SANDIA REPORT
SAND2008-7687
Unlimited Release
November 2008

FCLib:
The Feature Characterization Library

Wendy S. K. Doyle
Ann C. Gentile
W. Philip Kegelmeyer
Craig D. Ulmer

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's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-7687
Unlimited Release
November 2008

FCLib: The Feature Characterization Library

W.S.K. Doyle
Google Incorporated
WAT-337F c/o Google
651 N. 34th Street
Seattle, WA 98103

A.C. Gentile and C.D. Ulmer
Visualization and Scientific Computing
Sandia National Laboratories
P.O. Box 969 MS 9152
Livermore, CA 94551-0969

W.P. Kegelmeyer
Informatics and Decision Science
Sandia National Laboratories
P.O. Box 969 MS 9159
Livermore CA 94551-0969

Abstract

The Feature Characterization Library (FCLib) is a software library that simplifies the process of interrogating, analyzing, and understanding complex data sets generated by finite element applications.

This document provides an overview of the library, a description of both the design philosophy and implementation of the library, and examples of how the library can be utilized to extract understanding from raw datasets.

Acknowledgments

The FCLib authors would like to thank Jay Dyke, Tim Shelton, Tim Kostka, and Nathan Spencer of SNL who provided many of the problems that FCLib tools were designed to address and worked with us to develop meaningful analyses; Ken Buch of SNL, who developed Machine Learning capabilities utilizing FCLib; and Robert Banfield and Larry Hall of the University of Florida.

This research was supported by ASC's Pre and Post Processing Environments (PPPE) Data Discovery (DD) Program.

Contents

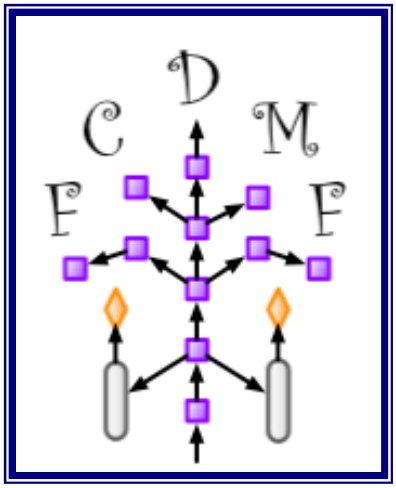
Figures

Tables

Nomenclature

ALEGRA	A family of shock and multiphysics codes developed at Sandia
DOE	Department of Energy
Exodus	A standard file format for Sandia FEM datasets
FCLib	Feature Characterization Library
FEM	Finite Element Modeling
LS-DYNA	Livermore Software's FEM tool
netCDF	Network Common Data Form: a library for managing datasets
ROI	Region of Interest
SNL	Sandia National Laboratories

1. INTRODUCTION



One very common strategy for doing data analysis in spatial-temporal datasets is to focus on regions of the dataset where interesting things are happening. If we define a feature as a coherent region that persists over time, this data analysis strategy becomes *feature-based data analysis*. The entire process of finding features, analyzing features, and using the results is called *feature characterization*.

Although feature-based data analysis is common, the feature characterization process is not very well supported. Most users manually identify and analyze features, a process which is tedious and prone to errors. Additionally, when tools are created they are usually one-off solutions that are difficult to reapply to new problems.

The goal of Sandia's Feature Characterization project (FCDMF) is to provide general resources for the creation and use of feature characterizations. The codebase developed by the FCDMF project is the Feature Characterization Library, called FCLib, which is a toolkit for creating characterizations and characterization applications. The philosophy of FCLib is to automate as many of the tedious parts of doing characterizations as possible, while remaining flexible enough to create a wide range of characterizations.

1.1 Obtaining FCLib

FCLib was developed as an open-source project. The FCLib homepage is https://fclib.ca.sandia.gov. The FCLIb code and related documentation can be found there. The FCLib development team can be reached at fclib-help@sandia.gov.

1.2 Features of FCLib

FCLib consists of a library of routines and a small number of command-line tools. The library routines can be roughly divided into the following categories:

- **Data Representation and Access**: FCLib provides its own internal data structures for representing finite element structures (e.g., mesh, elements) and associated data (e.g., variables, subsets). Rather than requiring the user to manipulate the data structures directly, FCLib provides higher-level functions which enable read/write capabilities to data, such as the mesh's coordinate values, or a variable's data values.
- **Characterization Building Blocks**: FCLib provides a number of higher-level data access or interpretation functions that can be used for building characterizations. For example, FCLib provides functions that locate elements in a mesh that share edges or faces with a given set of elements. These functions can be used as building blocks in a higher-level characterization that requires an ordered traversal of the mesh.
- **Feature Tracking**: FCLib provides facilities for managing and tracking features in a generalized manner. For example, the user can define a new feature algorithm using FCLib's data manipulation and characterization building blocks, and then use FCLib's feature processing functions to track and plot the feature as it evolves through time.

- **Characterizations**: Finally, FCLib supports a variety of characterizations. For example, the library can determine minimum and maximum values of a variable in a feature.

The categories necessarily overlap as one analyst's characterization will be another analyst's building block. The command-line tools include a few examples of generic and custom characterization applications built with FCLib.

