SANDIA REPORT
SAND2001-0913
Unlimited Release
Printed April 2001

ACME Algorithms for Contact in a Multiphysics Environment API Version 0.3a

Kevin H. Brown, Randall M. Summers, Michael W. Glass, Arne S. Gullerud,
Martin W. Heinstein, and Reese E. Jones

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2001-0913
Unlimited Release
Printed April 2001

ACME
Algorithms for Contact in a Multiphysics Environment
API Version 0.3a

Kevin H. Brown and Randall M. Summers
Computational Physics R&D Department

Micheal W. Glass
Thermal/Fluid Computational Engineering Sciences Department

Arne S. Gullerud and Martin W. Heinstein
Computational Solid Mechanics & Structural Mechanics Department

Reese E. Jones
Science-Based Materials Modeling Department

Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-0819

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

List of Figures

List of Tables

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:

```
http://www.jal.sandia.gov/SEACAS/contact/index.html
```

This document describes the Application Programming Interface (API) for the ACME search and transient dynamics enforcement library. (In an attempt to avoid confusion, capitalized terms are used in this document to refer to specific terminology for which detailed definitions are provided. A glossary of these terms is given in Appendix A.) 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. Sections 2, 3, and 4 give the details of the C++, C, and Fortran interfaces, respectively. Section 5 provides an example of how to use the C++ API. 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 will use the Fortran convention (i.e., indexes start with 1) and all floating-point data is double precision.

This beta 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 multistate support is provided in this release (i.e., ACME has no ability to revert to previous states). ACME only supports conventional nodes (shell nodes and 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. 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. Currently, the topology consists of collections of nodes, faces, and analytic surfaces. Nodes and faces 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 1.1.4). Providing the full functionality required of ACME will necessitate adding Edge_Blocks and Element_Blocks. When added, these items will be analogous to the Face_Blocks (see the description in Section 1.1.2). Also, the full functionality required of ACME will necessitate adding multiple states; for this initial release of ACME, only a single state (with one or two configurations) will be supported.

1.1.1 Node_Blocks

A Node_Block is a collection of nodes of the same type. Currently, the only type of node supported in ACME is a conventional node that has position but no additional attributes. Eventually three types of nodes will be supported:

NODE: A traditional node with position.

NODE_WITH_SLOPE: A shell node that has a first derivative as an attribute.

NODE_WITH_RADIUS: A node that has a radius as an attribute. This radius is associated with the size of a spherical domain, as with smooth particle hydrodynamics (SPH) particles.

Since in this release only one type of node is supported, currently all nodes *must* be placed in a single Node_Block.

Each Node_Block is assigned an integer identifier (ID). This ID corresponds to the order 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 and NodeSurface_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 1.1.4). Currently, 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 are supported. Other face types will be added as needed. These are provided in an enumeration in the ContactSearch header file:

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

Each Face_Block is assigned an ID. This ID corresponds to the order the blocks were specified, in the same manner IDs were assigned to Node_Blocks. This ID is used in returning NodeFace_Interactions.

1.1.3 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, 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 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.

1.1.4 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. Three values are currently permitted, provided in an enumeration in the ContactSearch header file:

```
enum Search_Interaction_Type{
      NO_INTERACTION = 0,
      SLIDING_INTERACTION,
      TIED_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 interactions between entities each time a search is executed. TIED_INTERACTION (a value of 2) requests that an interaction between entities persist and can be used for mesh tying. (The explicit transient dynamic enforcement algorithms provided with this release of ACME do not yet support enforcement of tied interactions; this capability is scheduled to be added for release 0.4.)

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 1.). 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.

Every face 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. Currently, a node 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 Search_Data array is a three-dimensional Fortran array with the following size

```
     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 either a node-face or a node-Analytic_Surface pair. The second index indicates the Entity_Key for the node in an interaction, and the third index indicates the Entity_Key for the face or Analytic_Surface in an interaction.

1.2 Search Algorithms

ACME provides three 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 1.. In this idealized example, the subtleties of what happens at the edge of a face are ignored. Any node that is outside the face, where “outside” is defined by the outward unit normal **n**, is not penetrating and has a positive Gap. Any node 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 1.1.4). The Motion_Tolerance accounts for movement of the node if two configurations are used and is computed by ACME.

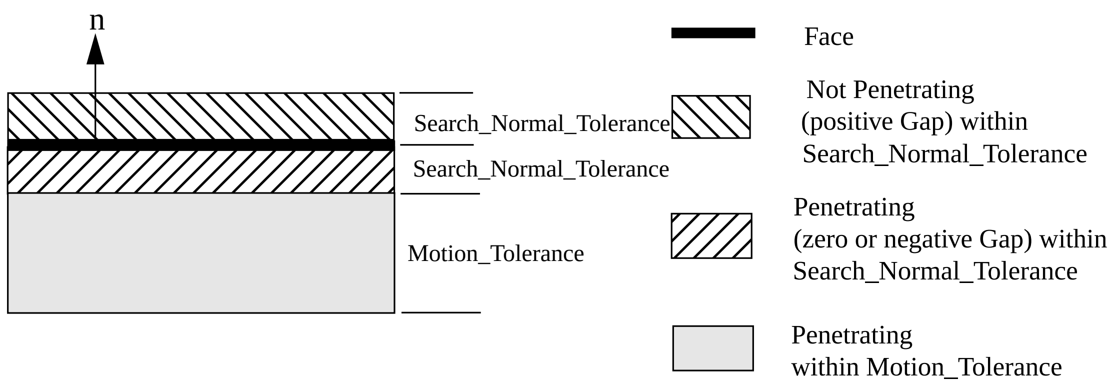

n
Search_Normal_Tolerance
Search_Normal_Tolerance
Motion_Tolerance
Face
Not Penetrating (positive Gap) within Search_Normal_Tolerance
Penetrating (zero or negative Gap) within Search_Normal_Tolerance
Penetrating within Motion_Tolerance

Figure 1. Idealized 2D face with Search_Normal_Tolerance

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 2., a node-face 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 1.1.4).

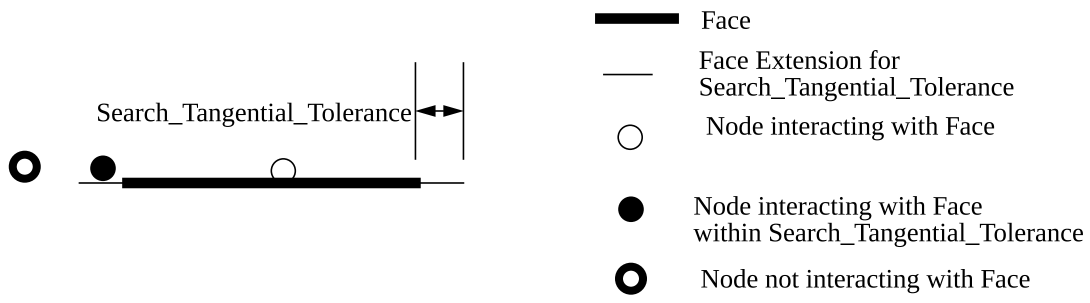

Face
Face Extension for Search_Tangential_Tolerance
Search_Tangential_Tolerance
Node interacting with Face
Node interacting with Face within Search_Tangential_Tolerance
Node not interacting with Face

Figure 2. Idealized 2D face with Search_Tangential_Tolerance

1.2.1 Static_Search _1_Configuration

The Static_Search_1_Configuration algorithm uses only one configuration for the topology. The interactions are determined using only a closest point projection algorithm. Interactions are defined only for nodes that are within the Search_Normal_Tolerance (either negative or positive Gap) and the Search_Tangential_Tolerance since the Motion_Tolerance is implied to be zero.

1.2.2 Static_Search_2_Configuration

The Static_Search_2_Configuration 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.

1.2.3 Dynamic_Search_2_Configuration

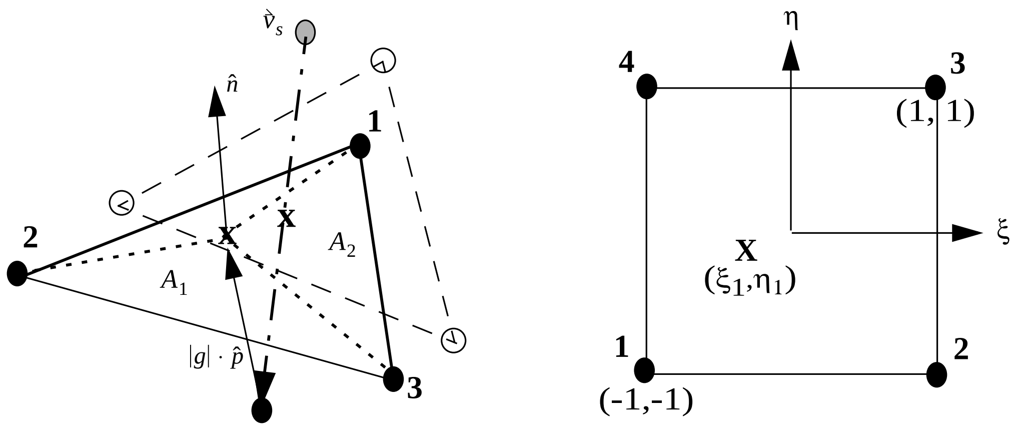
The Dynamic_Search_2_Configuration algorithm also requires two configurations (Current and Predicted) for the topology. A dynamic intersection algorithm based on linear interpolation of the motion is used to initiate interaction if the current and predicted Gaps are on opposing sides of the face (e.g., 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 criteria as in the Static_Search_2_Configuration algorithm (see Figure 1.).

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, two types of interactions are supported: NodeFace_Interactions and NodeSurface_Interactions. ACME does not determine the *best* interaction between these two 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., FaceFace_Interaction and EdgeFace_Interaction) will be added in the future.

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 3.. The first diagram illustrates an interaction defined using the dynamic intersection algorithm. 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_s. 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 1. 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.



