_interactions) that contains the Element_Block ID for the slave element in each interaction.
slave_element_indexes_in_block is an array (of length number_of_interactions) that contains the index in the Element_Block (using Fortran indexing conventions) for the slave element in each interaction.
master_element_block_ids is an array (of length number_of_interactions) that contains the Element_Block ID for the master element in each interaction.
master_element_indexes_in_block is an array (of length number_of_interactions) that contains the index in the Element_Block (using Fortran indexing conventions) for the master element in each interaction.

master_element_procs is an array (of length number_of_interactions) that contains the processor that owns the master_element in each interaction.

eei_index is an array (of length number_of_interactions) that contains the offset into the eei_data array for the data for each interaction (i.e., the data for interaction j begins at eei_data[eei_index[j]]).

eei_data is an array (of length eei_data_size) that contains the data for each interaction (see Section 1.3.4). The data for each interaction is contiguous.

### 3.9.13 Deleting Interactions

The following function permits the host code to delete all previously found interactions before conducting a new search. This function is of particular use when a single search object conducts two different enforcements. For example, if an analysis uses both ContactGapRemoval and ContactTDEnforcement with a single ContactSearch object, then the interactions used for ContactGapRemoval can negatively affect the ContactTDEnforcement. In this case, it is better to delete the interactions determined for ContactGapRemoval before doing a search for ContactTDEnforcement.

```cpp
C++
ContactErrorCode
ContactSearch::Delete_All_Interactions();
```

```c
C
FORTRAN(delete_all_interactions)();
```

```fortran
Fortran
delete_all_interactions()
```
Search Functions
4. Gap Removal Enforcement Functions

The gap removal enforcement will compute a displacement increment needed to remove overlaps, as discussed in Section 1.5. This section describes functions that construct and operate on ContactGapRemoval “objects.” For the C++ API, these are true objects permitted by the object-oriented capabilities of the language. In the C and Fortran APIs, these functions create and operate on a ContactGapRemoval “object,” only one of which is currently allowed.

In this Section the ACME Gap Removal Enforcement C++, C and Fortran APIs are presented. The C++ API uses the full object-oriented capabilities of the language. The C and Fortran APIs are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The FORTRAN macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address.

The Enforcement_Interface.h header file, located in the ACME enforcement directory, includes the prototypes for the C and Fortran functions described in this chapter, and the ContactGapRemoval.h file includes the C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactEnforcement.h and ContactGapRemoval.h; these indicate the acceptable integer values that may be used in the C and Fortran APIs.

4.1 Constructing a ContactGapRemoval Object

There is one general purpose constructor for the ContactGapRemoval object. There are two restart constructors for this object. They are of the same form as all the other objects, as discussed in Sections 1.12, 2.3, and 2.4, so they will not be discussed further in this section.

The prototype for the initial ContactGapRemoval constructor is:

\[
\begin{align*}
\text{C++} & \quad \text{ContactGapRemoval::ContactGapRemoval(} \\
& \quad \quad \text{double* Enforcement_Data,} \\
& \quad \quad \text{ContactSearch* search,} \\
& \quad \quad \text{ContactSearch::ContactErrorCode& error }); \\
\text{C} & \quad \text{FORTRAN(build_gap_removal)(} \\
& \quad \quad \text{double* Enforcement_Data,} \\
& \quad \quad \text{int* error }); \\
\text{Fortran} & \quad \text{build_gap_removal(} \\
& \quad \quad \text{Enforcement_Data,} \\
& \quad \quad \text{error }); \\
\end{align*}
\]

where

\[
\text{Enforcement Data is a real array (of length 1*(number of entity keys)*(number of entity keys)) that stores the kinematic partition factor. It is structured [f_key*number_entity_keys+n_key] where f_key is the face key and n_key is the node}
\]
Gap Removal Enforcement Functions

The kinematic partition factor controls the master/slave relationship between two entities as described in Section 1.5.

search is the ContactSearch object from which the topology, interactions, and configurations are obtained.

error is the error code (described in Section 1.7) that will reflect any errors that were detected.

4.2 Computing the Gap Removal Displacements

This member function computes the displacement increments necessary to remove the initial gaps that are contained in the ContactSearch object topology. A static 1-configuration search should be used to define the interactions prior to calling this member function (regardless of the type of mechanics being solved). An iterative algorithm is used to remove the gaps. For all continuum element meshes, the iterative algorithm converges in one iteration. For meshes that contain shell elements, the user must set up the problem correctly or no solution exists and the gaps will not be removed. If the user creates a mesh where a shell element is sandwiched between two continuum elements, an overlap exists and the continuum elements are the pure masters, there is no degrees of freedom to move the shell element to remove the gaps (because the continuum faces are fixed due to the partitioning). If this example problem was set up as a symmetric removal, a solution does exist but the algorithm will have to iterate to get it. The user must specify the maximum number of iterations to use (they are computationally inexpensive and only done once so it a large number of iterations is practical) and a gap that is considered trivial. The algorithm iterates until either one of the criteria is met and then exists.

C++
ContactErrorCode ContactGapRemoval::Compute_Gap_Removal(
    int max_iterations,
    double trivial_gap,
    double* displ_cor);

C
FORTRAN(compute_gap_removal)(
    int* max_iterations,
    double* trivial_gap,
    double* displ_cor,
    int* error);

Fortran
compute_gap_removal(
    max_iterations,
    trivial_gap,
    displ_cor,
    error)

where

max_iterations is the maximum number of iterations to allow the iterative solver to use.
trivial_gap is the maximum gap the user wants after the gap_removal.
displ_cor is the displacement correction needed at each node to remove the initial gaps.

4.3 Destroying a ContactGapRemoval Object

C++ ~ContactGapRemoval();
C

FORTRAN(clean-up_gap_removal)();

Fortran  cleanup_gap_removal()
Gap Removal Enforcement Functions
5. Explicit Transient Dynamic Enforcement Functions

This section describes functions that construct and operate on ContactTDEnforcement “objects” that are used to enforce contact conditions between various combinations of nodes/faces of continuum elements and shells. For the C++ API, these are true objects permitted by the object-oriented capabilities of the language. In the C and Fortran APIs, these functions create and operate on a ContactTDEnforcement “object,” only one of which is currently allowed.

In this Section the ACME Explicit Transient Dynamic Enforcement C++, C and Fortran APIs are presented. The C++ API uses the full object-oriented capabilities of the language. The C and Fortran APIs are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The FORTRAN macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address.

The Enforcement_Interface.h header file, located in the ACME enforcement directory, includes the prototypes for the C and Fortran functions described in this chapter, and the ContactTDEnforcement.h file includes the C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactEnforcement.h and ContactTDEnforcement.h; these indicate the acceptable integral values that may be used in the C and Fortran APIs.

5.1 Creating a ContactTDEnforcement Object

There is one general purpose constructor for the ContactTDEnforcement object. There are two restart constructors for this object. They are of the same form as all the other objects, as discussed in Sections 1.12, 2.3, and 2.4, so they will not be discussed further in this section.

The prototype for the initial ContactTDEnforcement constructor is:

```cpp
ContactTDEnforcement::ContactTDEnforcement(
    const double* Enforcement_Data,
    ContactSearch* search,
    ContactSearch::ContactErrorCode& error);
```

```c
FORTRAN(build_td_enforcement)(
    double* Enforcement_Data,
    int* error);
```

```fortran
build_td_enforcement(
    Enforcement_Data,
    error)
```

where

Enforcement_Data is a real array (of length 2*(number of entity keys)*(number of entity keys)) that stores the kinematic partition factor and the friction model id. It is structured
Explicit Transient Dynamic Enforcement Functions

\[(f_{\text{key}} \times \text{number entity keys} + n_{\text{key}}) \times \text{size data per pair} + \text{variable index}\] where \(f_{\text{key}}\) is the face key and \(n_{\text{key}}\) is the node key. The kinematic partition factor controls the master/slave relationship between two entities. A factor between 0 and 1 specifies a fixed kinematic partitioning. A value of 2 indicates the code should compute this for each interaction pair based on the physical data. The friction model id pertains to the constitutive behavior of the interactions and is described in the following section.

search is the ContactSearch object from which the topology, interactions, and configurations are obtained.

error is the error code (described in Section 1.7) that will reflect any errors that were detected.