The biggest feature of the library is that it is built to be *feature aware*--that is, to understand and operate on features. Another important feature of the library is its simplified interface. The API was written to support multiple levels of users--from those who will use the built-in "check-box" characterizations (so called because of our prototype development of a GUI that allows users to "check" the characterizations that they want) to more advanced users who wish to build their own characterizations.

FCLib is coded in C and operates on unstructured mesh data.

1.3 Data Representation and Access

One of the fundamental benefits of FCLib is that it organizes simulation data and analysis functions in a manner that allows tedious, low-level data manipulation tasks to be hidden from the end user. In order to make use of this programming environment, data is organized into the categories below. Note that some categories can be considered as “owning” another category, e.g., sequences are owned by the dataset, subsets are owned by the meshes.

- **Dataset**: A dataset is a single file that exists on disk. FCLib currently supports multiple file formats for reading and writing data. A dataset contains one or more meshes, zero or more sequences, and zero or more variables. Datasets do not contain subsets.
- **Sequence**: Datasets that house multiple timesteps employ a sequence to specify when each timestep took place.
- **Mesh**: A mesh represents the physical structure of one or more objects in a dataset. A mesh’s structure is defined by the coordinates and connectivity of its vertices. A mesh is comprised of one or more elements. These elements do not necessarily have to be contiguous in space (e.g., a single mesh in one of the bolt examples contains multiple bolts that are located at different spatial locations). Element faces and edges are inferred from the coordinates and their connectivity. Meshes contain zero or more variables and zero or more subsets. Meshes do not own sequences, but can instead reference the dataset’s sequences.
- **Variable and Sequence Variable**: A variable can be defined on a dataset or on a mesh as a whole (in essence, global variables from the perspective of the meshes), or, for a single mesh, can hold data values for each vertex, edge, face, or element. A single data value may have one or more components. A sequence variable is an array of variables for a given time sequence. All variables in a sequence variable must reference the same mesh.
- **Subset and Sequence Subset**: A subset provides a means of identifying individual locations in a mesh or variable that are of interest. A subset may contain anywhere from zero to all of the members of the object it describes. A sequence subset is comprised of one or more subsets that are associated with a sequence.

FCLib’s hierarchy of data objects may be at odds with that of a file format from which data is read. For example, Exodus defines its vertices at the global level, rather than the mesh level, and

thus there are implications in such a translation (this particular issue is covered in more detail in Section 3.3). While users do not strictly need to make use of the hierarchy of data objects FCLib provides, there are a number of built-in functions to allow users to understand the hierarchy and to locate descendants and parents in the hierarchy. For example, the fc_dump tool reads in an input file, creates the corresponding FCLib data structures, and then writes out the information as FCLib represents it in its hierarchy. In the writeout, then, sequence information is written out at a dataset level, and then on a mesh by mesh basis the program examines the mesh's coordinates and connectivities, and all of the mesh's variables, sequence variables, subsets, and sequence subsets and generates an information summary. In a similar manner it is possible to examine an item and then use parent references to ascend the hierarchy.

1.4 Characterizations and Characterization Building Blocks

FCLib provides a number of built-in, generic characterizations and characterization building blocks that enable users to implement analysis functions rapidly. The following is a list of example characterizations that are available in FCLib. These characterizations are organized by the type of data that they process.

- Mesh topology based (mesh entities are vertices, edges, faces, or elements):
    - Get mesh entity children (e.g., get vertices that make up an element).
    - Get mesh entity parents (e.g., get elements that contain a vertex).
    - Get mesh entity neighbors.
    - Skin (e.g., get the entities that make up the outer layer of a set of mesh entities).
    - Segment (separate a set of mesh entities into separate connected components).
- Mesh coordinates based:
    - Edge lengths, surface area, and region volumes.
    - Bounding boxes.
    - Centroid, variable-weighted centroid.
    - Get mesh entities within a box or sphere.
    - Kernel smooth variable.
- Variable based:
    - Variable math (e.g., add two variables to get a third).
    - Threshold (e.g. get set of entities that pass threshold criteria).
    - Statistics (min, max, mean, standard deviation)
- Time based:
    - Feature tracking
    - Entity variable history

1.5 Feature Tracking

One of the more powerful capabilities of FCLib is that it provides a general framework for feature tracking. Feature tracking refers to the process of identifying a region of interest (ROI) in a dataset and then monitoring its evolution as time progresses in the dataset. This section provides a brief discussion of how FCLib's feature tracking works. A detailed example of how this capability was used in a can crush analysis problem is provided in Section 5.4.

The first task in feature tracking is identifying one ore more ROIs that have meaning to the end user. In FCLib this task is performed through the use of characterization functions that are either built-in or supplied by the user. These functions quantify whether data points are significant or not in a particular analysis. For example, a user might employ a characterization function that (1) locates all points in the mesh where a stress value exceeds a specific tolerance and then (2) uses FCLib's segmentation functions to group nearby points into distinct ROIs. A collection of related ROI is called a feature. Multiple features (e.g., features for different time steps) are then stored in a FeatureGroup container.

The second task in feature tracking involves analyzing a set of features in order to derive relationship information about the features. FCLib provides functions for comparing and tracking differences between ROI based on their overlap. The most common operation is to use the tracking capability to monitor how a collection of features evolve over multiple timesteps. By changing the manner in which overlap is calculated between ROI, users can adjust the granularity at which parent-child relationships are extracted.