Local Coordinates: $\xi_1 = \frac{A_1}{A_T}$
(of contact point **X**) $\xi_2 = \frac{A_2}{A_T}$
$\xi_3 = \frac{A_3}{A_T}$

Gap:	g	(not returned)
Unit Pushback Vector:	$\hat{p}$	
Unit Surface Normal:	$\hat{n}$	
Algorithm:	Dynamic Intersection	

Local Coordinates: $\xi_1 = \xi_1$
(of contact point **X**) $\xi_2 = \eta_1$

Gap:	g (not shown)
Unit Pushback Vector:	$\hat{p}$ (not shown)
Unit Surface Normal:	$\hat{n}$ (not shown)
Algorithm:	Closest Point Projection (1-Configuration)

Figure 3. 3D NodeFace_Interactions

Table 1. NodeFace_Interaction Data for 3D

Location (Fortran Indexing)	Quantity
1	Local Coordinate 1 (ξ_1 for Q4 or Q8, ξ_1 for T3 or T6)
2	Local Coordinate 2 (η_1 for Q4 or Q8, ξ_2 for T3 or T6)
3	Gap
4-6	Unit Pushback Vector (x, y & z components)
7-9	Unit Surface Normal (x, y & z components)
10	Algorithm Used to Define Interaction {1=Closest Point Projection (1 Configuration), 2=Closest Point Projection (2 Configuration), 3=Dynamic Intersection (2 Configuration)}

1.3.2 NodeSurface_Interactions

A NodeSurface_Interaction is returned as a set of data: 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 4. shows the interaction data that are returned to the host code for each interaction. Table 2. gives the layout for the data for a NodeSurface_Interaction.

For this release of ACME, NodeSurface_Interactions 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 search. This limitation will be removed in a future release.

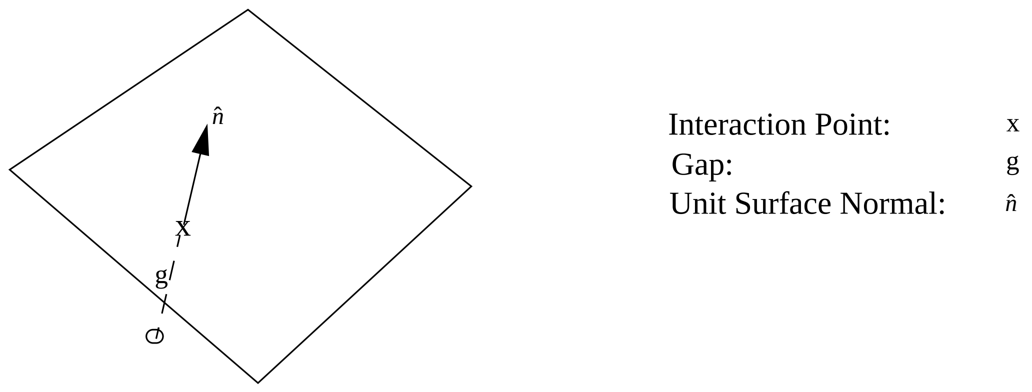


Figure 4. 3D NodeSurface_Interaction Data

Table 2. NodeSurface_Interaction Data for 3D

Location (Fortran Indexing)	Quantity
1-3	Interaction Point (x, y & z coordinates)
4	Gap
5-7	Unit Surface Normal (x, y & z components)

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 in Figure 5.) 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 6.). Interactions with disconnected faces will be returned to the host code regardless of the angle.

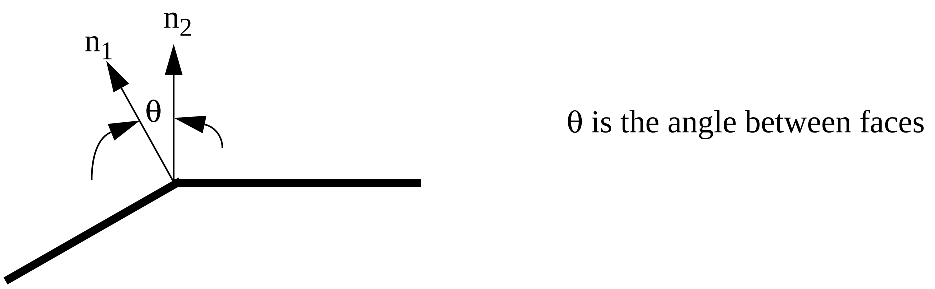


Figure 5. Definition of Angle Between Faces

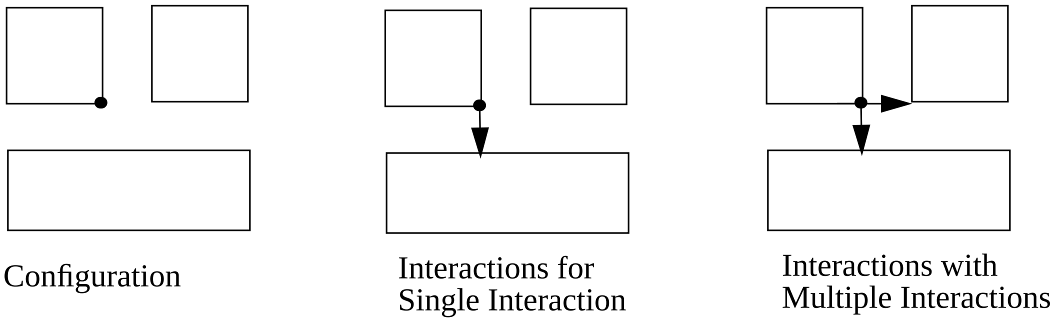

Configuration
Interactions for Single Interaction
Interactions with Multiple Interactions

Figure 6. Interactions for Single vs. Multiple Interaction Definition

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 1). The contact point, normal gap, and pushback direction are not modified by normal smoothing.

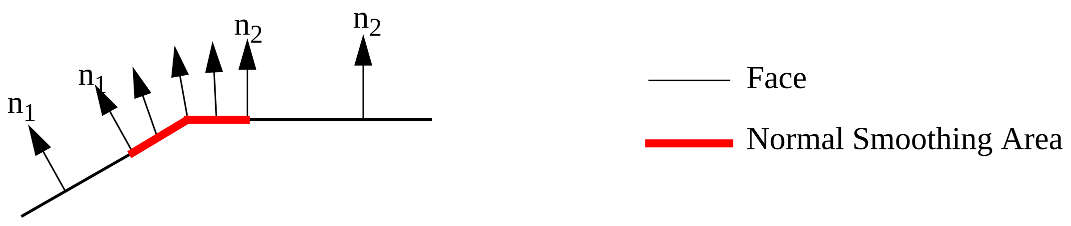


Figure 7. Normal Smoothing Across an Edge

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 8.). 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.

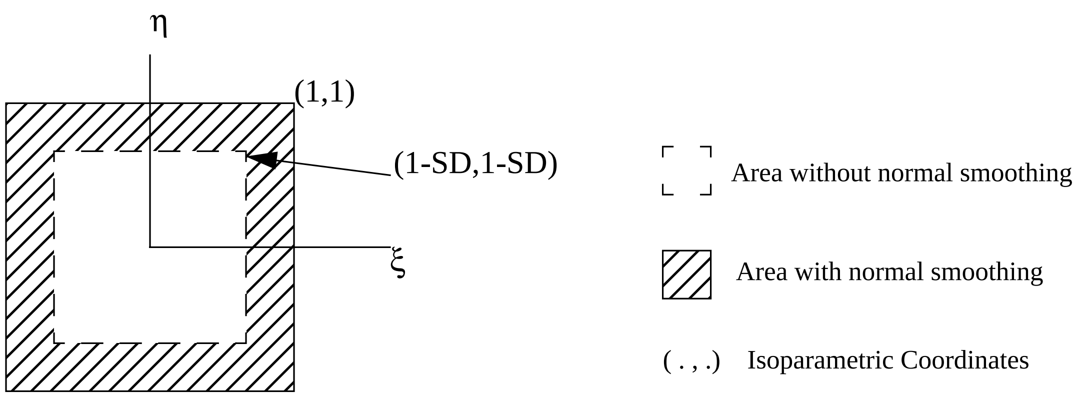

η
(1,1)
(1-SD,1-SD)
ξ
Area without normal smoothing
Area with normal smoothing
(. , .) Isoparametric Coordinates

Figure 8. Region of Normal Smoothing for a QuadFaceL4

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 9.. 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.

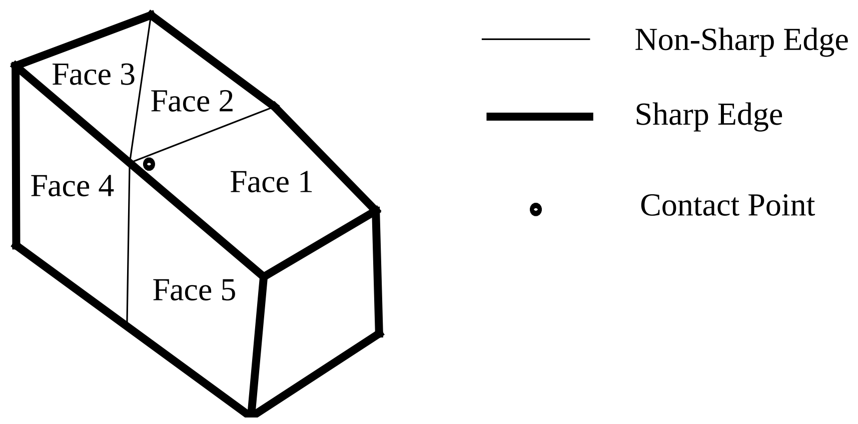


Figure 9. Illustration of Normal Smoothing Resolution

1.5 Explicit Transient Dynamic Enforcement

An optional explicit transient dynamic enforcement capability is included in this version of ACME. The algorithm was written assuming that the host code is integrating the equations of motion using a central difference integrator. It should only be used in conjunction with the Dynamic_Search_2_Configuration search method. 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 algorithms have been well tested for a single interaction per node. A beta algorithm is included to allow enforcement of multiple interactions, although it may not work for all problems. For this release, the enforcement does not operate on NodeSurface_Interactions.

1.6 Gap Removal Enforcement

An optional gap removal enforcement is also included in this version of 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.