### 5.2 Defining Enforcement Models

The contact behavior is controlled by enforcement models. Thirteen types of enforcement models are currently supported; TD\_FRICTIONLESS, TD\_CONSTANT\_FRICTION, TD\_TIED, TD\_SPOT\_WELD, TD\_PRESSURE\_DEPENDENT, TD\_VELOCITY\_DEPENDENT, TD\_SPRING\_WELD, TD\_ADHESION, TD\_COHESIVE\_ZONE, TD\_JUNCTION, TD\_THREADED, TD\_PV\_DEPENDENT, TD\_AREA\_WELD. Note that the TD\_POINT\_WELD has been renamed TD\_SPRING\_WELD for clarity and that these models work in essentially the same fashion for shells as they do for faces of continuum elements. For the C++ API, please notice that Add\_Enforcement\_Model is a function inherited by ContactTDEnforcement. Finally, all models depend on local data (i.e., failure models are not global). To define the enforcement model to be used, the following function must be called:

```cpp
ContactSearch::ContactErrorCode
ContactEnforcement::Add_Enforcement_Model(
    Enforcement_Model_Types type,
    int* ID,
    int* integer_data,
    double* real_data);
```

```fortran
FORTRAN(td_add_enf_model)(
    int* type,
    int* id,
    int* integer_data,
    double* real_data,
    int* error);
```

```fortran
td_add_enf_model ( 
    type,
    id,
    integer_data,
    real_data,
    error)
```

where

type is the enforcement model type (as shown in Table 15).
id is a positive integer identifier for the model.
integer_data is an array of integer data that is particular to the model (as shown in Table 15).
real_data is an array of real data that is particular to the model (as shown in Table 15).
error is the return error code for the C and Fortran APIs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer Data</th>
<th>Real Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_FRICTIONLESS = 1</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>TD_CONSTANT_FRICTION=2</td>
<td>None</td>
<td>Friction Coefficient</td>
</tr>
<tr>
<td>TD_TIED=3</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>TD_SPOT_WELD=4</td>
<td>Failure_Steps</td>
<td>Normal Capacity, Tangential Capacity</td>
</tr>
<tr>
<td></td>
<td>Failed_Model_ID</td>
<td></td>
</tr>
<tr>
<td>TD_PRESSURE_DEPENDENT=5</td>
<td>None</td>
<td>Friction Coefficient, Reference Pressure, Offset Pressure, Pressure Exponent</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TD_VELOCITY_DEPENDENT=6</td>
<td>None</td>
<td>Static Coefficient, Dynamic Coefficient, Velocity Decay</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>TD_SPRING_WELD=7</td>
<td>Normal_Traction_Function_ID, Tangential_Function_Table_ID, Failure_Steps, Failed_Model_ID</td>
<td>Failure_Criterion_Exponent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD_ADHESION=8</td>
<td>Adhesion_Function_ID</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD_COHESIVE_ZONE=9</td>
<td>Traction_Displacement_Function_ID</td>
<td>Critical_Normal_Gap, Critical_Tangential_Gap</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD_JUNCTION=10</td>
<td>Normal_Traction_Table_ID, Tangential_Traction_Table_ID</td>
<td>Tangential_Traction_Gap_Cutoff</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TD_THREADED=11</td>
<td>Normal_Traction_Table_ID, Tangential_Traction_Table_ID, Tangential_Traction_Gap_Table_ID</td>
<td>Normal_Capacity, Tangential_Capacity, Failure_Criteria_Exponent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Explicit Transient Dynamic Enforcement Functions

Table 15 Transient Dynamic Enforcement Models and Data

<table>
<thead>
<tr>
<th>Type</th>
<th>Integer Data</th>
<th>Real Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_PV_DEPENDENT=12</td>
<td>None</td>
<td>Static_Coefficient, Dynamic_Coefficient, Velocity_Decay, Reference_Pressure, Offset_Pressure, Pressure_Exponent</td>
</tr>
<tr>
<td>TD_AREA_WELD=13</td>
<td>Failure_Steps, Failed_Model_ID</td>
<td>Normal Capacity, Tangential Capacity</td>
</tr>
<tr>
<td>TD_USER=14</td>
<td>Num_Integer_Vals, Num_Real_Vals, Num_Hist_Vals, {integer_array}</td>
<td>{real_array}</td>
</tr>
</tbody>
</table>

Notes: The following models are not currently supported for ContactNodeSurfaceInteractions: TD_TIED, TD_SPOT_WELD, TD_SPRING_WELD, TD_ADHESION, TD_COHESIVE_ZONE, TD_JUNCTION, TD_THREADED, TD_AREA_DEPENDENT.

For the TD_FRICTIONLESS model, only compressive normal tractions are allowed to enforce impenetrability.

The TD_CONSTANT_FRICTION model is a Coulomb model with the tangential traction limit given by $\mu p$, where $\mu$ is the Friction Coefficient and $p$ is the normal pressure.

No relative motion is allowed for the TD_TIED model.

For the TD_SPOT_WELD model, the behavior is the same as the TD_TIED model until a failure limit based on Normal and Tangential Capacity is reached. At this point, the model takes the requested number of Failure Steps to transition to the Failed Model prescribed by the Failed_Model_ID.

The TD_PRESSURE_DEPENDENT model is similar to the TD_CONSTANT_FRICTION model, except the tangential limit is given by $\mu \left( \frac{p + p_o}{p_r} \right)^k$, where $p_o$ is the Offset Pressure, $p_r$ is the Reference Pressure and $k$ is the Pressure Exponent.

The TD_VELOCITY_DEPENDENT model is also similar to the TD_CONSTANT_FRICTION model, except here the tangential limit is given by $(\mu_s - \mu_d) \exp(-d \|v\|) + \mu_d \|v\|$, where $\mu_s$ is the Static Coefficient, $\mu_d$ is the Dynamic Coefficient, $d$ is the Velocity Decay and $\|v\|$ is the magnitude of the (tangential) relative velocity.
The TD_SPRING_WELD model is similar to the TD_SPOT_WELD model, except that in the pre-failure regime, the interactions follow the force-displacement relations given by the two tables Normal_Traction_Function (only in tension) and Tangential_Traction_Function. The values of ordered pairs in these tables are expected to be non-negative and the last ordinates are used as critical force values for the failure criterion (i.e. they can not be zero).

The TD_ADHESION model is an enhanced version of TD_FRICTIONLESS, where, in tension, a restoring force is given by the tabular Adhesion_Function. The values of the Adhesion_Function are expected to be non-negative.

The TD_COHESIVE_ZONE model follows Tvergaard and Hutchinson's\(^1\) developments, where the normal and tangential cohesive tractions are coupled through a quadratic failure variable. This variable is the only argument to a single (tabular) force-displacement relation which is scaled by Critical_Normal_Gap and Critical_Tangential_Gap to obtain the tensile normal force and tangential force, respectively. In compression the normal force sufficient to prevent penetration is used. Also, since the failure variable is non-negative any part of the tabular force-displacement curve with negative abscissas is ignored.

TD_JUNCTION is a friction model with adhesion. Tensile normal forces are given by a tabular function of gap, i.e. Normal_Traction_Table. The limit of the tangential traction forces is given by the Tangential_Traction_Table which is a function of relative velocity (slip) and is subject to a cutoff when the gap exceeds the Tangential_Traction_Gap_Cutoff.

The TD_THREADED is a weld-type model. Tensile normal forces are given by a tabular function of the normal distance, Normal_Traction_Table, and the tangential forces are given by the multiplication of two tabular functions: Tangential_Traction_Table and Tangential_Traction_Gap_Table, which are functions of the tangential and normal relative distances respectively.

TD_PV_DEPENDENT is a multiplicative combination of the TD_PRESURE_DEPENDENT and TD_VELOCITY_DEPENDENT friction models.

TD_AREA_WELD is essentially the same as the TD_SPOT_WELD except the constitutive limits are scaled by nodal areas before being compared with the nodal forces.

Please note the following models are traction based (in the sense that the constitutive values are scaled by a nodal area before being applied as a nodal force): TD_AREA_WELD, TD_COHESIVE_ZONE, TD_ADHESION, TD_THREADED, TD_CONSTANT_FRICTION, TD_PRESURE_DEPENDENT, TD_VELOCITY_DEPENDENT, TD_PV_DEPENDENT. Also, for all the models with adhesive/cohesive behavior (i.e., TD_ADHESION, TD_COHESIVE_ZONE,

---

Explicit Transient Dynamic Enforcement Functions

TD_SPRING_WELD, TD_JUNCTION), the Search_Normal_Tolerance should be set to be exactly equal to the Critical_Normal_Gap (in the case of TD_ADHESION, this is the last abscissa of the Adhesion Function table).

Through an FORTRAN based interface the ACME user has the ability to add a ContactTDEnfModel at run-time, given support from the host code. The user must supply a number of subroutines that are called from a generic TD enforcement model (TD_USER) via callbacks. The parameters required for initialization are Num_Integer_Vals: the number of integer parameters, Num_Real_Vals: the number of real parameters, Num_Hist_Vals: the number of history variables being requested and the integer and real arrays of parameters whose lengths correspond to Num_Integer_Vals and Num_Real_Vals, respectively. The subroutines that the user can create to control the behavior of the interaction follow the templates below. Any user-defined subroutine will overwrite the stubs given below.

```fortran
subroutine user_initialize_model(enf, id, rdata, idata, istat)
  integer idata(*), id, istat
  real*8 rdata(*)
  istat = 0
  return
end

subroutine user_initialize_time_step(enf, id, rdata, idata, istat)
  integer idata(*), id, istat
  real*8 rdata(*)
  istat = 0
  return
end

subroutine user_init_node_state_data(enf, id, rdata, idata, sdata, istat)
  integer idata(*), id, istat
  real*8 rdata(*), sdata(*)
  istat = 0
  return
end

subroutine user_interaction_type(enf, id, rdata, idata, sdata)
  integer idata(*), id, istat
  real*8 rdata(*), sdata(*)
  istat = 0
  return
end
```

The types are enumerated in ContactTDEnfModel.h.
Explicit Transient Dynamic Enforcement Functions

c and are repeated here:
c TDEM_FRICTIONLESS=0, // return value 0
c TDEM_FRICTIONAL, // 1
c TDEM_TIED, // 2
c TDEM_ADHESIVE, // 3
c TDEM_SPRINGY, // 4
c TDEM_ADHESIVE_FRICTION, // 5
c
The control what constraints the enforcement routine tries to impose
c and what predictors will be used to generate a trial force for the
c user-defined material model. A FRICTIONLESS interaction will only
c have impenetrability enforced. A FRICTIONAL interaction will have
c impenetrability enforced and, in addition, a trial force due to
c trying to prevent tangential motion i.e. stick. A TIED interaction
c will enforce that the node and surface are coincident at the same
c (convected) point for all time. An ADHESIVE interaction is like a
c FRICTIONLESS interaction with the addition of a trial force for
c separation that attempts to bring the node and surface together.
c A SPRINGY interaction is similar to a TIED interaction in that the
c node and surface are always bound together at the same "contact
c point" but in this case with a penalty-like trial force instead
c of a Lagrange multiplier. An ADHESIVE_FRICTION interaction is
c the combination of the FRICTIONAL and the ADHESIVE types so that
c the trial force is composed of a normal force (in the case of
c separation) and a tangential force that attempts to stick the
c node and surface together.