Feature graphs that depict the evolution of ROI can be written out and plotted graphically with graphviz.

2. GENERAL USE

While FCLib provides a large number of functions for data analysis, it is relatively straightforward to make use of the library and develop point tools for application-specific analysis. The library is written in C and requires that a small number of libraries be linked in at compile time with a user's application. The FCLib software distribution provides a number of tutorial examples that walk the user through the process of building analysis applications. API information for the library is documented through doxygen-generated HTML pages that are constructed when the library is built. Finally, as an open source project, the user is free to inspect both the point tools and the actual library calls in order to fully explore the library.

The library itself is arranged as a set of modules. This section provides an overview of each of these modules in order to illuminate the structure and capabilities of FCLib.

2.1 Data Types

The *data types module* in FCLib defines a number of enumerated types that help make the API flexible and more readable. In addition to performing general library management control (e.g., verbosity, return codes, etc.), these enumerated types allow a single function to be utilized with a variety of data types. Specific examples of these enumerations include the following:

- **Element Type**: A variety of fundamental element types are supported in FCLib, including points, lines, triangles, quadrilaterals, tetrahedra, pyramids, prism, hexahedra, and arbitrary shapes.
- **Data Type**: Data values in nearly all functions can be composed of many different numerical representations, such as floating point or integer.
- **Math Type**: Data values can be scalar, vector, or tensor.
- **Association Type**: This type is used to define how data values are associated with a mesh. For example, a variable may associate data values with each vertex, edge, face, or element in a mesh, or for as a single data value for a mesh or dataset.

Many of the function calls in FCLib require flags using the above data types in order to be precise about the operation that is to be performed. While at first glance this appears to make the interface complex, it reduces the total number of functions required by the API and fosters better reuse within the library.

2.2 Simple Data Objects

The *simple data objects module* provides a basic set of data management functions that are used throughout the library and are generic enough for general use. The majority of these functions are containers for storing and accessing data objects. Internally, FCLib houses container items in sorted order. This organization makes it possible to locate items rapidly. Values are sorted as they are inserted into their containers.

2.3 Data Interface

The routines in the *data interface module* section are the primary interface between the computational routines in the Feature Characterization library and the actual data. As outlined in Section 1.3, the five major data object types are datasets, sequences, meshes, variables, and subsets.

2.3.1 Dataset
A dataset serves as a container for all data relating to a simulation. Dataset objects can be created from files using FileIO operations (Section 2.3.6) or explicitly by the user without having to write the results out to a file. Moving data between datasets is also possible.

2.3.2 Sequence
A sequence is a set of values, typically time, over which a variable or subset can be defined, one such entity at each step in the sequence. FCLib can have multiple sequences, although Exodus supports only one. The sequence is associated with the entire dataset. The values of a sequence are called its “coordinates” while the number of values of a sequence is its number of “steps”.

In addition to functions for creating, destroying, and accessing sequence data and meta data, some functions exist for manipulating sequences. The latter functions include capabilities to shift and scale a sequence and to convert a sequence (or sequences) with irregular spacing into a regularly spaced sequence(s). These capabilities are intended to be used in conjunction with functions in the Series module which provides sequence-based analyses.

2.3.3 Mesh
A mesh provides basic geometry information about a structure in a dataset. Meshes are defined by two sets of values: (1) a list of coordinates for all vertices in the mesh and (2) a connectivity map which specifies the vertices that make up each element in the mesh. While all of a mesh’s elements must be the same order, it is not necessary for elements to form a contiguous region in space. The face and edge information is then built from the coordinates and connectivities. The mesh interface provides a number of commands for querying the mesh structure, including higher-level operations that extract face and edge information about the mesh.

2.3.4 Subset
A subset is a set of ids over a mesh. It is associated with some subentity of a mesh, such as vertices, elements, faces, or edges. In addition to functions for creating, destroying, and accessing the subset data and meta data, functions exist for determining the intersections and complements of a subset(s).

2.3.5 Variable
A variable is data, such as temperature, over a mesh. It is associated with some subentity of a mesh such as vertices or elements. Internally, a sequence variable is an array of variable handles associated with a sequence, with one variable per step of the sequence. In this way functions that work on a single variable can also work on a single step of a sequence variable. Often a user's computations on a sequence variable thus consist of looping over a single variable function call, one for each timestep. Global variables also exist which are a single value owned by the dataset.

In mapping onto the Exodus constructs, variables are Exodus attributes and sequence variables are Exodus results. Variables with vertex associations are Exodus nodal results (for sequence variables) and attributes (for non-sequence variables) which means that they are defined for all nodes (filled in with data value equal to 0 for any nodes in any element block that does not define that variable). Note that Exodus does not support multiple data types and everything gets converted to doubles, including chars. Exodus only stores single component variables and relies on naming conventions (endings x,y,z for vectors, xx,yy,xy for tensors) for the consumer. While the general Exodus file reader reads variable data in as single components (e.g., velocity_x), FCLib provides higher-level functions for handling multi-component data (e.g., velocity_x, velocity_y, and velocity_z are merged together into a single velocity variable with three

components). More information on FCLib's handling of Exodus variable data can be found in Section 3.3.