This enforcement should be used after performing a Static_Search_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_Search_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).
6) Set the Search_Data array appropriate for the analysis.
7) Time Step using
    a) a Dynamic_Search_2_Configuration search;
    b) a ContactTDEnforcement enforcement.

1.7 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 attempting 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:

```
enum ContactErrorCode{
   NO_ERROR = 0,
   ID_NOT_FOUND,
   UNKNOWN_TYPE,
   INVALID_ID,
   INVALID_DATA,
   UNIMPLEMENTED_FUNCTION,
   EXODUS_ERROR};
```

The return value is meant as an easy check for the host code to determine if an error occurred 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., what unimplemented function was called or, possibly in the future, what 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 extract 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.

1.8 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). Additional element blocks, one for each edge type, are created to represent the edges (BAR for LineEdgeL2 and BAR3 for LineEdgeQ3). Because ExodusII does not support node blocks, all the nodes are output without their associated Node_Block.

The nodal output variables include both the nodal data (displacement and node normal) and the 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 3. gives a description of all the nodal data written to the ExodusII file.

Table 3. Nodal Variables for ExodusII Output

Name	Description
displ[xyz]	X, Y & Z components of displacement
nnorm[xyz]	X, Y & Z components of the unit node normal
numcon	number of kinematic constraints at the node
convec[xyz]	X, Y & Z components of kinematic constraint vector (provided by host)
face_id[123]	The ID of the face involved in interaction 1, 2, or 3 (0 if no interaction)
alg[123]	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)
gap[123]	The Gap for interaction 1, 2, or 3 (0 if no interaction)
pbdir[123][xyz]	X, Y, & Z components of the pushback direction for interaction 1, 2, or 3 (0 if no interaction)
ivec[123][xyz]	X, Y, & Z components of a vector that, when drawn from the node, gives the location of the interaction point for interaction 1, 2, or 3 (0 if no interaction).
norm[123][xyz]	X, Y, & Z components of the normal to the surface at the interaction point for interaction 1, 2, or 3.
iveca[xyz]	X, Y, & Z components of a vector that, when drawn from the node, gives the location of the interaction point with an Analytic_Surface (0 if no interaction). This item is included only for problems with Analytic_Surfaces.
EnfVar[xyz]	X, Y, & Z components of a vector that is the force for ContactTDEnforcement and the displacement correction for ContactGapRemoval.

The “element” data actually consist of the face and edge data (since both are output as element blocks). Table 4. gives the names and descriptions of the element data written to the ExodusII file.

Table 4. Element Variables for ExodusII Output

Name	Entity	Description
fnorm[xyz]	Faces	Unit face normal at centroid
curvature	Edge	0 = Unknown 1 = Convex 2 = Concave 3 = Concave with smoothing 4 = Convex with smoothing

2. C++ Application Programming Interface (API)

The C++ API allows for direct construction of ContactSearch, ContactTDEnforcement, and ContactGapRemoval objects. There are no static variables, so an arbitrary number of objects may be simultaneously active.

There are two constructors for the ContactSearch object. The first is intended for general use. The second is used to construct a search object for restart that is identical to the one written to a restart file in a previous calculation. The ContactSearch object is neither copyable or assignable.

There is currently only one constructor each for the ContactTDEnforcement and ContactGapRemoval objects, which are intended for general use. A constructor for restarts is not yet available since there is no internal data needed upon restart.

2.1 Version and Date

ACME provides functions to extract its current version number and release date. In addition, a function is provided to check the compile-time compatibility of the ACME library and the host code with respect to the MPI library.

2.1.1 Version

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. This version of ACME is 0.3a. The function prototype is:

```
const char* ContactSearch::Version();
```

2.1.2 VersionDate

The following function returns the release date for ACME, which is a character string of the form ‘January 5, 2000’ (the current release date). The prototype for this function is:

```
const char* ContactSearch::VersionDate();
```

2.1.3 Contact_MPI_Compatibility

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 prototype for this function is:

```
int Contact_MPI_Compatibility(int host_compile);
```

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.

2.2 Errors

As discussed in Section 1.7, there are C-style character strings that can be extracted that give a detailed description of what error(s) occurred. These strings are specific to the current processor. Therefore, each processor may have a different number of error messages.

2.2.1 Number_of_Errors

The following function determines how many error messages the current processor has:

```
int ContactSearch::Number_of_Errors();
```

2.2.2 Error_Message

The following function can be used to extract the character strings for each error message on this processor (the number of which can be determined by the function in the previous section):

```
const char* ContactSearch::Error_Message( int i );
```

where

i is the Fortran index of the error message (i.e., 1 to Number_of_Errors())

2.3 Creating a ContactSearch Object

There is one general constructor for the ContactSearch object. A second constructor for restart is described in Section 2.11.

2.3.1 ContactSearch

The prototype for this constructor is:

```
ContactSearch::ContactSearch(
   int dimensionality,
   int number_of_states,
   int number_of_entity_keys,
   int number_of_node_blocks,
   const ContactNode_Type* node_block_types,
   const int* number_of_nodes_in_blocks,
   const int* node_global_ids,
   int number_of_face_blocks,
   const ContactFace_Type* face_block_types,
   const int* number_of_faces_in_blocks,
   const int* 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 );
```

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_entity_keys is the number of entity keys that will be used. This is currently the sum of the number of Face_Blocks and the number of Analytic_Surfaces.

number_of_node_blocks is the number of Node_Blocks in the topology. Currently, since we only support one type of node (namely NODE), we only support one Node_Block.

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 (part of the ContactSearch header file) is:

```
enum ContactNode_Type{ NODE=1 };
```

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_global_ids is an array containing the host code ID for each node.

number_of_face_blocks is the number of Face_Blocks in the topology.

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 (part of the ContactSearch header file) is:

```
enum ContactFace_Type{QUADFACEL4=1, QUADFACEQ8,
                      TRIFACEL3, TRIFACEQ6};
```

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

connectivity is a one-dimensional array that gives the connectivity (using Fortran indexing in the one and only Node_Block) for each face. This may change when multiple Node_Blocks 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.

error is the error code. This reflects any errors detected during execution of this method.

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. Furthermore, any integer value can be used for mpi_communicator, which is ignored.

2.4 Search_Data

As described in Section 1.1.4, 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.

2.4.1 Check_Search_Data_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.

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

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.

2.4.2 Set_Search_Data

The following interface allows the host code to specify the Search_Data array (see Section 1.1.4), 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).

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

2.5 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 is:

```
enum Analytic_Surface_Type{
   PLANE=1, SPHERE, CYLINDER_INSIDE, CYLINER_OUTSIDE };
```

2.5.1 Add_Analytic_Surface

The interface to add an Analytic_Surface is:

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

where as_data is an array dependent on the type of surface being added, as_type. 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 10.). Table 5. gives a complete description of the array data for each Analytic_Surface type.

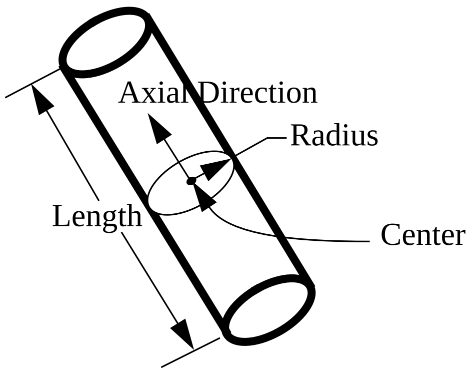


Figure 10. Analytic Cylindrical Surfaces

Table 5. C++ Data Description for Analytic_Surfaces

	Plane	Sphere	Cylinder_ Inside	Cylinder_ Outside
as_data[0]	X-Coordinate of Point	X-Coordinate of Center	X-Coordinate of Center	X-Coordinate of Center
as_data[1]	Y-Coordinate of Point	Y-Coordinate of Center	Y-Coordinate of Center	Y-Coordinate of Center
as_data[2]	Z-Coordinate of Point	Z-Coordinate of Center	Z-Coordinate of Center	Z-Coordinate of Center

Table 5. C++ Data Description for Analytic_Surfaces

	Plane	Sphere	Cylinder_ Inside	Cylinder_ Outside
as_data[3]	X-Component of Normal Vector	Radius	X-Component of Axial Vector	X-Component of Axial Vector
as_data[4]	Y-Component of Normal Vector		Y-Component of Axial Vector	Y-Component of Axial Vector
as_data[5]	Z-Component of Normal Vector		Z-Component of Axial Vector	Z-Component of Axial Vector
as_data[6]			Radius	Radius
as_data[7]			Length	Length

2.5.2 Set_Analytic_Surface_Configuration

The following interface updates the configuration(s) for an Analytic_Surface. This method has not yet been implemented in ACME, and returns an error if called.

```
ContactErrorCode ContactSearch::Set_Analytic_Surface_Configuration(
   int as_id,
   const double* as_data );
```

where

as_id is the ACME ID for the Analytic_Surface.
as_data is described in Table 5..

2.6 Node_Block Data

Currently the only valid type of Node_Block is NODE, which has no attributes. Future versions will include NODE_WITH_SLOPE and NODE_WITH_RADIUS.

2.6.1 Set_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:

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

where:

config_type is an enumeration in the ContactSearch header file:

```
enum ContactNode_Configuration{
   CURRENT_CONFIG=1,
   PREDICTED_CONFIG};
```

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.

2.6.2 Set_Node_Block_Attributes

The following function will be used to add the slope for NODE_WITH_SLOPE or the radius for NODE_WITH_RADIUS when these types are supported. Currently, this function returns an error if it is called.

```
ContactErrorCode Set_Node_Block_Attributes(
  ContactNode_Configuration config_type,
  int node_block_id,
  const double* attributes );
```

where

config_type is the type of configuration, either CURRENT_CONFIG or PREDICTED_CONFIG.
node_block_id is the ACME ID for this Node_Block.
attributes is an array of the attributes for this Node_Block.

2.7 Search Algorithms

2.7.1 Set_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 and normal smoothing.

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

where

option is an enumeration in the ContactSearch header file:

```
enum Search_Option {
   MULTIPLE_INTERACTIONS,
   NORMAL_SMOOTHING};
```

status is another enumeration in the ContactSearch header file:

```
enum Search_Option_Status {
   INACTIVE=0,
   ACTIVE};
```

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, respectively.