---------------------------------------------------------------------
subroutine user_interaction_type(enf, ni, id, rdata, itype, *
   idata, index, istat)
   integer istat, id, itype, index, idata(*)
   real*8  rdata(*)
   istat = 0
   return
end

---------------------------------------------------------------------

user_is_active simply takes the current value of the gap ("gap" in
c the argument list and the material parameters in rdata and idata to
c determine whether this interaction is "active" i.e. will generate a
c force based on proximity of the two surfaces.

---------------------------------------------------------------------
subroutine user_is_active(enf, ni, id, rdata, idata, *
   itype, index, gap, istat)
   integer istat, id, itype, index, idata(*)
   real*8  gap, rdata(*)
   istat = 0
   return
end

---------------------------------------------------------------------

user_limit_force takes the value of the trial force passed in by
c "force" and returns the constitutively determined value using the
c the arguments:
c gap : the normal closest-point distance between the node and surface,
c rel_disp : the absolute vectorial distance between node & contact point
Explicit Transient Dynamic Enforcement Functions

- **c slip**: the incremental vectorial distance between node & contact point
- **c normal**: outward unit normal to the surface
- **c dt**: timestep
- **c area**: tributary area of the node
- **c The last argument is 0 if the force was left as is, and nonzero if**
  - it was modified.

```fortran
subroutine user_limit_force(enf, ni, id, rdata, idata, itype, index, gap, rel_disp, slip, normal, dt, area, force, istat)
   integer istat, id, itype, index, idata(*)
   real*8 gap, dt, area
   real*8 rel_disp(*), slip(*), normal(*), force(*), rdata(*)
   force(1) = 0.0
   force(2) = 0.0
   force(3) = 0.0
   istat = 1
   return
end
```

In addition, a number of query functions are provided:

```fortran
c---
c query a table for it’s last abscissa value

c Input:

c   enf: enforcement object, passed thru to the userquery_xxx() functions

c   id_table: table id

c Output:

c   value: last abscissa value

c---
call userquery_table_last_abscissa(enf, id_table, value)

c---
c query a table for an interpolated value

c Input:

c   enf: enforcement object, passed thru to the userquery_xxx() functions

c   id_table: table id

c   abscissa: abscissa value

c Output:

c   value: interpolated ordinate value

c---
```
call userquery_table_interpolate(enf, id_table, abscissa, ordinate)

c query the enforcement object for the number of nodes

c Input:

c  enf: enforcement object, passed thru to the userquery_xxx() functions

c Output:

c  nnodes: the number of nodes

call userquery_number_of_nodes(enf, nnodes)


c query a node for it’s state data

c Input:

c  enf: enforcement object, passed thru to the userquery_xxx() functions

c  model_id: enforcement model id

c  node: node index (1, nnodes)

c  ioffset: state data offset (1, num_state_variables)

c Output:

c  value: value of the stat data

call userquery_node_state_data(enf, model_id, node, ioffset, value)

and user functions to set parameters

c set a node’s state data

c Input:

c  enf: enforcement object, passed thru to the userquery_xxx() functions

c  model_id: enforcement model id

c  node: node index (1, nnodes)

c  ioffset: state data offset (1, num_state_variables)

c  value: value of the stat data

call userset_node_state_data(enf, id_model, node, istate_data_offset, value)
Explicit Transient Dynamic Enforcement Functions

```fortran
!
!
c----------------------------------------------------------------------
c set a node’s state data

c Input:
c
  enf: enforcement object, passed thru to the userquery_xxx() functions
  model_id: enforcement model id
  nfi: node/face interaction object

c----------------------------------------------------------------------
call userset_nfi_failure_tied(enf, model_id, nfi)
!
An example of a weld model is described below:

subroutine user_sprngweld_init_model(enf, id, rdata, *
                                       idata, istat)
    integer istat, id, idata(*)
    real*8  rdata(*), value
    call userquery_table_last_abscissa(enf, idata(2), value)
      rdata(2) = value
    call userquery_table_last_abscissa(enf, idata(3), value)
      rdata(3) = value
    istat = 1
    return
end

subroutine user_sprngweld_init_tstep(enf, id, rdata, *
                                       idata, istat)
    integer idata(*), id, istat, index
    real*8  rdata(*), value
    call userquery_number_of_nodes(enf, nnodes)
      value = 0.0
    index = 4
    do i=1,nnodes
      call userset_node_state_data(enf, id, i, index, value)
    enddo
    istat = 1
    return
end

subroutine user_sprngweld_init_sdata(enf, id, rdata, idata, *
                                       sdata, istat)
    integer idata(*), id, istat
    real*8  rdata(*), sdata(*)
    sdata(1) = idata(1)
    sdata(2) = 0.0
    sdata(3) = 0.0
    sdata(4) = 0.0
    istat = 1
    return
end

subroutine user_sprngweld_itype(enf, zni, id, rdata,
```
Explicit Transient Dynamic Enforcement Functions

*                           idata, itype, index, istat)
integer istat, id, itype, index, idata(*), indx
real*8  rdata(*), value
indx = 1
call userquery_node_state_data(enf, id, index, indx, value)
if (value.gt.0.0) then
   istat = 4
else
   istat = -1
endif
return
end

subroutine user_sprngweld_is_active(enf, zni, id, rdata, idata, *                                   itype, index, gap, istat)
integer istat, id, itype, index, idata(*), indx
real*8  gap, rdata(*), value
indx = 1
call userquery_node_state_data(enf, id, index, indx, value)
if (value.gt.0.0) then
   istat = 1
else
   istat = -1
endif
return
end

subroutine user_sprngweld_limit_force(enf, zni, id, rdata, idata, *                                      itype, node_index, *                                      gap, rel_disp, *                                      slip, normal, dt, area, *                                      force, istat)
integer istat, id, itype, node_index, nfi_index, index, idata(*)
real*8 gap, dt, area, f_n, value, s_i
real*8 strength, already_decremented, s_n, s_t(3)
real*8 mag_f_t, mag_s_t, norm_ratio, tang_ratio
real*8 failure_crit, scale, f_n_fail, f_t_fail
real*8 rel_disp(*), slip(*), normal(*), force(*), rdata(*)

s_i = idata(1)
if (itype.eq.0) then
   index = 1
   call userquery_node_state_data(enf, id, node_index, *                                 index, strength)
   if (strength.gt.0.0) then
      s_n = 0.0
      do i=1,3
         s_n = s_n+rel_disp(i)*normal(i)
      enddo
      do i=1,3
         s_t(i) = rel_disp(i) - s_n*normal(i)
      enddo
      mag_s_t = 0.0
      do i=1,3
Explicit Transient Dynamic Enforcement Functions

```fortran
mag_s_t = mag_s_t + s_t(i) * s_t(i)
enddo
mag_s_t = sqrt(mag_s_t)
index = 4
call userquery_node_state_data(enf, id, node_index, index, 
  already_decremented)
if (strength.eq.s_i) then
  f_n = 0.0
  do i=1,3
    f_n = f_n + force(i) * normal(i)
  enddo
  if (s_n.gt.0.0) then
    call userquery_table_interpolate(enf, idata(2), 
      s_n, value)
    f_n = -value
  endif
  norm_ratio = abs(max(s_n, 0.0d0)/rdata(2))
  call userquery_table_interpolate(enf, idata(3), 
    mag_s_t, value)
  mag_f_t = -value
  tang_ratio = abs(mag_s_t/rdata(3))
  failure_crit = norm_ratio**rdata(1) + 
    tang_ratio**rdata(1)
  if (failure_crit.ge.1.0) then
    if (already_decremented.eq.0.0) then
      strength = strength - 1.0
      already_decremented = 1.0
      index = 1
      call userset_node_state_data(enf, id, node_index, 
        index, strength)
    endif
    index = 4
    call userset_node_state_data(enf, id, node_index, 
      index, already_decremented)
  endif
  index = 2
  call userset_node_state_data(enf, id, node_index, 
    index, f_n)
  index = 3
  call userset_node_state_data(enf, id, node_index, 
    index, mag_f_t)
endif
if (mag_s_t.gt.1.0e-10) then
  do i=1,3
    force(i) = f_n*normal(i) + (mag_f_t/mag_s_t)*s_t(i)
  enddo
else
  do i=1,3
    force(i) = f_n*normal(i)
  enddo
endif
else
  if (already_decremented.eq.0.0) then
    strength = strength - 1.0
    already_decremented = 1.0
    index = 1
  endif
```