Variable data is lazily loaded, in the sense that a variable's data is not loaded until the user specifically requests access to the data. While FCLib generally provides a clean copy of data to a user that can be read or written without side effects, there are additional functions that provide pointer access to the data without the overhead of copying it. These functions require the user treat the data as read only and should therefore be utilized with caution.

Functions exist to:

- create, copy, and delete variables,
- get their meta data (e.g., association, number of components, data type, number of data points, and name),
- get their big data (values) or pointers to such,
- perform conversions (e.g., convert sequence variables to non-sequence variables, single-component to multi-component, and one association to another).

2.3.6 FileIO

The File IO functionality consists of an interface defined in a generic wrapper, with specific implementations defined in separate files. The specific implementations should generally not be used directly. At this time FCLib defines specific implementations for two different file types: Exodus and LS-Dyna.

Generic capabilities required in the FileIO module (with implementation in the specific implementation file) include the abilities to read and write the dataset from and to the appropriate file format, including the mesh coordinates and connectivities, sequence data, variable data, and subsets. Any file type-specific issues are handled in the specific implementations as well (e.g., Exodus Attributes). These implementations must employ data structures for holding any information that is specific to the file type. For example for subsets in Exodus, the Exodus file reader/writer must maintain Exodus SetId and Exodus Association values in order to correctly convert an Exodus set to and from the analogous FCLib representation.

Some particular design issues related to FileIO for particular formats are discussed further in Section 3.3.

2.4 Utilities

Various convenience functions are provided in the *utilities module*. A set of floating point operations are included in the module to perform comparisons between different values when precision is an issue. Additionally, the utilities module provides miscellaneous convenience utilities, such as functions for decomposing file paths into individual components (e.g., directory name, base name, and extension).

2.5 Geometric Relations

The *geometric relations module* provides functions that compute relationships between the coordinates of mesh vertices. Functions exist for deriving specific geometric quantities such as diameters, centroids, areas, volumes, and normals of subsets and meshes. Additional functions provide more general geometric relationships, such as the Euclidean distance between two vertices, the angle between two vectors, and proximity measures between subsets and between

meshes. Location information can be obtained via bounding box functions and functions to determine if one item contains another (e.g., if an element contains a point or a bounding box contains an element). Additional miscellaneous functions manipulate bounding boxes, calculate mesh deformations, smooth variable values, and determine if and where a ray intersects with a triangle. Versions of most functions exist to calculate quantities based on either the original or the displaced coordinates. Many of these functions utilize FCLib's internal data structures to produce results faster than what could be achieved by an end-user application.

2.6 Topological Relations

The *topology relations module* provides functions for computing relationships between elements and vertices of the mesh itself, with no consideration of the actual physical coordinates of the vertices. The topology functions thus include operations such as computing which elements share a vertex, which vertices are part of an element, which vertices are part of an element that shares a face with a given element, etc.

This module also includes function to obtain membership relations (such as getting a mesh entity's parents or children), entity neighbors, and higher-level connectivity information (segmenting and array of entities based on their neighbor relationships).

2.7 Variable Math

The *variable math module* provides three sets of functions for performing mathematical operations on and between variables. In general these functions create variables which contain the desired result. There are versions of these functions for both normal variables and sequence variables. All sensible operations between variables, sequence variables, and constants are supported. The functions all follow a particular naming convention that utilizes the placement of "operator" and "var" to indicate the order of the variables involved and the mathematical operator performed. The three sets of functions are as follows.

- **Built In**: The first group of functions can be used to execute simple, built-in computations such as addition or multiplication on the individual components of the inputs. They preserve or promote type as necessary with the major expectation that the resultant values are always either integers or doubles.
- **User Supplied**: The second group of functions allows the user to provide a pointer to a function that performs a computation on individual members of the inputs. These functions automatically perform data conversions (e.g., change a variable to a sequence variable) in order to make operations work.
- **Non-Standard**: Additional functions perform operations that do not match the previous two function groups. These functions perform operations when there are fundamental differences between the inputs, such as the number of components in the inputs, or outputs, such as creating magnitude variables.

The flexibility of the variable math module allows users to extend the library with new computational algorithms without having to understand all of the inner workings of the library.

2.8 Statistics Routines

The *statistics routines module* includes functions that involve simple statistics (min, max, std) over variables, sequence variables, subsets, and sequences.

2.9 Series Routines

The *series routines* module consists of functions intended for sequence based analyses. These include capabilities that (1) compare two sequence variables by generating a characteristic value representing the comparison, (2) generate characteristic values of a single sequence variable, and (3) map a sequence variable into another sequence variable. Mapping functions include window averaging routines in both non-time-series and time-series versions. The former considers number of sequence steps involved in the window and the latter considers the sequence values to be time values and is then concerned with the time range of the sequence steps involved in the window. Additional functions exist to calculate derivatives, integrals, and interpolations of sequence variables. There are also functions for comparing sequence variables by determining distances and areas between their curves. Finally there is a least squares fit.

2.10 Threshold

The *threshold* modules consists of functions that, given a variable and criteria, return a subset consisting of the entities that satisfy that criteria (e.g., returns a subset of all elements whose temperature is greater than 100).

2.11 Shape

The *shape module* is intended to give information about shapes (shapes of meshes etc.). These may have a topological flavor to them, but since they are not hierarchical (e.g., children-parent), they are being located here rather than topological relations. This is meant to work in conjunction with the Element Death module (e.g., given some information about the shape of a mesh is there some information about a dead element region that would be of interest, such as the region cutting through the shape?).