2.7.2 Static_Search_1_Configuration

This search algorithm can be called only after a current configuration has been specified.

```
ContactErrorCode ContactSearch::Static_Search_1_Configuration();
```

2.7.3 Static_Search _2_Configuration

This search algorithm can be called only if both current and predicted configurations have been specified.

```
ContactErrorCode ContactSearch::Static_Search_2_Configuration();
```

2.7.4 Dynamic_Search_ 2_Configuration

The dynamic search can be called only if both the current and predicted configurations have been specified.

```
ContactErrorCode ContactSearch::Dynamic_Search_2_Configuration();
```

2.8 Extracting NodeFace_Interactions

The functions in this section allow the host code to extract the NodeFace_Interactions from the ContactSearch object. Typically, the host code should determine how much memory is needed to hold the interactions using the function in Section 2.8.1 and then extract the NodeFace_Interactions using the function in Section 2.8.2.

2.8.1 Size_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.

```
void ContactSearch::Size_NodeFace_Interactions(
  int& number_of_interactions,
  int& 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 the next section).
nfi_data_size is the size of the data returned for each interaction.

2.8.2 Get_NodeFace_Interactions

The following function allows the host code to extract the active NodeFace_Interactions from the ContactSearch object. The prototype for this function is:

```
void ContactSearch::Get_NodeFace_Interactions(
  int* node_block_ids,
  int* node_indexes_in_block,
  int* face_block_ids,
  int* face_indexes_in_block,
  int* face_procs,
  double* nfi_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.
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.
face_procs is an array (of length number_of_interactions) that contains the processor that owns the face in each interaction.
nfi_data is an array (of length number_of_interactions*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 nfi_data_size locations contain the data for the first interaction).

2.9 Extracting NodeSurface_Interactions

The functions in this section allow the host code to extract the NodeSurface_Interactions from the ContactSearch object. Typically, the host code would determine how much memory is needed to hold the interactions and then extract the NodeSurface_Interactions using the functions in this section.

2.9.1 Size_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.

```
void ContactSearch::Size_NodeSurface_Interactions(
  int& number_of_interactions,
  int& nsi_data_size );
```

where

number_of_interactions are the number of active NodeSurface_Interactions that will be returned by the function Get_NodeSurface_Interactions (see the next section).
nsi_data_size is the size of the data returned for each interaction.

2.9.2 Get_NodeSurface_Interactions

The following function allows the host code to extract the active NodeSurface_Interactions from the ContactSearch object. The prototype for this function is:

```
void ConstactSearch::Get_NodeSurface_Interactions(
  int* node_block_ids,
  int* node_indexes_in_block,
  int* analyticsurface_ids,
  double* 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).

2.10 ExodusII Plotting

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 1.8 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.

2.10.1 Exodus_Output

The prototype for this capability is

```
ContactErrorCode ContactSearch::Exodus_Output(
   int exodus_id,
   double time );
```

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.

2.11 Restart Functions

The search object supports restart through a binary data stream that the host code can extract for writing to a file, and it provides a separate constructor to initialize the ContactSearch object to its previous state.

2.11.1 Restart_Size

The following function allows the host code to determine how large of an array to allocate for the ContactSearch object to give its restart information. The return value is the number of double locations that are needed.

```
int ContactSearch::Restart_Size();
```

2.11.2 Extract_Restart_Data

The following function allows the host code to extract all the information needed to initialize a ContactSearch object to its current state.

```
ContactErrorCode ContactSearch::Extract_Restart_Data(
   double* restart_data);
```

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).

2.11.3 ContactSearch (restart)

As noted above, a second constructor is available to allow for restarts:

```
ContactSearch::ContactSearch(
   const double* restart_data,
   const MPI_Comm& mpi_communicator,
   ContactErrorCode& 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).
mpi_communicator is currently unused (it is treated as int currently).
error is the error code that will reflect any errors that were detected.

2.12 Registering an Enforcement Object with the Search

To allow for enforcement data to be plotted on the optional ExodusII plot files (see section 2.10), an Enforcement object may be registered with a ContactSearch object. This is an entirely optional feature and is only useful if the host code is requesting ACME to create ExodusII plot files.

2.12.1 Register_Enforcement

The following function may be called for either a ContactTDEnforcement or a Contact-GapRemoval object. The ContactTDEnforcement object will add the contact force to the plotting database and the ContactGapRemoval object will add the displacement correction to the plotting database; both objects will store the data in variables called EnfVarx, Enf-Vary, and EnfVarz.

```
void Register_Enforcement(
  ContactEnforcement* enforcement );
```

where

enforcement is either a ContactTDEnforcement object or a ContactGapRemoval object.

2.13 Creating a ContactTDEnforcement Object

There is one general purpose constructor for the ContactTDEnforcement object. A constructor for restart use is not yet available. The only data required for restart is the EnforcementData, which the host code can get from the input deck.

2.13.1 ContactTDEnforcement

The prototype for the ContactTDEnforcement constructor is:

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

where

Enforcement_Data is a real array (of length (number of entity keys)*(number of entity keys)) which gives the kinematic partition factor (similar to Search_Data). The kinematic partition factor controls the master/slave relationship between two entities.
search is the ContactSearch object from which the topology, interactions and configurations are obtained.
error is the error code that will reflect any errors that were detected.

2.14 Extracting Contact Forces

2.14.1 Compute_Contact_Force

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

```
ContactErrorCode ContactTDEnforcement::Compute_Contact_Force(
   double dt_old,
   double dt,
   const double* mass,
   double* force );
```

where

dt_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.
force is the return array containing the computed contact forces.

2.15 Creating a ContactGapRemoval Object

There is one general purpose constructor for the ContactGapRemoval object. A constructor for restart use is not yet available. The only data required for restart is the EnforcementData which the host code can get from the input deck. The prototype for the ContactTDEnforcement constructor is:

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

where

Enforcement_Data is a real array (of length (number of entity keys)*(number of entity keys)) which gives the kinematic partition factor (similar to Search_Data). The kinematic partition factor controls the master/slave relationship between two entities.
search is the ContactSearch object from which the topology, interactions and configurations are obtained.
error is the error code that will reflect any errors that were detected.

2.16 Extracting the Gap Removal Displacements

This member function computes the displacement increments necessary to remove any initial gaps that are contained in the ContactSearch object topology. A Static_Search_1_Configuration search should be used to define the interactions prior to calling this member function (regardless of the type of mechanics being solved).

```
ContactErrorCode ContactGapRemoval::Compute_Gap_Removal(
   double displ_cor);
```

where

displ_cor is the displacement correction needed at each node to remove the initial gaps.

3. C Application Programming Interface (API)

The C API is a collection of functions that have a pure C interface. These functions operate on the ContactSearch and ContactTDEnforcement objects, only one of each of which is currently allowed. Functions are provided to allow destruction of ContactSearch or ContactTDEnforcement objects and creation of new objects at any point. Multiple objects can be supported in the future if the need ever arises.

The FORTRAN() macro converts the function by appending an underscore to the end of the function name. This macro is used because, in actuality, the C and Fortran APIs have been combined into a single interface. Because of this, in the C API, all data must be passed by address, not by value.

Two header files include the prototypes for the functions described in this chapter. The files are Search_Interface.h in the search directory and Enforcement_Interface.h in the enforcement directory.

3.1 Version and Date

ACME provides functions to extract its current version number and release date. In addition, a function is provided to check the compile-time compatibility of the ACME library and the host code with respect to the MPI library.

3.1.1 version

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. This version of ACME is 0.3a. The function prototype is:

```
void FORTRAN(version)( char* vers );
```

where

vers is an array of characters of length 81 (including terminal ‘\n’).

3.1.2 versiondate

The following function returns the release date for ACME, which is a character string of the form ‘January 5, 2000’ (the current release date). The prototype for this function is:

```
void FORTRAN(versiondate)( char* vers_date );
```

where

vers_date is an array of characters of length 81 (including terminal ‘\n’).

3.1.3 contact_mpi_compatibility

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 prototype for this function is:

```
void FORTRAN(contact_mpi_compatibility)(
  int* host_compile,
  int* error );
```

where

> host_compile is the value of MPI_COMPILE used during compilation of the host code.
> error is the error code.

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.

3.2 Errors

As discussed in Section 1.7, there are C-style character strings that can be extracted that give a detailed description of what error(s) occurred. These strings are specific to the current processor. Therefore, each processor may have a different number of error messages.

3.2.1 number_of_search_errors

The following function determines how many error messages the current processor has:

```
void FORTRAN(number_of_search_errors)( int* num_errors );
```

3.2.2 get_search_error_message

The following function can be used to extract the character strings for each error message on this processor (the number of which can be determined by the function in the previous section):

```
void FORTRAN(get_search_error_message)( int* i, char* message );
```

where

> i is the Fortran index of the error message (i.e., 1 to num_errors).
> message is an array of characters of length 81 (including terminal ‘\n’).

3.3 Creating a ContactSearch "Object"

3.3.1 build_search

The following function "constructs" a ContactSearch object for the C API. This function must be called prior to any other calls described in the API.

```
void FORTRAN(build_search)(
  int* dimensionality,
  int* number_of_states,
  int* number_of_entity_keys,
  int* number_of_node_blocks,
  int* node_block_types,
  int* number_of_nodes_in_blocks,
  int* node_global_ids,
  int* number_of_face_blocks,
  int* face_block_types,
  int* number_of_faces_in_blocks,
  int* connectivity,
  int* number_of_nodal_comm_partners,
  int* nodal_comm_proc_ids,
  int* number_of_nodes_to_partner,
  int* communication_nodes,
  MPI_Comm* mpi_communicator,
  int* 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_entity_keys is the number of entity keys that will be used. This is currently the sum of the number of Face_Blocks and the number of Analytic_Surfaces.

number_of_node_blocks is the number of Node_Blocks in the topology. Currently, since we only support one type of node (namely NODE), we only support one Node_Block.

node_block_types is an array (of length number_of_node_blocks) describing the type of nodes in each block. Currently, the only accepted type value for a Node_Block is 1 (NODE).

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_global_ids is an array containing the host code ID for each node.

number_of_face_blocks is the number of Face_Blocks in the topology.

face_block_types is an array (of length number_of_face_blocks) describing the type of faces in each Face_Block. Accepted values are QUADFACEL4=1, QUADFACEQ8=2, TRIFACEL3=3, TRIFACEQ6=4.

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

connectivity is a one-dimensional array that gives the connectivity (using Fortran indexing in the one and only Node_Block) for each face. This may change when multiple Node_Blocks 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.
error is the error code. This reflects any errors detected during execution of this method.

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. Furthermore, any integer value can be used for mpi_communicator, which is ignored.

3.4 Search_Data

As described in Section 1.1.4, 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.

3.4.1 check_search_data_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.