116
call userset_node_state_data(enf, id, node_index, 
       index, strength)
*  
    index = 4
call userset_node_state_data(enf, id, node_index, 
       index, already_decremented)
*  
endif
scale = (strength+1)/s_i
index = 2
call userquery_node_state_data(enf, id, node_index, 
       index, f_n_fail)
if (s_n.gt.0.0) then
  f_n = scale*f_n_fail
else
  f_n = 0.0
  do i=1,3
     f_n = f_n+force(i)*normal(i)
  enddo
endif
if (mag_s_t.gt.1.0e-10) then
  index = 3
  call userquery_node_state_data(enf, id, node_index, 
         index, f_t_fail)
  do i=1,3
     force(i) = f_n*normal(i) + 
              (scale*f_t_fail/mag_s_t)*s_t(i)
  enddo
else
  do i=1,3
     force(i) = f_n*normal(i)
  enddo
endif
if (strength.le.0.0) then
  call userset_nfi_failure_tied(enf, id, zni)
endif
endif
else
  force(1) = 0.0
  force(2) = 0.0
  force(3) = 0.0
endif
istat = 1
return
end

5.3 Controlling the Algorithm

The enforcement algorithms are iterative in nature (i.e., a predictor-corrector algorithm is used). The accuracy of the algorithm can be improved by doing additional iterations. Initial testing indicates that 5 iterations dramatically improve the accuracy of the solution (especially for frictional problems). If there is a mesh mismatch and the automatic kinematic partitioning is used, more iterations may be necessary. The following member func-
Explicit Transient Dynamic Enforcement Functions

tion allows the user to set the number of iterations to use (the default is 1 if it is not explicitly set).

\[
\text{C++} \quad \text{ContactSearch::ContactErrorCode}
\text{ContactTDEnforcement::Set_Number_of_Iterations(
    \text{int number_iterations});}
\]

\[
\text{C} \quad \text{FORTRAN(set_td_iterations)(
    \text{int* number_iterations,}
    \text{int* error});}
\]

\[
\text{Fortran} \quad \text{set_td_iterations(}
    \text{number_iterations,}
    \text{error});}
\]

where

number_iterations is the number of iterations to use in the enforcement.
error is the return error code for the C and Fortran APIs.

The convergence tolerance for the iteration loop can also be specified. The L2 norm of the contact force being added at the current iteration, i.e. the increment in the contact force, is compared to this tolerance to determine whether the iteration loop should exit early. The tolerance defaults to 1.0E-10 if not explicitly set via

\[
\text{C++} \quad \text{ContactSearch::ContactErrorCode}
\text{ContactTDEnforcement::Set_Convergence_Tolerance(
    \text{Real tol});}
\]

\[
\text{C} \quad \text{FORTRAN(set_td_convg_tol)(
    \text{double* tol,}
    \text{int* error});}
\]

\[
\text{Fortran} \quad \text{set_td_convg_tol(}
    \text{tol,}
    \text{error});}
\]

where

tol is the convergence tolerance to use in the enforcement.
error is the return error code for the C and Fortran APIs.

5.4 Specifying Symmetric Nodes

This enforcement was originally written for Lagrangian based contact. Its use has expanded to a number of codes and algorithms including ALEGRA/SHISM which is a coupled Lagrangian/ALE technique for modeling penetration events (e.g., earth penetrators, armor, etc.). Due to the formulation of this problem, the tip of the penetrator must be treated in a different manner than a typical contact. Specifically, the tip node of the penetrator and the
corresponding node of the ALE mesh must have consistent constraints. The search does not give the symmetric interactions in general. The ability to enforce this symmetry is provided with the following function. This should ONLY be used with the ALEGRA/SHISM capability.

\[ \text{C++} \]
ContactSearch::ContactErrorCode
ContactTDEnforcement::Enforce_Symmetry_on_Nodes(
  int exodus_id_1,
  int exodus_id_2 );

\[ \text{C} \]
FORTRAN(set_td_enf_symm_nodes)(
  int* exodus_id_1,
  int* exodus_id_2,
  int* error );

\[ \text{FORTRAN} \]
set_td_enf_symm_nodes(
  exodus_id_1,
  exodus_id_2,
  error );

where

\[ \text{exodus_id}_1 \text{ is the exodus id of the tip node of the penetrator.} \]
\[ \text{exodus_id}_2 \text{ is the exodus id of the corresponding node on the ALE target mesh.} \]
\[ \text{error is the return error code for the C and Fortran APIs.} \]

### 5.5 Computing the Contact Forces

The following member function computes the contact forces necessary to enforce the contact constraints that are contained in the ContactSearch object.

\[ \text{C++} \]
ContactSearch::ContactErrorCode
ContactTDEnforcement::Compute_Contact_Force(
  double dt_old,
  double dt,
  double* mass,
  double* density,
  double* wavespeed,
  double* force); 

\[ \text{C} \]
FORTRAN(compute_td_contact_force)(
  double* dt_old,
  double* dt,
  double* mass,
  double* density,
  double* wavespeed
  double* force,
  int* error);

\[ \text{Fortran} \]
compute_td_contact_force(
  dt_old,
Explicit Transient Dynamic Enforcement Functions

\[
dt, 
mass, 
density, 
wavespeed, 
force, 
error
\]

where

\( dt_{\text{old}} \) is the previous time step for a central difference integrator. 
\( dt \) is the current time step for a central difference integrator. 
mass is an array that contains the nodal mass for each node. 
density is an array that contains the nodal “density”. Computing this as the nodal mass divided by the sum of the contributing element volumes works well. This array is used in computing the automatic kinematic partitioning. 
wavespeed is an array that contains the nodal “wavespeed”. Computing this as the average of the contributing element wavespeeds works well. This array is used in computing the automatic kinematic partitioning. 
force is the return array containing the computed contact force vectors for each node. 
error is the return error code for the C and Fortran APIs.

5.6 Extracting Plot Variables

The ContactTDEnforcement includes support for extracting variables that can be useful for plotting. The nodal variables that can be extracted are listed in Table 16 and the global ones in Table 17.

<table>
<thead>
<tr>
<th>Plot Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFACE = 1</td>
<td>The value of CONFACE denotes status of interactions at a node. A value of 0.5 indicates the node is not in contact. A value of 1, 2, or 3 denotes the number of interactions at that node.</td>
</tr>
<tr>
<td>NORMAL_FORCE_MAG = 2</td>
<td>This variable holds the normal force magnitude for the node. If multiple constraints exist at a node, this value is for the last constraint.</td>
</tr>
<tr>
<td>TANGENTIAL_FORCE_MAG = 3</td>
<td>This variable holds the tangential force magnitude for the node. If multiple constraints exist at this node, the value is for the last constraint.</td>
</tr>
</tbody>
</table>
The interface for getting these variables is:

**C++**

```cpp
ContactSearch::ContactErrorCode
ContactTDEnforcement::Get_Plot_Variable(
    Contact_TDEnf_Plot_Vars plot_var,
    Real* data);
```

**C**

```c
FORTRAN(get_td_plot_variable)(
    int& plot_var,
    Real* data,
    int& error);
```

**Fortran**

```fortran
get_td_plot_variable(
    plot_var,
    data,
    error)
```

where

- `plot_var` is the value for the variable given in Table 16.
- `data` is an array (number of nodes long) in which the values will be loaded.
- `error` is the return error code.

### Table 16 Nodal TD Plot Variables

<table>
<thead>
<tr>
<th>Plot Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDIRNOR[XYZ] = [4,5,6]</td>
<td>The vector components of the normal direction for the constraint at a node are held in CDIRNOR. If multiple constraints exist at a node, these values are for the last constraint.</td>
</tr>
<tr>
<td>CDIRTAN[XYZ] = [7,8,9]</td>
<td>The vector components of the tangential direction for the constraint at a node are held in CDIRTAN. If multiple constraints exist at a node, these values are for the last constraint.</td>
</tr>
<tr>
<td>SLIPMAG = 10</td>
<td>SLIPMAG is the incremental slip for this time step at a node. If multiple constraints exist at a node, this value is for the last constraint.</td>
</tr>
</tbody>
</table>

### Table 17 Global TD variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORCE_X=1</td>
<td>the sum of the contact forces in the x direction</td>
</tr>
</tbody>
</table>
Explicit Transient Dynamic Enforcement Functions

Table 17 Global TD variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORCE_Y=2</td>
<td>the sum of the contact forces in the y direction</td>
</tr>
<tr>
<td>FORCE_Z=3</td>
<td>the sum of the contact forces in the z direction</td>
</tr>
<tr>
<td>FORCE_NORM=4</td>
<td>the l2 norm of the sum of contact forces</td>
</tr>
<tr>
<td>DISSIPATION=5</td>
<td>the inner product of the contact force vector and the velocity</td>
</tr>
<tr>
<td>CONSTRAINT_NORM=6</td>
<td>the max norm of the forces enforcing constraints, i.e. those acting as Lagrange multipliers</td>
</tr>
<tr>
<td>INC_FORCE_NORM=7</td>
<td>the l2 norm of the force being added at the current iteration</td>
</tr>
</tbody>
</table>

All these variables have units of force, except DISSIPATION which has units of power. The interface for getting these variables is:

**C++**

```cpp
ContactSearch::ContactErrorCode
ContactTDEnforcement::Get_Global_Variable(
    Contact_TDEnf_Global_Vars var,
    Real& value);
```

**C**

```c
FORTRAN(get_td_global_variable)(
    int& var,
    Real value,
    int& error);
```

**Fortran**

```fortran
get_td_global_variable(
    var,
    value,
    error)
```

where

plot_var is the value for the variable given in Table 17.

### 5.7 Destroying a ContactTDEnforcement Object

**C++**

```cpp
~ContactTDEnforcement();
```

**C**

```c
FORTRAN(cleanup_td_enforcement)();
```

**Fortran**

```fortran
cleanup_td_enforcement()
```
6. Tied Kinematics Enforcement Functions

The ContactTiedKinematics object will compute positions for nodes to satisfy a no-relative-motion requirement (limited to a pure master-slave relationship) as discussed in Section 1.7. For the C++ API, these are true objects permitted by the object-oriented capabilities of the language. In the C and Fortran APIs, these functions create and operate on a ContactTiedKinematics “object,” only one of which is currently allowed.