An FC_Shape is a structure that contains the number of sides of a shape, arrays of subsets of the faces making up each side, arrays of subsets of the element making up each side, and the adjacency matrix describing the relationship of the sides.

The fundamental functions create shapes from meshes or subsets. In these functions, the user specifies an angle and the faces of the mesh or subset that are traversed such that if the normals of two adjacent faces differ by an angle greater than that of the specified angle, the two faces are considered to be on different sides. This methodology can only realistically be applied to simple shapes. There are additional functions to reshape an existing shape by using a new angle or to reshape it into a shape with fewer sides. This latter function is used to merge small, perhaps curved faces into a major side.

Special functions exist for simple well-defined and well-used shapes like a screw and a thin shape. A thin shape is a shape that is in some dimension narrow and in a roughly perpendicular direction has a pair of large opposing sides that are the major sides that a user is interested in.

Once the shape is determined, additional functions in this module are used to get areas of sides and side normals, and characteristics based on the adjacency matrix, such as distance matrix, and shape ends (sides that are adjacent to only one other side) and opposing sides. This type of characterization is useful in determining if a dead element region cuts through a shape verses eroding away a part of side.

2.12 Element Death

In many simulations, changing mesh topology is approximated by allowing elements to "die". The mesh topology stays the same, but any elements that are labeled "dead" no longer participate

in the simulation. Dead elements can be used to model rips, tears and other changes in the mesh.

The input of most of the routines in this module is a subset representing a dead element region, and is assumed to have the association type of FC_AT_ELEMENT. It is also assumed that the coordinates of vertices within a dead element region cannot be trusted (the vertices on the boundary of the dead element region may be o.k. if they are still on live elements).

A dead element region does not have to be a single topological segment, but most of the results are more easily interpreted if this is true.

Functions in the *element death module* are used to determine the effect of a dead element region on a mesh or FC_Shape. In particular, functions exist to determine the "exposed skin", defined as the subset of the entities that would become exposed (become part of the mesh skin) if given elements were removed from the mesh and the "decayed skin", defined as the subset which is the intersection of a dead element region and the skin. There are also segmenting functions to determine the segmenting of a mesh or subset as a result of a dead element region (e.g., does a dead element region break the mesh, erode the side of a mesh, etc.). Further there are characterizations of the size of the dead-element region.

2.13 Feature Tracking

The *feature tracking module* provides a general framework for studying how phenomena evolve over time in a dataset. Features can be identified at timesteps and then associated with one another through time, so that the user can study the evolution of a feature.

A Region of Interest (ROI) is internally represented as a C structure with a subset that exists at a single timestep comprised of the entities that make up the ROI. As of this writing, a Feature is a C structure with an array of these subsets, one for each timestep. In future work, the array may be replaced by use of the SeqSubset (whose creation postdates the creation of the Feature Tracking data structures). Feature information is then accessed via the FeatureGroup which is a container for the results of the Feature Tracking as a whole.

This module contains the functionality by which ROI are matched up into a single Feature spanning time. It provides the machinery by which one ROI is determined to "overlap" another sufficiently to be deemed to be a single feature. Functions that determine the ROI and define the overlap are to be provided by the user. However, a default function for overlap is provided that is based on geographical overlap.

3. FCLIB DESIGN NOTES

In this section we discuss some design issues of FCLib. This is not meant to be a comprehensive design document, but rather a presentation of key points that will help improve understanding of the library's characteristics. In particular we present design issues developers must be aware of when modifying the internals of the library and when exchanging data with different file formats.

3.1 Entity and Data Terminology

The library manages multiple datasets and each dataset can have multiple meshes. A sequence is the coordinates of a parameter space orthogonal to the space of the mesh. The most common type of sequence would be the time values of a time series. A subset is a set of mesh subentities (e.g. vertices or elements). A variable is a function, such as temperature, over a mesh. It is associated with some subentity of a mesh such as vertices or elements. A sequence variable is actually an array of variable handles associated with a sequence, with one variable per step of the sequence. Similarly, a sequence subset is an array of subset handles associated with a sequence, with one subset per step of the sequence.

It is very important to note that the library makes a distinction between meta data and "big data", and that access to these is treated very differently. Big data are the really large arrays of data that we want to avoid duplicating or moving around. Currently, the coordinate arrays for the meshes and the sequences, and the data from the variables, are considered big data and everything else is meta data.

When users ask for metadata they get *copies* of the data that can be manipulated freely. Users are responsible for freeing these copies. On the other hand, when users ask for big data, they receive a *pointer* to the data (the names of these routines typically end with 'Ptr'). Users must treat this data as read only, and should never attempt to free the big data directly (memory is instead released by calling FCLib-specific functions). For performance reasons, big data is lazily loaded when possible (this is discussed in more detail in Section 3.3.1), or, when appropriate, built only when necessary (this is discussed for the mesh in Section 3.4.2)

3.2 Data Management using Opaque Handles

For robustness FCLib manages data through an "opaque handles" usage model. In opaque handles a library maintains and manipulates application data on behalf of the user. This data is referenced through a handle identifier that is different than the pointer to the actual data. This technique provides an object-oriented feel to the programming interface and discourages casual direct access to complex data structures that are managed by the library. As such, FCLib's functions require users to reference objects through a small number of strongly-typed handles.