```
void FORTRAN(check_search_data_size)(
   int* size_data_per_pair,
   int* number_of_entity_keys,
   int* 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 error code.

3.4.2 set_search_data

The following interface allows the host code to specify the Search_Data array (see Section 1.1.4), 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).

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

3.5 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.

3.5.1 add_analytic_surface

The interface to add an Analytic_Surface is:

```
void FORTRAN(add_analytic_surface)(
   int* analytic_surface_type,
   double* as_data,
   int* error );
```

where

analytic_surface_type = 1, 2, 3, or 4 for a PLANE, SPHERE, CYLINDER_INSIDE, or CYLINDER_OUTSIDE, respectively.
as_data is dependent on the type of Analytic_Surface and is described in Table 6.
error is the error code.

3.5.2 set_analytic_surface_configuration

The following interface updates the configuration(s) for an Analytic_Surface. This method has not yet been implemented in ACME, and returns an error if called.

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

where

as_id is the ACME ID for the Analytic_Surface.
as_data is described in Table 6..
error is the error code.

Table 6. C Data Description for Analytic_Surfaces

	Plane	Sphere	Cylinder_ Inside	Cylinder_ Outside
as_data[0]	X-Coordinate of Point	X-Coordinate of Center	X-Coordinate of Center	X-Coordinate of Center

Table 6. C Data Description for Analytic_Surfaces

	Plane	Sphere	Cylinder_ Inside	Cylinder_ Outside
as_data[1]	Y-Coordinate of Point	Y-Coordinate of Center	Y-Coordinate of Center	Y-Coordinate of Center
as_data[2]	Z-Coordinate of Point	Z-Coordinate of Center	Z-Coordinate of Center	Z-Coordinate of Center
as_data[3]	X-Component of Normal Vector	Radius	X-Component of Axial Vector	X-Component of Axial Vector
as_data[4]	Y-Component of Normal Vector		Y-Component of Axial Vector	Y-Component of Axial Vector
as_data[5]	Z-Component of Normal Vector		Z-Component of Axial Vector	Z-Component of Axial Vector
as_data[6]			Radius	Radius
as_data[7]			Length	Length

3.6 Node_Block Data

Currently the only valid type of Node_Block is NODE, which has no attributes. Future versions will include NODE_WITH_SLOPE and NODE_WITH_RADIUS.

3.6.1 set_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:

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

where

config_type is the 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.
error is the error code.

3.6.2 set_node_block_attributes

The following function will be used to add the slope for NODE_WITH_SLOPE or the radius for NODE_WITH_RADIUS when these types are supported. Currently, this function returns an error if it is called.

```
void FORTRAN(set_node_block_attributes)(
  int* config_type,
  int* node_block_id,
  double* attributes,
  int* error );
```

where

config_type is the configuration (CURRENT_CONFIG = 1, PREDICTED_CONFIG = 2).
node_block_id is the ACME ID for this Node_Block.
attributes is an array of the attributes for this Node_Block.
error is the error code.

3.7 Search Algorithms

3.7.1 set_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 and normal smoothing.

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

where

option may be either 0 (MULTIPLE_INTERACTIONS) or 1 (NORMAL_SMOOTHING}.
status may be 0 (INACTIVE) or 1 (ACTIVE).
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, respectively.
error is the error code.

3.7.2 static_search_1_configuration

This search algorithm can be called only after a current configuration has been specified.

```
void FORTRAN(static_search_1_configuration)( int* error );
```

3.7.3 static_search_2_configuration

This search algorithm can be called only if both current and predicted configurations have been specified.

```
void FORTRAN(static_search_2_configuration( int* error );
```

3.7.4 dynamic_search_2_configuration

The dynamic search can be called only if both the current and predicted configurations have been specified.

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

3.8 Extracting NodeFace_Interactions

The functions in this section allow the host code to extract the NodeFace_Interactions from the ContactSearch object. Typically, the host code would determine how much memory is needed to hold the interactions using the function in Section 3.8.1 and then extract the NodeFace_Interactions using the function in Section 3.8.2.

3.8.1 size_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.

```
void FORTRAN(size_nodeface_interactions)(
  int* number_of_interactions,
  int* 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 the next section).
nfi_data_size is the size of the data returned for each interaction.

3.8.2 get_nodeface_interactions

The following function allows the host code to extract the active NodeFace_Interactions from the ContactSearch object. The prototype for this function is:

```
void FORTRAN(get_nodeface_interactions)(
  int* node_block_ids,
  int* node_indexes_in_block,
  int* face_block_ids,
  int* face_indexes_in_block,
  int* face_procs,
  double* nfi_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.

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.

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

nfi_data is an array (of length number_of_interactions*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 nfi_data_size locations contain the data for the first interaction).

3.9 Extracting NodeSurface_Interactions

The functions in this section allow the host code to extract the NodeSurface_Interactions from the ContactSearch object. Typically, the host code would determine how much memory is needed to hold the interactions and then extract the NodeSurface_Interactions using the functions in this section.

3.9.1 size_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.

```
void FORTRAN(size_nodesurface_interactions)(
  int* number_interactions,
  int* nsi_data_size );
```

where

number_of_interactions are the number of active NodeSurface_Interactions that will be returned by the function Get_NodeSurface_Interactions (see the next section).

nsi_data_size is the size of the data returned for each interaction.

3.9.2 get_nodesurface_interactions

The following function allows the host code to extract the active NodeSurface_Interactions from the ContactSearch object. The prototype for this function is:

```
void FORTRAN(get_nodesurface_interactions)(
  int* node_block_ids,
  int* node_indexes_in_block,
  int* analyticsurface_ids,
  double* 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).

3.10 ExodusII Plotting

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 1.8 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.

3.10.1 exodus_output

The prototype for this capability is

```
void FORTRAN(exodus_output)(
  int* exodus_id,
  double* time,
  int* 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 error code.

3.11 Restart Functions

The search object supports restart through a binary data stream that the host code can extract for writing to a file, and it provides a separate constructor to initialize the ContactSearch object to its previous state.

3.11.1 restart_size

The following function allows the host code to determine how large of an array to allocate for the ContactSearch object to give its restart information. The return value is the number of double locations that are needed.

```
void FORTRAN(restart_size)( int* size );
```

3.11.2 extract_restart_data

The following function allows the host code to extract all the information needed to initialize a ContactSearch object to its current state.

```
void FORTRAN(extract_restart_data)(
  double* restart_data,
  int* error);
```

where

restart_data is an array of type double. The length of this array is obtained from the function restart_size().
error is the error code. This reflects any errors detected during execution of this function.

3.11.3 build_search_restart

The following function “constructs” a ContactSearch object for restart.

```
void FORTRAN(build_search_restart)(
  double* restart_data,
  MPI_Comm* comm,
  int* error);
```

where

restart_data is an array of type double. The length of this array is obtained from the function restart_size().
comm is an MPI_Communicator.
error is the error code. This reflects any errors detected during execution of this function.

3.12 Registering an Enforcement Object with the Search

To allow for “enforcement data” to be plotted on the optional ExodusII plot files (See section 3.10), an Enforcement object may be registered with the a ContactSearch object. This is an entirely optional feature and is only useful if the host code is requesting ACME to create ExodusII plot files.

3.12.1 reg_td_enforcement_w_search

The following function may be called to register a ContactTDEnforcement “object” with the ContactSearch “object”. The ContactTDEnforcement object will add the contact force to the plotting database.

```
void FORTRAN(reg_td_enforcement_w_search)();
```

3.12.2 reg_gap_removal_w_search

The following function may be called to register a ContactGapRemoval “object” with the ContactSearch “object”. The ContactGapRemoval object will add the displacement correction to remove the initial gaps to the plotting database.

```
void FORTRAN(reg_gap_removal_w_search)();
```

3.13 Creating a ContactTDEnforcement “Object”

3.13.1 build_td_enforcement

The following function “constructs” a ContactTDEnforcement object for the C API. This function must be called prior to any other ContactTDEnforcement calls described in the API.

```
void FORTRAN(build_td_enforcement)(
   double* enforcement_data,
   int* error );
```

where

enforcement_data is a real array (of length (number of entity keys)*(number of entity keys)) which gives the kinematic partition factor (similar to Search_Data).
error is the error code.

3.14 Extracting Contact Forces

3.14.1 compute_td_contact_forces

```
void FORTRAN(compute_td_contact_forces)(
   double* dt_old,
   double* dt,
   double* mass,
    double* force,
    int* error );
```

where

dt_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.
force is the return of array of the computed contact forces.
error is the error code.

3.15 Creating a ContactGapRemoval "Object"

3.15.1 build_gap_removal

The following function "constructs" a ContactGapRemoval object for the C API. This function must be called prior to any other ContactGapRemoval calls described in the API.

```
void FORTRAN(build_gap_removal)(
    double* enforcement_data,
    int* error );
```

where

enforcement_data is a real array (of length (number of entity keys)*(number of entity keys)) which gives the kinematic partition factor (similar to Search_Data).
error is the error code.

3.16 Extracting the Gap Removal Displacements

3.16.1 compute_gap_removal

```
void FORTRAN(compute_td_contact_forces)(
    double* displ_cor,
    int* error );
```

where

displ_cor is the displacement correction needed at each node to remove the initial gaps.
error is the error code.