In this Section the ACME Tied Kinematic Enforcement C++, C and Fortran APIs are presented. The C++ API uses the full object-oriented capabilities of the language. On the other hand, the C and Fortran APIs, which in actuality have been combined into a single interface, are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The FORTRAN macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address, not by value. For Fortran, there exists no capability to pass data by value, so simply specifying the name of the variable or array will allow it to be passed appropriately.

The Enforcement_Interface.h header file, located in the ACME enforcement directory, includes the prototypes for the C and Fortran functions described in this chapter, and the ContactTiedKinematics.h file includes the C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactEnforcement.h and ContactTiedKinematics.h; these indicate the acceptable integral values that may be used in the C and Fortran APIs.

Currently, a tied interaction between the edge of a shell and another face is enforced as a pinned connection, i.e. the relative motion of the nodes on the shell edge do not move, but the shell is free to rotate about the edge.

6.1 Constructing a ContactTiedKinematics Object

There is one general purpose constructor for the ContactTiedKinematics object. There are two restart constructors for this object. They are of the same form as all the other objects, as discussed in Sections 1.12, 2.3, and 2.4, so they will not be discussed further in this section.

The prototype for the initial ContactTiedKinematics constructor is:

\[
\text{C++} \quad \text{ContactTiedKinematics}::\text{ContactTiedKinematics}( \\
\quad \text{double* Enforcement Data,} \\
\quad \text{ContactSearch* search,} \\
\quad \text{ContactSearch::ContactErrorCode& error}); \\
\]

\[
\text{C} \quad \text{FORTRAN(build_tied_kinematics)(} \\
\quad \text{double* Enforcement Data,} \\
\quad \text{int* error}); \\
\]

\[
\text{Fortran} \quad \text{build_tied_kinematics}( \\
\quad \text{double* Enforcement Data,} \\
\quad \text{int* error}); \\
\]
Tied Kinematics Enforcement Functions

\[ \text{Enforcement\_Data, error);} \]

where

Enforcement\_Data is a real array (of length 1*(number of entity keys)*(number of entity keys)) that stores the kinematic partition factor. It is structured \([\text{n\_key*number\_entity\_keys+f\_key}]\) where n\_key is the node key and f\_key is the face key. The kinematic partition factor controls the master/slave relationship between two entities as described in Section 1.5. For this object the kinematic partition MUST specify a pure master/slave relationship.

search is the ContactSearch object from which the topology, interactions, and configurations are obtained.

error is the error code (described in Section 1.7) that will reflect any errors that were detected.

6.2 Computing the ContactTiedKinematic Displacements

A static 1-configuration search should be used to define the interactions prior to calling this member function. This function computes the final position of all nodes so that they kinematically satisfy the tied constraints.

C++
```
ContactErrorCode
ContactTiedKinematics::Compute_Position(
    double* position);
```

C
```
FORTRAN(compute_tied_position)(
    double* position,
    int* error);
```

Fortran
```
compute_tied_position(  
    position,  
    error)
```

where

position is an array of positions ordered (x,y,z) for node 1, (x,y,z) for node 2, etc. On input this is the current positions of all nodes. On output it is the positions of all the nodes that satisfy the no-relative-motion requirement between the slave nodes and the master surface face.

6.3 Destroying a ContactTiedKinematics Object

C++
```
~ContactTiedKinematics();
```

C
```
FORTRAN(cleanup_tied_kinematics)();
```

Fortran
```
cleanup_tied_kinematics()
```
7. Volume Transfer

This section describes functions that construct and operate on ContactVolumeTransfer “objects.” For the C++ API, these are true objects permitted by the object-oriented capabilities of the language. In the C and Fortran APIs, these functions create and operate on a ContactVolumeTransfer “object,” only one of which is currently allowed.

In this Section the ACME Volume Transfer C++, C and Fortran APIs are presented. The C++ API uses the full object-oriented capabilities of the language. The C and Fortran APIs are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The Fortran macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address and not value. For Fortran, it is not possible to pass by value, so simply specifying the name of the variable or array is sufficient.

The Enforcement_Interface.h header file, located in the ACME enforcement directory, includes the prototypes for the C and Fortran functions described in this chapter, and the ContactVolumeTransfer.h file includes the C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactEnforcement.h and ContactVolumeTransfer.h; these indicate the acceptable integer values for the C and Fortran APIs.

7.1 Constructing a Volume Transfer Object

There is one general purpose constructor for the ContactVolumeTransfer object. There are two restart constructors for this object. They are of the same form as all the other objects, as discussed in Sections 1.12, 2.3 and 2.4, so they will not be discussed further in this section.

The prototype for the initial ContactVolumeTransfer constructor is:

C++

ContactVolumeTransfer::ContactVolumeTransfer(
    const double* Enforcement_Data,
    ContactSearch* search,
    ContactSearch::ContactErrorCode& error );

C

FORTRAN(build_vol_tran)(
    double* Enforcement_Data,
    int* error );

Fortran

build_vol_tran(
    Enforcement_Data,
    error )

where,

Enforcement_Data is a real array (of length (number of entity keys)*(number of entity keys)) that stores the “near search tolerance”. It is structured [f_key*number_entity_keys+n_key] where n_key is the node key and f_key is the face key. The near search tolerance is used when determining if a slave node is in an element for nodal data interpolation/extrapola-
Volume Transfer

tion and must be set as greater than or equal to 0 (no checks are performed for negative values). Setting near search tolerance to 0 for an entity pair ensures that extrapolation outside an element is not performed. Setting it to a small value may be necessary in some cases where machine round-off causes node/element interactions not to be found. Note that the tolerance is an absolute number.

search is the ContactSearch object from which the topology, interactions, and configurations are obtained.

error is the error code (described in Section 1.7) that will reflect any errors that were detected.

7.2 Computing the Transfered Element and Nodal Data

The following member function performs the transfer of element and nodal data from the master to slave mesh. Nodal variable data is interpolated from the master (donor) to slave (receiver) mesh using linear interpolation. Element variable data is mapped between the two meshes using volume fraction weighting with the result being divided by the volume fraction of the receiving mesh filled by the donor mesh (volume_fraction below). Output includes both the transferred nodal and element variables and the volume fraction of the receiving mesh filled by the donor mesh.

C++
ContactSearch::ContactErrorCode
ContactVolumeTransfer::Compute_Volume_Transfer(
    int num_node_vars,
    int num_elem_vars,
    const double* donor_node_vars,
    const double* donor_elem_vars,
    double* receiver_node_vars,
    double* receiver_elem_vars,
    double* volume_fraction );

C
FORTRAN(compute_vol_tran)(
    int num_node_vars,
    int num_elem_vars,
    double* donor_node_vars,
    double* donor_elem_vars,
    double* receiver_node_vars
    double* receiver_elem_vars,
    double* volume_fraction,
    int* error);

Fortran
compute_vol_tran(
    num_node_vars,
    num_elem_vars,
    donor_node_vars,
    donor_elem_vars,
    receiver_node_vars,
    receiver_elem_vars,
    volume_fraction,
    error )

where

num_node_vars is the number of nodal variables to be transfered.
num_elem_vars is the number of element variables to be transferred.
donor_node_vars is an array that contains the nodal variables to be transferred at each node and so is
num_node_vars*number_of_nodes long. The array is structured such that variables cycle
faster than nodes.
donor_elem_vars is an array that contains the element variables to be transferred at each node and so
is num_elem_vars*number_ofelems long. The element variables cycle faster than do the
elements.
receiver_node_vars is an array which is initially empty and is filled with the transferred nodal data.
The array is num_node_vars*number_of_nodes long and nodal variables cycle faster than do the nodes.
receiver_elem_vars is an array which is initially empty and is filled with the transferred element da-
ta. The array is num_node_vars*number_of_elements long and nodal variables cycle fast-
er than do the nodes.
volume_fraction is an array number_of_elements long which, upon exit, contains the volume frac-
tion of the receiver elements filled by the donor elements.

7.3 Destroying a ContactVolumeTransfer Object

C++  ~ContactVolumeTransfer();

C    FORTRAN(cleanup_vol_trans)();

Fortran  cleanup_vol_trans()
Volume Transfer
8. MPC Enforcement

This section describes functions that construct and operate on ContactMPCs “objects.” For the C++ API, these are true objects permitted by the object-oriented capabilities of the language. In the C and Fortran APIs, these functions create and operate on a ContactMPCs “object,” only one of which is currently allowed.

In this section the ACME MPC Enforcement C++, C and Fortran APIs are presented. The C++ API uses the full object-oriented capabilities of the language. The C and Fortran APIs are a collection of functions that have a pure C interface and can be called from either C or Fortran routines. The FORTRAN macro that surrounds all calls in the C syntax converts the function by appending an underscore to the end of the function name, if appropriate. Because of this, all data in the C API must be passed by address.

The Enforcement_Interface.h header file, located in the ACME enforcement directory, includes the prototypes for the C and Fortran functions described in this chapter, and the ContactMPCs.h file includes the C++ prototypes. Enumerations for symbolic types used in the C++ API are also found in ContactEnforcement.h and ContactMPCs.h; these indicate the acceptable integer values for the C and Fortran APIs.