Internally, FCLib utilizes a hierarchy to keep data organized and employs validation functions to catch instances where the user has supplied bad inputs to the library. Application data is stored in a "slot". In order to handle multiplicity, similar object slots are stored in "tables" which are effectively arrays of slots. The handle that is supplied to a user for referencing an object provides all the index information necessary for the library to either locate a slot's data or determine that the reference is invalid.

3.3 Datasets: Managing File I/O

At a fundamental level, FCLib is typically used to read in a data file, perform a characterization, and then write out results or an output data file. Given that a general design goal of FCLib is to make it a tool for performing analysis in different application domains, it was necessary to

engineer the file I/O interface in a way that multiple, diverse file formats could be supported. Additionally, the large size of the datasets involved in modern simulations motivated us to consider techniques where file operations were performed only when necessary.

In order to accommodate these requirements, FCLib was constructed with a generic FileIO API that supports multiple format-specific interfaces. This API provides a consistent interface to the user and allows format-specific translation operations to take place behind the scenes without specific guidance by the user. This interface is built upon the concept of lazy loading and the separation of metadata from big data.

3.3.1 Lazy Loading and Releasing of Big Data

Data files often have large amounts of data that is not pertinent to the user's intended characterization. For this reason, one is encouraged to implement "lazy loading" wherever possible. That is, upon determining the existence of a variable in a data file, one can build the variable's data structure, but not read in the actual data values until that data is requested. For example, in the Exodus specific FileIO module, ExodusIO, sequence variable metadata and uniquely identifying Exodus variable identification are read in during the initial load, but the data field remains NULL. Upon request of a sequence variable, the data field is checked and, if NULL, only then is the actual data read in.

Lazy loading also allows one to selectively release big data that exists in the data file in order to free up memory, since the data can always be reloaded from the file when necessary. This is similar to the concept of releasing built data, although there the data is initially built and then rebuilt rather than loaded in and re-loaded in from the file. In order to allow releasing and re-loading of the data without loss of intermediate changes, changes to the data values from the file's data values are not allowed. The "committed" flag exists to keep track of such data structures. When a data item is loaded from a file and its metadata read in, the committed flag should be set to indicate if *both* the variable data exists on the disk (that is, is committed to the disk) and is able to be lazily loaded; it should be set to zero otherwise, and it is by default. Methods that allow changes to a data structure's values (e.g, adding subset members) must then explicitly check to see if the data structure is committed before performing the change. Functions that release structures release only the committed structures.

Note that only lazily loadable data can have the committed flag set, since it is otherwise not reloadable. For example, in the Exodus module, data values for node sets, element sets, mesh variables and global sequence variables are currently read in during the load and are not lazily loadable. Thus their committed flag is not set and they will not be released upon a release call. A side effect of this is that these structures values are then allowed to change from their values in the file. You may prefer to think of the "committed" flag as a "reloadable" flag. If you want to change the values of a committed structure, you can make a new structure and copy the values of the original structure over to the new structure, and then alter the new item's values.

3.3.2 Information about Supported File Formats

In this section we highlight some issues in file formats that we use. This is not meant to be a comprehensive discussion of the file formats, but rather a calling out of some design issues and related limitations in the handling of representations of data to/from various file formats.

- **Exodus**: Exodus is a well-known file format used at Sandia FEM codes that stores data into a “meshes and variables” data model. Exodus is the preferred data format for working with FCLib because it is well documented and is currently supported by multiple applications. FCLib currently assumes that an Exodus dataset will be contained in a single Exodus file. While codes such as Sierra generate multiple Exodus files for a simulation (i.e., one file per processor node), results can be concatenated using a tool such as SierraConcat. Exodus is built on top of NetCDF, a generic database file format. However, NetCDF interfacing is handled entirely by the Exodus library.
- **LS-DYNA Input Decks**: LS-DYNA is a commercial mechanics simulation code produced by LSTC. The LS-DYNA simulation tool reads input from a keyword file that contains all the mesh information required to run a simulation. FCLib currently supports the ability to read and write these keyword files.
- **LS-DYNA Results**: An LS-DYNA simulation writes its output results to binary d3plot files. FCLib currently provides basic support for reading these files. However, it is important to note that the d3plot files are a proprietary format that is poorly documented. While we have made every attempt to be compatible with this format, we have discovered inconsistencies between the format specifications and output generated by LSTC’s simulation tools. FCLib produces warnings when known issues in the file format are discovered.
- **Sierra Input Decks**: FCLib’s FileIO module also provides basic support for Sierra input deck. This module was used in the spotweld analysis tool and may not be current with more recent releases of Sierra.

3.3.3 File Format Caveats

Differences between FCLib’s internal data structures and those of the external file formats have led to some limitations and/or inconsistencies on representing externally supplied data. The following list summarizes some of the issues to be aware of when using FCLib in conjunction with different file formats, or in expanding FCLib to handle new file formats.