3.17 Clean Up

The following functions will clean up all internal memory for ACME. These actually delete the ContactSearch, ContactTDEnforcement, and ContactGapRemoval objects. Once they have been called, any other calls to the API will result in an error. These should be called prior to terminating a calculation.

3.17.1 cleanup_search

```
void FORTRAN(cleanup_search)();
```

3.17.2 cleanup_td_enforcement

```
void FORTRAN(cleanup_td_enforcement)();
```

3.17.3 cleanup_gap_removal

```
void FORTRAN(cleanup_gap_removal)();
```

4. Fortran Application Programming Interface (API)

The Fortran API is actually a collection of C functions that can be called from Fortran routines. (A FORTRAN macro is applied to these functions to append an underscore to the name, if appropriate.) These functions can then operate on the ContactSearch object, only one of which is currently allowed. Functions are provided to allow destruction of one ContactSearch Object and creation of a new object at any point. Multiple objects can be supported in the future if the need ever arises.

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.

4.1 Version and Date

ACME provides functions to extract its current version number and release date. In addition, a function is provided to check the compile-time compatibility of the ACME library and the host code with respect to the MPI library.

4.1.1 version

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. This version of ACME is 0.3a. The function prototype is:

```
version( vers )
```

where

vers is an array of characters of length 80.

4.1.2 versiondate

The following function returns the release date for ACME, which is a character string of the form ‘January 5, 2000’ (the current release date). The prototype for this function is:

```
versiondate( vers_date )
```

where

vers_date is an array of characters of length 80.

4.1.3 contact_mpi_compatibility

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 prototype for this function is:

```
contact_mpi_compatibility( host_compile, error )
```

where

> host_compile is the value of MPI_COMPILE used during compilation of the host code.
> error is the error code.

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.

4.2 Errors

As discussed in Section 1.7, there are C-style character strings that can be extracted that give a detailed description of what error(s) occurred. These strings are specific to the current processor. Therefore, each processor may have a different number of error messages.

4.2.1 number_of_search_errors

The following function determines how many error messages the current processor has:

```
number_of_search_errors( num_errors )
```

4.2.2 get_search_error_message

The following function can be used to extract the character strings for each error message on this processor (the number of which can be determined by the function in the previous section):

```
get_search_error_message( i, message )
```

where

> i is the Fortran index of the error message (i.e., 1 to num_errors)
> message is an array of characters of length 81.

4.3 Creating a ContactSearch "Object"

4.3.1 build_search

This subroutine "constructs" a ContactSearch object for the Fortran API. This subroutine must be called prior to any other calls described in the API.

```
build_search(
   dimensionality,
   number_of_states,
number_of_entity_keys,
number_of_node_blocks,
node_block_types,
number_of_nodes_in_blocks,
node_global_ids,
number_of_face_blocks,
face_block_types,
number_of_faces_in_blocks,
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_entity_keys is the number of entity keys that will be used. This is currently the sum of the number of Face_Blocks and the number of Analytic_Surfaces.
number_of_node_blocks is the number of Node_Blocks in the topology. Currently, since we only support one type of node (namely NODE), we only support one Node_Block.
node_block_types is an array (of length number_of_node_blocks) describing the type of nodes in each block. Currently, the only accepted type value for a Node_Block is 1 (NODE).
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_global_ids is an array containing the host code ID for each node.
number_of_face_blocks is the number of Face_Blocks in the topology.
face_block_types is an array (of length number_of_face_blocks) describing the type of faces in each Face_Block. Accepted values are QUADFACEL4=1, QUADFACEQ8=2, TRIFACEL3=3, and TRIFACEQ6=4.
number_of_faces_in_blocks is an array (of length number_of_face_blocks) that gives the number of faces in each Face_Block.
connectivity is a one-dimensional array that gives the connectivity (using Fortran indexing in the one and only Node_Block) for each face. This may change when multiple Node_Blocks 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.
error is the error code. This reflects any errors detected during execution of this method.

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. Furthermore, any integer value can be used for mpi_communicator, which is ignored.

4.4 Search_Data

As described in Section 1.1.4, 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.

4.4.1 check_search_data_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.

```
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 error code.

4.4.2 set_search_data

The following interface allows the host code to specify the Search_Data array (see Section 1.1.4), 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).

```
set_search_data( search_data )
```

4.5 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.

4.5.1 add_analytic_surface

The interface to add an Analytic_Surface is:

```
add_analytic_surface(
   analytic_surface_type,
   as_data,
   error )
```

where

analytic_surface_type = 1, 2, 3, or 4 for a PLANE, SPHERE, CYLINDER_INSIDE, or CYLINDER_OUTSIDE, respectively.
as_data is dependent on the type of Analytic_Surface and is described in Table 7.
error is the error code.

4.5.2 set_analytic_surface_configuration

The following interface updates the configuration(s) for an Analytic_Surface. This method has not yet been implemented in ACME, and returns an error if called.

```
set_analytic_surface_configuration(
   id,
   as_data,
   error )
```

where

as_id is the ACME ID for the Analytic_Surface.
as_data is described in Table 7..
error is the error code.

Table 7. Fortran Data Description for Analytic_Surfaces

	Plane	Sphere	Cylinder_ Inside	Cylinder_ Outside
as_data(1)	X-Coordinate of Point	X-Coordinate of Center	X-Coordinate of Center	X-Coordinate of Center
as_data(2)	Y-Coordinate of Point	Y-Coordinate of Center	Y-Coordinate of Center	Y-Coordinate of Center
as_data(3)	Z-Coordinate of Point	Z-Coordinate of Center	Z-Coordinate of Center	Z-Coordinate of Center
as_data(4)	X-Component of Normal Vector	Radius	X-Component of Axial Vector	X-Component of Axial Vector

Table 7. Fortran Data Description for Analytic_Surfaces

	Plane	Sphere	Cylinder_ Inside	Cylinder_ Outside
as_data(5)	Y-Component of Normal Vector		Y-Component of Axial Vector	Y-Component of Axial Vector
as_data(6)	Z-Component of Normal Vector		Z-Component of Axial Vector	Z-Component of Axial Vector
as_data(7)			Radius	Radius
as_data(8)			Length	Length

4.6 Node_Block Data

Currently the only valid type of Node_Block is NODE, which has no attributes. Future versions will include NODE_WITH_SLOPE and NODE_WITH_RADIUS.

4.6.1 set_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:

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

where

config_type is the 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.
error is the error code.

4.6.2 set_node_block_attributes

The following function will be used to add the slope for NODE_WITH_SLOPE or the radius for NODE_WITH_RADIUS when these types are supported. Currently, this function returns an error if it is called.

```
set_node_block_attributes(
   config_type,
   node_block_id,
   attributes,
   error )
```

where

config_type is the configuration (CURRENT_CONFIG = 1, PREDICTED_CONFIG = 2).
node_block_id is the ACME ID for this Node_Block.
attributes is an array of the attributes for this Node_Block.
error is the error code.

4.7 Search Algorithms

4.7.1 set_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 and normal smoothing.

```
set_search_option(
   option,
   status,
   data,
   error);
```

where

option may be either 0 (MULTIPLE_INTERACTIONS) or 1 (NORMAL_SMOOTHING}.
status may be 0 (INACTIVE) or 1 (ACTIVE).
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, respectively.
error is the error code.

4.7.2 static_search_1_configuration

This search algorithm can be called only after a current configuration has been specified.

```
static_search_1_configuration( error )
```

4.7.3 static_search_2_configuration

This search algorithm can be called only if both current and predicted configurations have been specified.

```
static_search_2_configuration( error )
```

4.7.4 dynamic_search_2_configuration

The dynamic search can be called only if both the current and predicted configurations have been specified.

```
dynamic_search_2_configuration( error )
```

4.8 Extracting NodeFace_Interactions

The functions in this section allow the host code to extract the NodeFace_Interactions from the ContactSearch object. Typically, the host code would determine how much memory is needed to hold the interactions using the function in Section 4.8.1 and then extract the NodeFace_Interactions using the function in Section 4.8.2.

4.8.1 size_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.

```
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 the next section).
nfi_data_size is the size of the data returned for each interaction.

4.8.2 get_nodeface_interactions

The following function allows the host code to extract the active NodeFace_Interactions from the ContactSearch object. The prototype for this function is:

```
get_nodeface_interactions(
   node_block_ids,
   node_indexes_in_block,
   face_block_ids,
   face_indexes_in_block,
   face_procs,
   nfi_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.
- 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.
- face_procs is an array (of length number_of_interactions) that contains the processor that owns the face in each interaction.
- nfi_data is an array (of length number_of_interactions*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 nfi_data_size locations contain the data for the first interaction).

4.9 Extracting NodeSurface_Interactions

The functions in this section allow the host code to extract the NodeSurface_Interactions from the ContactSearch object. Typically, the host code would determine how much memory is needed to hold the interactions and then extract the NodeSurface_Interactions using the functions in this section.

4.9.1 size_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.

```
size_nodesurface_interactions(
  number_interactions,
  nsi_data_size )
```

where

- number_of_interactions are the number of active NodeSurface_Interactions that will be returned by the function Get_NodeSurface_Interactions (see the next section).
- nsi_data_size is the size of the data returned for each interaction.

4.9.2 get_nodesurface_interactions

The following function allows the host code to extract the active NodeSurface_Interactions from the ContactSearch object. The prototype for this function is:

```
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).

4.10 ExodusII Plotting

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 1.8 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.

4.10.1 exodus_output

The prototype for this capability is

```
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 error code.

4.11 Restart Functions

The search object supports restart through a binary data stream that the host code can extract for writing to a file, and it provides a separate constructor to initialize the ContactSearch object to its previous state.

4.11.1 restart_size

The following function allows the host code to determine how large of an array to allocate for the ContactSearch object to give its restart information. The return value is the number of double locations that are needed.

```
restart_size( size )
```

4.11.2 extract_restart_data

The following function allows the host code to extract all the information needed to initialize a ContactSearch object to its current state.

```
extract_restart_data(
   restart_data,
   error)
```

where

restart_data is a double precision array whose length is obtained from the subroutine restart_size().
error is an integer error code that reflects any errors that are detected.