8.1 Constructing a ContactMPCs Object

There is one general purpose constructor for the ContactMPCs object. There are two restart constructors for this object. They are of the same form as all the other objects, as discussed in Sections 1.12, 2.3 and 2.4, so they will not be discussed further in this section.

The prototype for the initial ContactMPCs constructor is:

C++

```
ContactMPCs::ContactMPCs(
    const double* Enforcement_Data,
    ContactSearch* search,
    ContactSearch::ContactErrorCode& error );
```

C

```
FORTRAN(build_mpc_eqns)(
    double* Enforcement_Data,
    int* error );
```

Fortran

```
build_mpc_eqns(
    Enforcement_Data,
    error )
```

where,

```
Enforcement_Data is a real array. This array is currently unused and any data passed to the object is ignored.
search is the ContactSearch object from which the topology, interactions, and configurations are obtained.
error is the error code (described in Section 1.7) that will reflect any errors that were detected.
```
8.2 Computing the Multiple Point Constraint (MPC) Equations

The following member function calculates the MPC equations when called. No output (other than an error code) is returned.

```cpp
ContactSearch::ContactErrorCode
ContactMPCs::Compute_MPCs(
    Id_Numbering_Scheme id_num_scheme);
```

```c
FORTRAN(compute_mpc_eqns)(
    int& id_num_scheme,
    int* error);
```

```fortran
compute_mpc_eqns(
    id_num_scheme,
    error )
```

where

id_num_scheme is the numbering scheme to be used when returning face and node ids. Currently only HOST_GLOBAL_ID (= 1) numbering is available. When HOST_GLOBAL_ID is chosen, face and node IDs are numbered as the face_global_ids and node_global_ids supplied by the host code when the ContactSearch object was created (see Section 3.1).

error is the return error code for the C and Fortran APIs.

8.3 Getting the Number of MPC Equations

The following member function returns the number of MPC equations. This function must be called after the Compute function of Section 8.2. This function returns the number of MPC equations.

```cpp
ContactSearch::ContactErrorCode
ContactMPCs::Number_of_MPC_Equations(
    int& number_of_equations);
```

```c
FORTRAN(get_num_mpceqns)(
    int& number_of_equations,
    int* error);
```

```fortran
get_num_mpceqns(
    number_of_equations,
    error )
```

where

number_of_equations is the number of MPC equations.
error is the return error code for the C and Fortran APIs.
8.4 Getting the MPC Equations

The following member function returns the Multiple Point Constraint equations in terms of the involved slave-nodes, master-faces, master-face nodes and associated constraint coefficients. This function must be called after getting the number of constraint equations through a call to the function of Section 8.3.

```cpp
ContactMPCs::Get_MPC_Equations(
    int number_of_equations,
    int* snode_pid,
    int* snode_lid,
    int* mface_pid,
    int* mface_lid,
    int* nface_nodes,
    int* fnode_pid,
    int* fnode_lid,
    double* fnode_coefs);
```

```c
FORTRAN(get_num_mpceqns)(
    int& number_of_equations,
    int* error);
```

```fortran
get_num_mpceqns(
    number_of_equations,
    error )
```

where

- `number_of_equations` is the number of MPC equations.
- `snode_pid` is an array (number_of_equations long) containing the most significant word of the slave node ID for each MPC equation. Either processor ID (ACME_LOCAL_ID) or the most significant word in the host code global node ID (HOST_GLOBAL_ID) is returned.
- `snode_lid` is an array (number_of_equations long) containing the least significant word of the slave node ID for each MPC equation. Either local ID (ACME_LOCAL_ID) or the least significant word in the host code global node ID (HOST_GLOBAL_ID) is returned.
- `mface_pid` is an array (number_of_equations long) containing the most significant word of the master face ID for each MPC equation. Either processor ID (ACME_LOCAL_ID) or the most significant word in the host code global face ID (HOST_GLOBAL_ID) is returned.
- `mface_lid` is an array (number_of_equations long) containing the least significant word of the master face ID for each MPC equation. Either local ID (ACME_LOCAL_ID) or the least significant word in the host code global face ID (HOST_GLOBAL_ID) is returned.
- `nface_nodes` is an array (number_of_equations long) containing the number of nodes attached to the master face for this MPC equation.
- `fnode_pid` is an array (8*number_of_equations long) containing the most significant word of the master face node ID for each MPC equation. Either processor ID (ACME_LOCAL_ID) or the most significant word in the host code global node ID (HOST_GLOBAL_ID) is returned. If the master face has fewer than 8 nodes, the remaining IDs are returned as zero.
- `fnode_lid` is an array (8*number_of_equations long) containing the least significant word of the master face node ID for each MPC equation. Either local id (ACME_LOCAL_ID) or the least significant word in the host code global node id (HOST_GLOBAL_ID) is returned. If the master face has fewer than 8 nodes, the remaining IDs are returned as zero.
MPC Enforcement

fnode_coefs is an array \((8*\text{number\_of\_equations} \text{ long})\) containing the coefficients for the MPC equation. The coefficients, \(C_i\), are written such that the MPC equation is given by
\[
1 + C_1 + C_2 + \ldots = 0.
\]
If the master face has fewer than 8 nodes, the remaining coefficients are returned as zero.
error is the return error code for the C and Fortran APIs.

8.5 Destroying a ContactVolumeTransfer Object

\text{C++} \quad \sim\text{ContactMPCs}();
\text{C} \quad \text{FORTRAN(cleanup_mpc_eqns)();}
\text{Fortran} \quad \text{cleanup_mpc_eqns()}

9. Example

This section outlines a simple single-processor search example with multiple face types and an Analytic_Surface using the C++ interface. The only differences in using the C or Fortran interface would be calling the analogous C/Fortran functions (the data and calling sequence would be the same).

9.1 Problem Description

Consider the problem shown in Figure 15, where two bodies impact each other as well as an analytic plane. One body is discretized with 8-node hexahedral elements and the other is discretized with 4-node tetrahedral elements (the discretizations are not shown in Figure 15). For this example, we consider a dynamic search for NodeFace_Interactions. As previously noted, all interactions with Analytic_Surfaces are static checks, regardless of the type of search, for this version of ACME. The host code is responsible for creating a topological representation of the surface to supply to ACME. The Face_Block numbering is shown in Figure 16, the surface topology is shown in Figure 17, and the connectivities for the faces are given in Table 18.

![Current Configuration](image1)

![Predicted Configuration](image2)

Figure 15 Example impact problem (two rectangular bodies and an Analytic_Surface)

![Current Configuration](image3)

Figure 16 Face_Block Numbering for Example Problem
Because all of the nodes are attached to faces, only one Node_Block is used (this block will then have an ID of 1). For the Exodus IDs, we will simply use the local ID. For the global IDs (which are two integers), we will use (0,local_id). For this example, consider the case where the user wants to specify one set of search tolerance values between the two bodies and another set between each body and the analytic plane, as well as specifying the interaction type between each. To accommodate this, the number of Face_Blocks will be four (one for the “side” face of the left body, one for the “bottom” face of the left body, one for the “side” face of the right body and one for the “bottom” face of the right body). The total number of Entity_Keys will then be 5 (one each for the Face_Blocks and an additional one for the PLANE Analytic_Surface).

Table 18 Face_Blocks for Example Problem

<table>
<thead>
<tr>
<th>Host Code Face ID</th>
<th>Face_Block ID</th>
<th>Index in Block</th>
<th>Connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1-5-2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>2-5-3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>3-5-4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4</td>
<td>5-1-4</td>
</tr>
</tbody>
</table>
9.2 Constructing a ContactSearch Object

The code fragment below represents the call (and error checking) to construct the ContactSearch object:

```cpp
ContactSearch::ContactErrorCode error;
ContactSearch search_obj(
    dimensionality, number_of_states, number_of_entity_keys,
    number_of_node_blocks, node_block_types,
    number_of_nodes_in_blocks, node_exodus_ids,
    node_global_ids, coordinates
    number_of_face_blocks, face_block_types,
    number_of_faces_in_block, face_global_ids, face_connectivity,
    number_of_elem_blocks, elem_block_types,
    number_of_elems_in_blocks, elem_global_ids, elem_connectivity,
    number_of_nodal_comm_partners, nodal_comm_proc_ids,
    number_of_nodes_to_partner, communication_nodes,
    mpi_communicator, error );
if( error ) { // an error occurred on some processor
    int num_err = search_obj.Number_of_Errors();
    for( int i=0 ; i<num_err ; i++ )
        cout << search_obj.Error_Message(i) << endl;
    exit(error);
}
```

The data below represent the values of the arguments in the constructor:

dimensionality = 3
number_of_states = 1
number_of_entity_keys = 5
number_of_node_blocks = 1
node_block_types = \{ NODE \}
number_of_nodes_in_blocks = \{ 14 \}
node_exodus_ids = \{ 11, 8, 13, 1, 4, 17, 21, 41, 17, 33, 19, 27, 38, 16 \}
node_global_ids = \{ (0,11),(0,8),(0,13),(0,1),(0,4),(0,17),(0,19),
                         (0,27),(0,33),(0,38),(0,21) \}
```

<table>
<thead>
<tr>
<th>Host Code</th>
<th>Face_Block ID</th>
<th>Index in Block</th>
<th>Connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>2</td>
<td>1</td>
<td>4-6-3</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>2</td>
<td>4-8-6</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>3</td>
<td>8-7-6</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>4</td>
<td>6-7-3</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
<td>1</td>
<td>9-11-14-13</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>1</td>
<td>9-10-12-11</td>
</tr>
</tbody>
</table>
Example

coordinates = array of reals given in Table 19.

number_of_face_blocks = 4
face_block_types = { TRIFACE63, TRIFACE63, QUADFACEL4, QUADFACEL4 }

number_of_faces_in_block = { 4, 4, 1, 1 }
face_global_ids = { (0,1), (0,2), (0,3), (0,4), (0,5), (0,6), (0,7), (0,8), (0,9), (0,10) }

face_connectivity = { [1, 5, 2, 2, 5, 3, 3, 5, 4, 5, 1, 4], [4, 6, 3, 4, 8 ,6, 8, 7, 6, 6, 7, 3], [9, 11, 14, 13], [9, 11, 12, 10] }

number_of_elem_blocks = 0

4.3 Adding an Analytic_Surface

The next step is to add the analytic plane. Since we have already added four Face_Blocks, the ID of the PLANE Analytic_Surface will be 5. The code fragment (and error checking) to add this Analytic_Surface is:

```cpp
error = search_obj.Add_Analytic_Surface( 
analytic_surfacetype, 
data );
if( error ){
    int num_err = search_obj.Number_of_Errors();
    for( int i=0 ; i<num_err ; i++ )
        cout << search_obj.Error_Message( i ) << endl;
    exit(error);
}
```

The data needed to add the Analytic_Surface are (see Table 14 for a description of the data):

```cpp
analyticsurface_type = PLANE
data = { [0.0, 0.0, 0.0], [0.0, 1.0, 0.0] }
```

4.4 Search Data

The next step is to set the Search_Data. For this example, assume the user only wants interactions for nodes of Face_Block 2 against faces of Face_Block 3, nodes of Face_Block 3 against faces of Face_Block 2 and nodes of Face_Blocks 1 and 4 against the PLANE Analytic_Surface. We will use a Search_Normal_Tolerance of 0.01 for interactions between the two bodies and a Search_Normal_Tolerance of 0.1 for the bodies against the
PLANE Analytic_Surface. We will use Search_Tangential_Tolerance values of half the respective Search_Normal_Tolerance values. Currently, a node only has one entity key (this is a limitation of the current implementation and will be addressed in a future release). The entity_key assigned to the node is from the first face it is connected to. As a result of this limitation, we must also allow interactions to be defined between nodes from face block 1 to interact with faces from face block 3 and nodes from face block 4 to interact with faces from face block 2. The call to add these data is:

```c
search_obj.Set_Search_Data( Search_Data );
```

The search data array, with 2 x 5 x 5 values, is:

```c
Search_Data = {
    0, 0.01, 0.005 // FB1 nodes against FB1 faces
    0, 0.01, 0.005 // FB2 nodes against FB1 faces
    0, 0.01, 0.005 // FB3 nodes against FB1 faces
    0, 0.01, 0.005 // FB4 nodes against FB1 faces
    0, 0.01, 0.005 // Analytic Plane against FB1 faces (don’t exist)
    0, 0.01, 0.005 // FB1 nodes against FB2 faces
    0, 0.01, 0.005 // FB2 nodes against FB2 faces
    1, 0.01, 0.005 // FB3 nodes against FB2 faces
    1, 0.01, 0.005 // FB4 nodes against FB2 faces
    0, 0.01, 0.005 // Analytic Plane against FB2 faces (don’t exist)
    1, 0.01, 0.005 // FB1 nodes against FB3 faces
    1, 0.01, 0.005 // FB2 nodes against FB3 faces
    0, 0.01, 0.005 // FB3 nodes against FB3 faces
    0, 0.01, 0.005 // FB4 nodes against FB3 faces
    0, 0.01, 0.005 // Analytic Plane against FB3 faces (don’t exist)
    0, 0.01, 0.005 // FB1 nodes against FB4 faces
    0, 0.01, 0.005 // FB2 nodes against FB4 faces
    0, 0.01, 0.005 // FB3 nodes against FB4 faces
    0, 0.01, 0.005 // FB4 nodes against FB4 faces
    0, 0.01, 0.005 // Analytic Plane against FB4 faces (don’t exist)
    1, 0.1, 0.05 // FB1 nodes against Analytic Plane
    0, 0.1, 0.05 // FB2 nodes against Analytic Plane
    0, 0.1, 0.05 // FB3 nodes against Analytic Plane
    1, 0.1, 0.05 // FB4 nodes against Analytic Plane
    0, 0.1, 0.05 // Analytic Plane against Analytic Plane

    0, 0.01, 0.005 // FB1 nodes against FB1 faces
    0, 0.01, 0.005 // FB2 nodes against FB1 faces
    0, 0.01, 0.005 // FB3 nodes against FB1 faces
    0, 0.01, 0.005 // FB4 nodes against FB1 faces
    0, 0.01, 0.005 // Analytic Plane against FB1 faces (don’t exist)
    0, 0.01, 0.005 // FB1 nodes against FB2 faces
    0, 0.01, 0.005 // FB2 nodes against FB2 faces
    1, 0.01, 0.005 // FB3 nodes against FB2 faces
    1, 0.01, 0.005 // FB4 nodes against FB2 faces
    0, 0.01, 0.005 // Analytic Plane against FB2 faces (don’t exist)
    1, 0.01, 0.005 // FB1 nodes against FB3 faces
    1, 0.01, 0.005 // FB2 nodes against FB3 faces
    0, 0.01, 0.005 // FB3 nodes against FB3 faces
    0, 0.01, 0.005 // FB4 nodes against FB3 faces
    0, 0.01, 0.005 // Analytic Plane against FB3 faces (don’t exist)
    0, 0.01, 0.005 // FB1 nodes against FB4 faces
    0, 0.01, 0.005 // FB2 nodes against FB4 faces
    0, 0.01, 0.005 // FB3 nodes against FB4 faces
    0, 0.01, 0.005 // FB4 nodes against FB4 faces
    0, 0.01, 0.005 // Analytic Plane against FB4 faces (don’t exist)
    1, 0.1, 0.05 // FB1 nodes against Analytic Plane
    0, 0.1, 0.05 // FB2 nodes against Analytic Plane
    0, 0.1, 0.05 // FB3 nodes against Analytic Plane
    1, 0.1, 0.05 // FB4 nodes against Analytic Plane
    0, 0.1, 0.05 // Analytic Plane against Analytic Plane
}
```

### 9.5 Setting the Search Options

For this example, multiple interaction definition is necessary but normal smoothing is not needed. A value of 30 degrees will be used for the SHARP_NON_SHARP_ANGLE. The code fragment to activate multiple interactions is

```c
// Activate multiple interaction
error = Set_Search_Option(  
    ContactSearch::MULTIPLE_INTERACTIONS,  
    ContactSearch::ACTIVE,
```
where `multiple_interaction_data` is a pointer to the `SHARP_NON_SHARP_ANGLE` which has been set to 30 degrees. The code fragment to deactivate normal smoothing is

```cpp
// Deactivate normal smoothing
error = Set_Search_Option(
    ContactSearch::NORMAL_SMOOTHING,
    ContactSearch::INACTIVE,
    dummy );
if( error ){
    int num_err = search_obj.Number_of_Errors();
    for( int i=0 ; i<num_err ; i++ )
        cout << search_obj.Error_Message( i ) << endl;
    exit(error);
}
```

Since normal smoothing is being deactivated, `dummy` is a pointer to double but will never be dereferenced so its value is irrelevant.

### 9.6 Specifying Configurations

At this point the topology is completely specified. The search object can be used to compute the interactions once the configurations are specified. Since we are going to perform a dynamic search, we need to specify the current and predicted configurations for the `Node_Blocks` (in this case only one block). The current configuration was stored when the object was constructed. We show both steps here because they are both needed in the general case of a dynamic topology. The code fragment to set the configurations is:

```cpp
// Supply the current position
for( int iblk=1 ; iblk<=number_of_node_blocks ; iblk++ ){
    error = search_obj.Set_Node_Block_Configuration(
        ContactSearch::CURRENT_CONFIG,
        iblk,
        current_positions[iblk-1] );
    if( error ){
        int num_err = search_obj.Number_of_Errors();
        for( int i=0 ; i<num_err ; i++ )
            cout << search_obj.Error_Message( i ) << endl;
        exit(error);
    }
}

// Supply the predicted position
error = search_obj.Set_Node_Block_Configuration(
    ContactSearch::PREDICTED_CONFIG,
    iblk,
    predicted_positions[iblk-1] );
if( error ){
```
The current and predicted positions for the nodes are shown in Table 19.

<table>
<thead>
<tr>
<th>Node</th>
<th>Current Position</th>
<th>Predicted Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{-1.1 0.1 0.0}</td>
<td>{-0.9 -0.1 0.0}</td>
</tr>
<tr>
<td>2</td>
<td>{-1.1 0.1 1.0}</td>
<td>{-0.9 -0.1 1.0}</td>
</tr>
<tr>
<td>3</td>
<td>{-0.1 0.1 1.0}</td>
<td>{ 0.1 -0.1 1.0}</td>
</tr>
<tr>
<td>4</td>
<td>{-0.1 0.1 0.0}</td>
<td>{ 0.1 -0.1 0.0}</td>
</tr>
<tr>
<td>5</td>
<td>{-0.6 0.1 0.5}</td>
<td>{-0.4 -0.1 0.5}</td>
</tr>
<tr>
<td>6</td>
<td>{-0.1 0.6 0.6}</td>
<td>{ 0.1 0.4 0.6}</td>
</tr>
<tr>
<td>7</td>
<td>{-0.1 1.1 1.0}</td>
<td>{ 0.1 0.9 1.0}</td>
</tr>
<tr>
<td>8</td>
<td>{-0.1 1.1 0.0}</td>
<td>{ 0.1 0.9 0.0}</td>
</tr>
<tr>
<td>9</td>
<td>{ 0.1 0.1 0.0}</td>
<td>{-0.1 -0.1 0.0}</td>
</tr>
<tr>
<td>10</td>
<td>{ 1.1 0.1 0.0}</td>
<td>{ 0.9 -0.1 0.0}</td>
</tr>
<tr>
<td>11</td>
<td>{ 0.1 0.1 1.0}</td>
<td>{-0.1 -0.1 1.0}</td>
</tr>
<tr>
<td>12</td>
<td>{ 1.1 0.1 1.0}</td>
<td>{ 0.9 -0.1 1.0}</td>
</tr>
<tr>
<td>13</td>
<td>{ 0.1 1.1 0.0}</td>
<td>{-0.1 0.9 0.0}</td>
</tr>
<tr>
<td>14</td>
<td>{ 0.1 1.1 1.0}</td>
<td>{-0.1 0.9 1.0}</td>
</tr>
</tbody>
</table>

9.7 Performing the Search

The search can now be performed with the following code fragment:

```cpp
int num_err = search_obj.Number_of_Errors();
for( int i=0 ; i<num_err ; i++ )
    cout << search_obj.Error_Message( i ) << endl;
exit(error);
```

```cpp
table 19  current and predicted Positions for example problem

<table>
<thead>
<tr>
<th>node</th>
<th>current position</th>
<th>predicted position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{-1.1 0.1 0.0}</td>
<td>{-0.9 -0.1 0.0}</td>
</tr>
<tr>
<td>2</td>
<td>{-1.1 0.1 1.0}</td>
<td>{-0.9 -0.1 1.0}</td>
</tr>
<tr>
<td>3</td>
<td>{-0.1 0.1 1.0}</td>
<td>{ 0.1 -0.1 1.0}</td>
</tr>
<tr>
<td>4</td>
<td>{-0.1 0.1 0.0}</td>
<td>{ 0.1 -0.1 0.0}</td>
</tr>
<tr>
<td>5</td>
<td>{-0.6 0.1 0.5}</td>
<td>{-0.4 -0.1 0.5}</td>
</tr>
<tr>
<td>6</td>
<td>{-0.1 0.6 0.6}</td>
<td>{ 0.1 0.4 0.6}</td>
</tr>
<tr>
<td>7</td>
<td>{-0.1 1.1 1.0}</td>
<td>{ 0.1 0.9 1.0}</td>
</tr>
<tr>
<td>8</td>
<td>{-0.1 1.1 0.0}</td>
<td>{ 0.1 0.9 0.0}</td>
</tr>
<tr>
<td>9</td>
<td>{ 0.1 0.1 0.0}</td>
<td>{-0.1 -0.1 0.0}</td>
</tr>
<tr>
<td>10</td>
<td>{ 1.1 0.1 0.0}</td>
<td>{ 0.9 -0.1 0.0}</td>
</tr>
<tr>
<td>11</td>
<td>{ 0.1 0.1 1.0}</td>
<td>{-0.1 -0.1 1.0}</td>
</tr>
<tr>
<td>12</td>
<td>{ 1.1 0.1 1.0}</td>
<td>{ 0.9 -0.1 1.0}</td>
</tr>
<tr>
<td>13</td>
<td>{ 0.1 1.1 0.0}</td>
<td>{-0.1 0.9 0.0}</td>
</tr>
<tr>
<td>14</td>
<td>{ 0.1 1.1 1.0}</td>
<td>{-0.1 0.9 1.0}</td>
</tr>
</tbody>
</table>

9.7 Performing the Search

The search can now be performed with the following code fragment:

```cpp
error = search_obj.Dynamic_Search_2_Configuration();
if( error ){
    cout << "Error in Dynamic_Search:: Error Code = " << error << endl;
    int num_err = search_obj.Number_of_Errors();
    for( i=0 ; i<num_err ; i++ )
    ```
cout << search_obj.Error_Message(i) << endl;
exit(error);
}

9.8 Extracting Interactions

The following coding will extract both the NodeFace_Interactions and the NodeSurface_Interactions:

// Get the NodeFace_Interactions
int number_of_NFIs, NFI_data_size;
search_obj.Size_NodeFace_Interactions(
    number_of_NFIs,
    NFI_data_size);
if( number_of_NFIs ){
    int* NFI_node_block_ids = new int[number_of_NFIs];
    int* NFI_node_indexes_in_block = new int[number_of_NFIs];
    int* NFI_node_entity_keys = new int[number_of_NFIs];
    int* NFI_face_block_ids = new int[number_of_NFIs];
    int* NFI_face_indexes_in_block = new int[number_of_NFIs];
    int* NFI_face_procs = new int[number_of_NFIs];
    double* NFI_data = new double[number_of_NFIs*NFI_data_size];
    search.Get_NodeFace_Interactions(NFI_node_block_ids,
                                       NFI_node_indexes_in_block, NFI_node_entity_keys,
                                       NFI_face_block_ids, NFI_face_indexes_in_block,
                                       NFI_face_procs,NFI_data);
}

// Get the NodeSurface_Interactions
int number_of_NSIs, NSI_data_size;
search_obj.Size_NodeSurface_Interactions(
    number_of_NSIs,
    NSI_data_size);
if( number_of_NSIs ){
    int* NSI_node_block_ids = new int[number_of_NSIs];
    int* NSI_node_indexes_in_block = new int[number_of_NSIs];
    int* NSI_analyticsurface_ids = new int[number_of_NSIs];
    double* NSI_data = new double[number_of_NSIs*NSI_data_size];
    search.Get_NodeSurface_Interactions(NSI_node_block_ids,
                                         NSI_node_indexes, NSI_analyticsurface_ids, NSI_data);
}

Table 20 gives the data for the NodeFace_Interactions and Table 21 gives the data for the NodeSurface_Interactions.
This completes the example for one time step. It is assumed the host code would take these interactions, enforce the constraints implied by these interactions and then integrate the governing equations to the next time step. At that point, the host code can supply the cur-
rent and predicted configurations for the new time step and call the search again to define new interactions. This process can then be repeated until the analysis is complete.

9.9 ExodusII Output

An ExodusII output file can be created which contains the topology and interactions with the following code fragment:

```cpp
int iows = 8;
int compws = 8;
char OutputFileName[] = "contact_topology.exo";
int exodus_id=ex_create(OutputFileName,EX_CLOBBER,&compws,&iows );
if( search->Exodus_Output( exodus_id, time ) ) {
    cout << "Error with exodus output" << endl;
    for( i=0 ; i<search->Number_of_Errors() ; i++ )
        cout << search->Error_Message(i) << endl;
}
ex_close( exodus_id );
```

Figure 18 shows plots from the ExodusII output for this example. The analytic plane is not shown in these plots because there is no way to include this plane in the ExodusII file.

![Figure 18 ExodusII Output for Example Problem](image)

a) The topology with a vector plot of displacement.
b) NodeFace_Interaction vector plot. Note the interaction vectors push back exactly to the opposing face.
c) NodeSurface_Interaction vector plot. The “top” of the vectors represent the location of the Analytic_Surface.
Distribution

Distribution (56):

David Crane (5)
Los Alamos National Laboratory
Division-ESA Group-EA
Tech Area 16 Building 242 Office 106
Mail Stop P946
Los Alamos, NM 87545

MS0382  9141  S. W. Bova
MS0382  9141  M. W. Glass (10)
MS0380  9142  M. K. Bhardwaj
MS0380  9142  M. L. Blanford
MS0380  9142  N. K. Crane
MS0380  9142  A. S. Gullerud
MS0380  9142  J. D. Hales
MS0380  9142  M. W. Heinstein
MS0380  9142  S. W. Key
MS0380  9142  J. R. Koteras
MS0380  9142  K. H. Pierson
MS0380  9142  G. M. Reese
MS0380  9142  T. F. Walsh
MS0382  9143  H. C. Edwards
MS1111  9226  K. D. Devine
MS0378  9231  S. Carroll
MS0378  9231  R. R. Drake
MS0378  9231  T. E. Voth
MS0378  9231  M. K. Wong
MS9161  8763  P. A. Klein
MS9405  8763  R. E. Jones

Copy to:

MS0384  9100  T. C. Bickel
MS0825  9110  W. L. Hermina
MS0847  9120  P. J. Wilson
MS0824  9130  J. L. Moya
MS0384  9140  H. S. Morgan
MS0382  9141  E. A. Boucheron
MS0378  9141  Day File
MS0380  9142  K. F. Alvin
MS0382  9143  J. R. Stewart
MS0372  9126  R. A. May
MS0372  9127  J. Jung
MS0321  9200  W. J. Camp
MS1110  9210  D. E. Womble
MS0316  9230  P. Yarrington
MS0378  9231  R. M. Summers
<table>
<thead>
<tr>
<th>Code</th>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS9161</td>
<td>8763</td>
<td>E-P Chen</td>
</tr>
<tr>
<td>MS9042</td>
<td>8752</td>
<td>C. Moen</td>
</tr>
<tr>
<td>MS0612</td>
<td>9612</td>
<td>Review &amp; Approval Desk</td>
</tr>
<tr>
<td>MS0899</td>
<td>9616</td>
<td>Technical Library (2)</td>
</tr>
<tr>
<td>MS9018</td>
<td>8945-1</td>
<td>Central Technical Files</td>
</tr>
</tbody>
</table>