Global vs Local Numbering

Exodus and LS-DYNA have a global node array and element blocks refer to these global vertices. However, FCLib was designed to be able to look at single meshes, and, to support this, it stores and renumbers the nodes local to each mesh. Nodes used in multiple meshes are thus duplicated and they will not be numbered the same (as each other or as the original numbering). Therefore the Exodus (or LS-DYNA) numbering and the FCLib numbering will be different and thus one cannot reliably compare numbers written out by FCLib with numbers in the original files. This also means that one can't compare against numbers from other tools, say Ensight, which are using the original dataset.

One option for handling this is to load up the dataset in FCLib, write it out with FCLib, and then use the rewritten dataset. After that, rewriting the dataset should not change the numbering if the meshes don't change. To accomplish this, one can use the fcconvert tool which was written to convert data from one file format to another, via FCLib’s internal data structures. In this case, one can choose both the original and the final formats to be the same. A related tool is the fcdump tool, which prints out FCLib’s internal representations of the input dataset.

Block and set name support
Dyna2names.pl converts the LS-DYNA keywords file (.k file) to a .names file which FCLib parses rather than reading them from any LS-DYNA files directly. It supports element block names, sideset names, and nodeset names only. This is a historical artifact as FCLib created the .names file as a convention first for Exodus files before Exodus stored the names of element blocks. Since then, Exodus has provided name support and so the reading and writing of .names files has been removed from the ExodusIO module.

Sequence support
Exodus and LS-DYNA only support 1 sequence: time. FCLib thus discards all but the first sequence when writing Exodus files.

Data types
Exodus does not support multiple data types. Therefore, all data values are stored as double-precision floating point values (including characters).

Multicomponent variables
Exodus only stores single component variables and relies on naming conventions (endings x,y,z for vectors, xx,yy,xy for tensors) for the consumer. The Exodus reader reads in all variables as single component variables. Our other readers package vectors into multicomponent variables. All the FCLib routines are intended to handle multicomponent variables. FCLib provides functions to automatically discover and merge a group of similarly-named, single-component variables into a multicomponent variable.

Exodus general data support issues
There are some concepts that Exodus does not support, or previously did not support, but will be supporting in upcoming versions. This affects which external concepts can be created in an FCLib dataset, and which FCLib concepts may be subsequently dropped when the dataset is written out in a file format.

- ***Edge and face support***
  Exodus will be supporting explicit definitions of faces and edges and their relationships to each other and to elements. FCLib currently does not support this, but supports the older convention of sidesets. A sideset is a set of faces on a 3D mesh or edges on a 2D mesh, which are stored as a global element/local side ID pair.
- ***Blocks and Subsets***
  Similarly, Exodus will be supporting nearly all types of blocks and subsets, and attributes and results upon them. FCLib currently supports only element blocks. FCLib reads in and write out only node and element sets and sidesets (for faces and edges).
- ***Sequence variables and non-sequence variables***
  FCLib reads in and write out node and element attributes as non-sequence variables and node and element results as sequence variables. Note that for vertex associations, these are exodus nodal results (for sequence variables) and attributes (for non-sequence variables), which means that they are defined for all nodes (filled in with 0 for any nodes in any element block that does not define that variable).

FCLIB ATTRIBUTE TYPE	NON_SEQ VAR	SEQ VAR
FC_AT_WHOLE_DATASET	N	Y
FC_AT_WHOLE_MESH	N (except DT_INT)	N
FC_AT_VERTEX	Y	Y
FC_AT_EDGE	Y	N
FC_AT_FACE	N	N
FC_AT_ELEMENT	N	Y

Table 1: FCLib variable concepts and Exodus Support. Letters indicate current support. Colors indicate possible support.

Table 1 shows the mapping between FCLib variable concepts and those of Exodus. If Exodus concept is currently read in or written out it is indicated by "Y"; if not, by "N". Those FCLib vars that can eventually be supported as Exodus vars (results) are colored in green. Those that can be supported by using an Exodus concept other than variables are shown in yellow. Those that cannot be supported at all are shown in red. The remaining block (white) is described in more detail below.

Exodus only has variable support for sequence variables. All of the FCLib sequence variables can be directly mapped into Exodus results, with the exception of FC_AT_WHOLE_MESH. These are shown in green and red, respectively. FCLib's FC_AT_WHOLE_DATASET is an Exodus global result.

Exodus does not have support for the corresponding non-sequence variables, however some of these can be handled by other means. FCLib’s non sequence nodal and element variables correspond to Exodus’s nodal and element attributes. Non-sequence variables with association FC_AT_WHOLE_DATASET cannot be supported, because there is no global attribute in Exodus, and thus the relevant table cell is colored red.

Non-sequence variables with association FC_AT_MESH cannot be supported with the exception of those of datatype FC_DT_INT. Support for this case is currently implemented by the Exodus "property".

3.4 Mesh

In this section we describe some design issues of the Mesh.

3.4.1 Internal Types

FC_ElementTypes describe the elements in a mesh. Given an FC_ElementType, the number of vertices, the number of edges, the number of faces, the topological dimension, and its corresponding FaceType are established (and there are calls to get these).

FC_AssociationTypes are used to describe the association of items with a mesh or dataset. For example, a data field may be associated with the elements of a mesh (i.e., able to vary from element to element), or with an entire mesh, or with all meshes in a dataset. Variables and subsets have Association Types. Non-global variables and subsets cannot have Association Type FC_AT_WHOLE_DATASET. Global variables can only have association FC_AT_WHOLE_DATASET. This latter means that Global Variables can only be single values; one cannot have, for instance, a global variable that is a data field, varying from element to

element in a mesh, with the same across all meshes, defined only once as a global variable.