4.11.3 build_search_restart

This subroutine “constructs” a ContactSearch object for restart.

```
build_search_restart(
   restart_data,
   comm,
   error)
```

where

restart_data is a double precision array whose length of this array is obtained from the subroutine restart_size().
comm is an integer, currently unused.
error is an integer error code that reflects any errors that are detected.

4.12 Registering an Enforcement Object with the Search

To allow for “enforcement data” to be plotted on the optional ExodusII plot files (See section 4.10), an Enforcement object may be registered with the a ContactSearch object. This is an entirely optional feature and is only useful if the host code is requesting ACME to create ExodusII plot files.

4.12.1 reg_td_enforcement_w_search

This function may be called to register a ContactTDEnforcement "object" with the ContactSearch "object". The ContactTDEnforcement object will add the contact force to the plotting database.

```
reg_td_enforcement_w_search()
```

4.12.2 reg_gap_removal_w_search

The following function may be called to register a ContactGapRemoval "object" with the ContactSearch "object". The ContactGapRemoval object will add the displacement correction to remove the initial gaps to the plotting database.

```
reg_gap_removal_w_search()
```

4.13 Creating a ContactTDEnforcement "Object"

4.13.1 build_td_enforcement

The following subroutine "constructs" a ContactTDEnforcement object for the Fortran API. This subroutine must be called prior to any other ContactTDEnforcement calls described in the API.

```
build_td_enforcement(
  enforcement_data,
  error )
```

where

enforcement_data is a real array (of length (number of entity keys)*(number of entity keys)) which gives the kinematic partition factor (similar to Search_Data).
error is the error code.

4.14 Extracting Contact Forces

4.14.1 compute_td_contact_forces

```
compute_td_contact_forces(
  dt_old,
  dt,
  mass,
  force,
  error );
```

where

dt_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.
force is the return of array of the computed contact forces.
error is the error code.

4.15 Creating a ContactGapRemoval "Object"

4.15.1 build_gap_removal

This subroutine "constructs" a ContactGapRemoval object for the Fortran API. This subroutine must be called prior to any other ContactGapRemoval calls described in the API.

```
build_gap_removal(
  enforcement_data,
  error )
```

where

enforcement_data is a real array (of length (number of entity keys)*(number of entity keys)) which gives the kinematic partition factor (similar to Search_Data).
error is the error code.

4.16 Extracting the Gap Removal Displacements

4.16.1 compute_gap_removal

```
compute_td_contact_forces(
  displ_cor,
  error)
```

where

displ_cor is the displacement correction needed at each node to remove the initial gaps.
error is the error code.

4.17 Clean Up

The following functions will clean up all internal memory for ACME. These actually delete the ContactSearch, ContactTDEnforcement, and ContactGapRemoval objects. Once they have been called, any other calls to the API will result in an error. These should be called prior to terminating a calculation.

4.17.1 cleanup_search

```
cleanup_search()
```

4.17.2 cleanup_td_enforcement

```
cleanup_td_enforcement()
```

4.17.3 cleanup_gap_removal

```
cleanup_gap_removal()
```

5. Example

This section outlines a simple single-processor 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).

5.1 Problem Description

Consider the problem shown in Figure 11., 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 11., however). 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 12., the surface topology is shown in Figure 13., and the connectivities for the faces are given in Table 8..

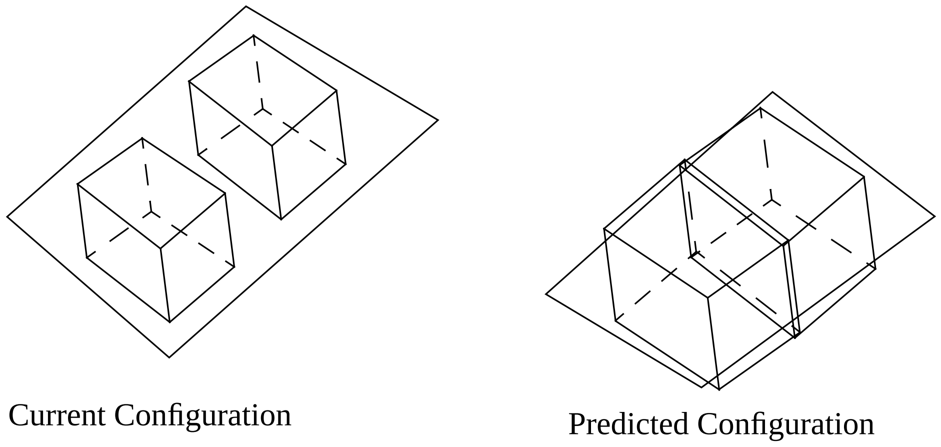


Figure 11. Example impact problem (two rectangular bodies and an Analytic_Surface)

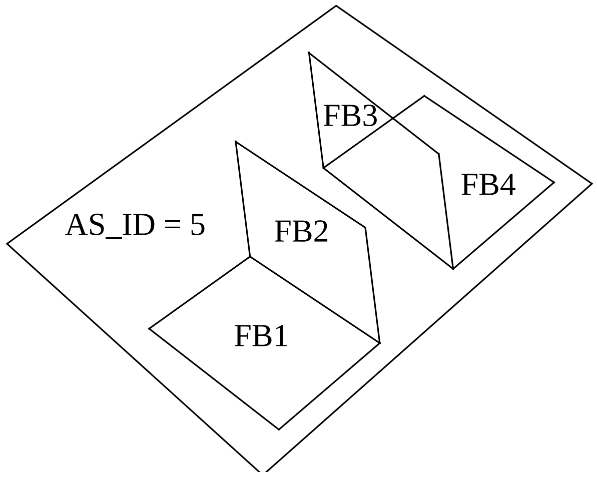


Figure 12. Face_Block Numbering for Example Problem

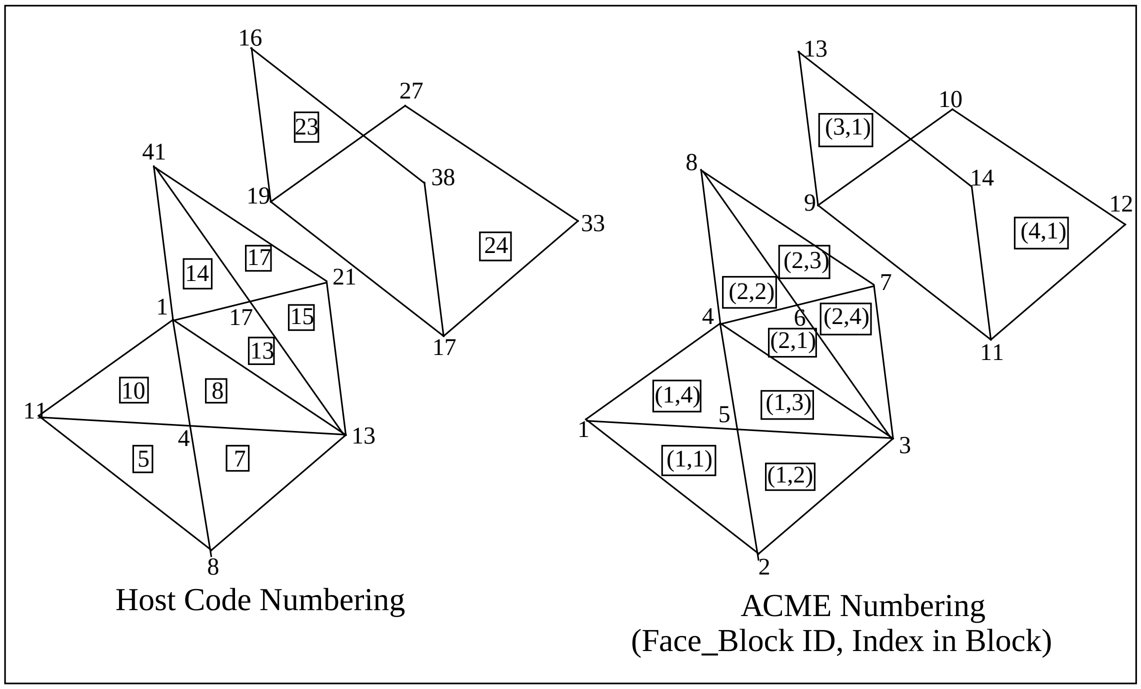


Figure 13. Surface Topology for Example Problem

As required by the current implementation, only one Node_Block is used (this block will then have an ID of 1). 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 8. Face_Blocks for Example Problem

Host Code Face ID	Face_Block ID	Index in Block	Connectivity
5	1	1	1-5-2
7	1	2	2-5-3
8	1	3	3-5-4
10	1	4	5-1-4

Table 8. Face_Blocks for Example Problem

Host Code Face ID	Face_Block ID	Index in Block	Connectivity
13	2	1	4-6-3
14	2	2	4-8-6
17	2	3	8-7-6
15	2	4	6-7-3
23	3	1	9-11-14-13
24	4	1	9-10-12-11

5.2 Constructing a ContactSearch Object

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

```
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_global_ids,
   number_of_face_blocks, face_block_types,
   number_of_faces_in_block, 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_global_ids = { 11,8,13,1,4,17,21,41,17,33,19,27,38,16 }
number_of_face_blocks = 4
face_block_types = { TRIFACEL3, TRIFACEL3, QUADFACEL4, QUADFACEL4 }
number_of_faces_in_block = { 4, 4, 1, 1 }
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_nodal_comm_partners = 0
nodal_comm_proc_ids = NULL
number_of_nodes_to_partner = NULL
communication_nodes = NULL
mpi_communicator = 0
```

5.3 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:

```
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 5. for a description of the data):

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

5.4 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:

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

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

```
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 FB4 faces (don’t exist)
   0, 0.01, 0.005 // FB1 nodes against FB4 faces
   1, 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
```

5.5 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

```
// Activate multiple interaction
error = Set_Search_Option(
   ContactSearch::MULTIPLE_INTERACTIONS,
   ContactSearch::ACTIVE,
   multiple_interaction_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);
   }
```

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

```
// 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.

5.6 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 code fragment to set the configurations is:

```
     // Supply the current position
     for( int iblk=1 ; iblk=number_of_node_blocks ; iblk++ ){
        error = search_obj.Set_NodeBlock_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_NodeBlk_Configuration(
           ContactSearch::PREDICTED_CONFIG,
           iblk,
           predicted_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);
        }
     }
```

The current and predicted positions for the nodes are shown in Table 9..

Table 9. Current and Predicted Positions for Example Problem

Node	Current Position	Predicted Position
1	{-1.1 0.1 0.0}	{-0.9 -0.1 0.0}
2	{ -1.1 0.1 1.0}	{-0.9 -0.1 1.0 }

Table 9. Current and Predicted Positions for Example Problem

Node	Current Position	Predicted Position
3	{ -0.1 0.1 1.0}	{ 0.1 -0.1 1.0}
4	{ -0.1 0.1 0.0}	{ 0.1 -0.1 0.0}
5	{ -0.6 0.1 0.5}	{ -0.4 -0.1 0.5}
6	{ -0.1 0.6 0.6}	{ 0.1 0.4 0.6}
7	{ -0.1 1.1 1.0}	{ 0.1 0.9 1.0}
8	{ -0.1 1.1 0.0}	{ 0.1 0.9 0.0}
9	{0.1 0.1 0.0}	{ -0.1 -0.1 0.0}
10	{1.1 0.1 0.0}	{0.9 -0.1 0.0 }
11	{0.1 0.1 1.0}	{ -0.1 -0.1 1.0}
12	{1.1 0.1 1.0}	{0.9 -0.1 1.0 }
13	{0.1 1.1 0.0}	{-0.1 0.9 0.0 }
14	{0.1 1.1 1.0}	{ -0.1 0.9 1.0}

5.7 Performing the Search

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

```
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);
}
```

5.8 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_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_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 10. gives the data for the NodeFace_Interactions and Table 11. gives the data for the NodeSurface_Interactions.

Table 10. NodeFace_Interactions for Example Problem

Node Block	Index in Block	Face Block	Index in Block	Local Coords	Gap	Unit Pushback Vector	Unit Surface Normal	Alg.
1	3	3	1	1, -1	-0.2	-1, 0, 0	-1, 0, 0	3
1	4	3	1	-1, -1	-0.2	-1, 0, 0	-1, 0, 0	3
1	6	3	1	0, 0	-0.2	-1, 0, 0	-1, 0, 0	3
1	7	3	1	1, 1	-0.2	-1, 0, 0	-1, 0, 0	3
1	8	3	1	-1, 1	-0.2	-1, 0, 0	-1, 0, 0	3
1	9	2	1	0, 0	-0.2	1, 0, 0	1, 0, 0	3
1	11	2	1	0, 0	-0.2	1, 0, 0	1, 0, 0	3
1	13	2	2	0, 1	-0.2	1, 0, 0	1, 0, 0	3
1	14	2	3	0, 1	-0.2	1, 0, 0	1, 0, 0	3

Table 11. NodeSurface_Interactions for Example Problem

Node Block	Index in Block	Surface ID	Gap	Interaction Point	Surface Normal
1	1	5	-0.1	-0.9, 0, 0	0, 1, 0
1	2	5	-0.1	-0.9, 0, 1	0, 1, 0
1	5	5	-0.1	-0.4, 0, 0.5	0, 1, 0
1	11	5	-0.1	-0.1, 0, 1	0, 1, 0
1	9	5	-0.1	-0.1, 0, 0	0, 1, 0
1	4	5	-0.1	0.1, 0, 0	0, 1, 0
1	3	5	-0.1	0.1, 0, 1	0, 1, 0
1	10	5	-0.1	0.9, 0, 0	0, 1, 0
1	12	5	-0.1	0.9, 0, 1	0, 1, 0

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 current 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.

5.9 ExodusII Output

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

```
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 ) ){
   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 14. 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.

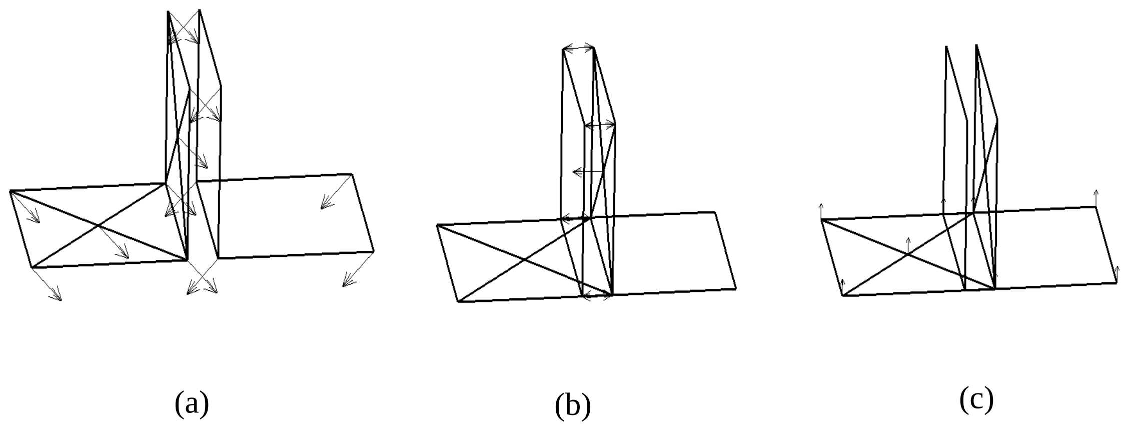


Figure 14. ExodusII Output for Example Problem

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.

Appendix A: Glossary of ACME Terms

ACME - Algorithms for Contact in a Multiphysics Environment, the current name for the search library.

Analytic_Surface - A rigid surface that can be described analytically by a geometric definition (e.g., planes and spheres).

ContactErrorCode - An error code returned by all public access functions in ACME.

ContactSearch - The top level object constructed by a host application to search for topological interactions.

ContactFace_Type - The type of faces in a Face_Block, currently QUADFACEL4, QUADFACEQ8, TRIFACEL3, or TRIFACEQ6.

ContactNode_Type - The type of nodes in a Node_Block, currently only NODE. (NODE_WITH_SLOPE and NODE_WITH_RADIUS will be available in a subsequent release.)

ContactTDEnforcement - The top level object constructed by a host application to determine forces from topological interactions found by the ContactSearch object for use in transient dynamics equations.

Dynamic_Search_2_Configuration - The search algorithm that uses a combination of a dynamic intersection and closest point projection to determine interactions.

Entity_Key - An identifier for a topological entity (currently node, face, or Analytic_Surface) used to extract user-specified parameters from the Search_Data array.

Face_Block - A collection of faces of the same type that have the same Entity_Key.

Gap - The distance between a node and a face, in the direction normal to that face in most cases, defined as positive if the node is not penetrating the face and zero or negative if the node is on or inside (penetrating) the face.

NODE - A traditional node with position and no other attributes.

Node_Block - A collection of nodes of the same type. Currently, all nodes must be placed in a single Node_Block of type NODE.

NodeFace_Interaction - A set of data returned by ACME to the host code that contains the interacting node, the face with which it interacts, and data describing the interaction (contact point in local coordinates, Normal_Gap, unit pushback vector, unit surface normal, and algorithm used).

NodeSurface_Interaction - A set of data returned by ACME to the host code that contains the interacting node, the Analytic_Surface with which it interacts, and additional data describing the interaction (contact point in global coordinates, Normal_Gap, and unit surface normal).

QUADFACEL4 - A 4-node quadrilateral face with linear interpolation.

QUADFACEQ8 - An 8-node quadrilateral face with quadratic interpolation.

Search_Data - An array containing user-specified parameters (currently three: Interaction_Status, Search_Normal_Tolerance and Search_Tangential_Tolerance) that must be set by the host code to control the search algorithms for all possible pairs of interacting topological entities.

Search_Normal_Tolerance - An absolute distance defined by the user to determine, in conjunction with any physical motion, whether two topological entities interact. This tolerance acts normal to the face.

Search_Tangential_Tolerance -An absolute distance defined by the user to determine, in conjunction with any physical motion, whether two topological entities interact. This tolerance acts tangential to the face.

Static_Search_1_Configuration - The search algorithm that uses only one configuration to determine interactions using a closest point projection.

Static_Search_2_Configuration - The search algorithm that uses two configurations, current and predicted, to determine interactions using a closest point projection.

TRIFACEL3 - A 3-node triangular face with linear interpolation.

TRIFACEQ6 - A 6-node triangular face with quadratic interpolation.

Distribution:

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

MS0321	9200	W. J. Camp
MS0321	9230	P. Yarrington
MS0819	9231	E. A. Boucheron
MS0819	9231	K. H. Brown (20)
MS0819	9231	S. Carrol
MS0819	9231	D. E. Carrol
MS0819	9231	R. M. Summers
MS0824	9112	A. C. Ratzel
MS0826	9143	H. C. Edwards
MS0826	9143	J. R. Stewart
MS0826	9114	P. R. Schunk
MS0826	9143	J. D. Zepper
MS0827	9140	J. M. McGlaun
MS0835	9141	S. W. Bova
MS0835	9141	R. J. Cochran
MS0835	9141	M. W. Glass
MS0835	9141	S. N. Kempka
MS0835	9141	R. R. Lober
MS0835	9142	K. H. Pierson
MS0841	9100	T. C. Bickel
MS0847	9124	K. F. Alvin
MS0847	9142	S. W. Attaway
MS0847	9142	M. K. Bhardwaj
MS0847	9142	M. L. Blanford
MS0847	9142	M. W. Heinstein
MS0847	9142	A. S. Gullerud
MS0847	9142	S. W. Key
MS0847	9142	J. R. Koteras
MS0847	9142	J. A. Mitchell
MS0847	9123	H. S. Morgan
MS0847	9142	J. S. Peery
MS0847	9142	G. M. Reese
MS1111	9226	K. D. Devine
MS1111	9226	C. T. Vaughan
MS9042	8728	C. Moen
MS9161	8726	E-P Chen
MS9161	8726	P. A. Klein
MS9405	8726	R. E. Jones (5)

Distribution

MS0612	9612	Review & Approval Desk
MS0899	9616	Technical Library (2)
MS9018	8945-1	Central Technical Files