3.4.2 Mesh Structure

Generally you could specify a mesh by the coordinates of its vertices, the connectivities of the vertices, and the element-to-vertex mappings. Other things, such as faces and edges, are then implied. Thus the FCLib Mesh structure object consists of:

- Metadata comprised of:
  - items such as the topological dimension of the mesh, the number of elements, etc.
  - ids for owned entities such as subsets, variables, etc.
- Big Data which is provided by the file defining the mesh:
  - the coordinate array
  - element to vertex connectivity information
- Big Data which is built when needed:
  - things such as the edge, face, and neighbor info

This distinction in the types of Big Data is important for at least the following reasons, which will be discussed in more detail below:

- You must be careful when writing functions that add data to the mesh (such as those involved in writing new readers) to be sure that in case additional data needs to be built, that it will be built properly. The FCLib Mesh API is set up to ensure that the right building will occur and if you bypass this to add things into the mesh structure directly you may bypass this building.
- Access to and write out of the built data may behave differently than that of other data in order to not waste space on things that may not be needed. Some interfaces to that data are optimized so that things are not rebuilt, and some things that are built are not written out should you want to write to a file, because they can just be built again.

3.4.3 Built Data for Meshes

The built data for a mesh consists of three types:

- ***downward connectivities*** - these refer to parent-to-child type relationships, such as: given a face, what are all its vertices, or given an element, what are the IDs of its edges? These are stored as fixed length (size determined by the topology) integer arrays. In downward connectivities the order of the child items (for a given parent) is important.
- ***upward connectivities*** - these refer to child-to-parent type relationships, such as: given a face, what are its parent element IDs? Since the number of these can vary per child, for each type of relationship there is an array that contains the number of parents for each particular child and an additional doubly indexed array that contains the actual ID. In upward connectivities, the ordering of the parent items (for a given child) is arbitrary.
- ***neighbors*** - these are peer relationships, such as the element IDs of the neighbors of a given element. The definition of these varies depending on the level of connectivity desired - for instance, the number of neighboring elements connected by a face vs. the number of neighboring elements connected by a vertex. As in the upward connectivities, these vary for each item, and therefore both an array of the number of neighbors for each case and a doubly indexed array containing the actual ids is kept.

Once upon a time, a number of different relationships for items were kept together in structures, however this was dropped in favor of the arrays primarily because it allows the writing of a

smaller set of operations that only need to work on arrays and thus can be passed any of a number of the different array options.

3.4.4 Building Mesh Data

There are five helper functions that cause this building when needed:

- _fc_buildEdgeConns
- _fc_buildFaceConns
- _fc_buildParents
- _fc_buildMeshVertexNeighborsViaEdge
- _fc_buildMeshElementNeighborsViaEntity

Higher level functions that need this information will call these helper functions that will build this data as needed. Details of these functions are beyond the scope of this document, but more detail can be found within documentation in the FCLib release. These will additionally make any other information that it is convenient to create simultaneously.

3.4.5 Releasing the Mesh

ReleaseMesh will release as much of the big and built data as it can. The big data released will be that data that was lazily loaded. The built data includes parent and neighbor data and the edge and face information. Therefore, if you want to keep only part of the data (if, for instance you are running low on space), you actually have to release all the data and then rebuild only the part you want. In the future, we would like to support releasing data on a finer granularity.

4. LIBRARY TESTING

A key challenge in developing and maintaining a software library is verifying that updates to the library do not (1) affect the correctness of the software (negatively) or (2) cause applications that depend on the library to break. Given that FCLib has been deployed at a number of computing installations around Sandia, we decided that it was important to implement a testing infrastructure that allowed us to produce high-quality releases. This testing infrastructure is comprised of three components: unit tests, regression tests, and software quality analysis tests.

4.1 Unit Tests

The unit testing facility builds a special test program that performs numerous operations with FCLib's functions. Each module is tested with a large number of inputs. First, each function call is tested with bad inputs to verify that the function catches the input error and returns the proper error code. Second, a number of good inputs are passed to each function to verify that the proper output results are produced. The expected results are encapsulated in the testing software. As such, the unit test program provides a great deal of assurance that the software is operating properly, and helps verify that changes to the library do not break its existing functionality.

4.2 Regression Tests

The regression testing facility is responsible for running FCLib's point tools with multiple datasets and comparing the output results to known-good results. These regression tests validate that the tools still produce the same results when changes are made to the library. The regression tests were especially useful when installing FCLib on multiple platforms because they are agnostic about the inner workings of the platform: ultimately all that matters is whether the output results were produced properly. This testing proved to be especially useful when porting FCLib to different platforms because it helped identify floating-point precision issues in particular systems.

4.3 Software Quality Analysis Testing

During the development process for FCLib, we utilized multiple design analysis tools to help improve the software quality of the library. First, we employed valgrind to help identify memory leaks and programming errors in both the library and the point tools. The valgrind tool replaces C's memory management routines with profiling routines that track every block of memory that is allocated during runtime. When the program ends, valgrind reports a summary of all the blocks that were not properly freed by the program. As such we utilized valgrind on our nightly unit tests to identify leaks. To our knowledge, there are no memory leaks in the current release.

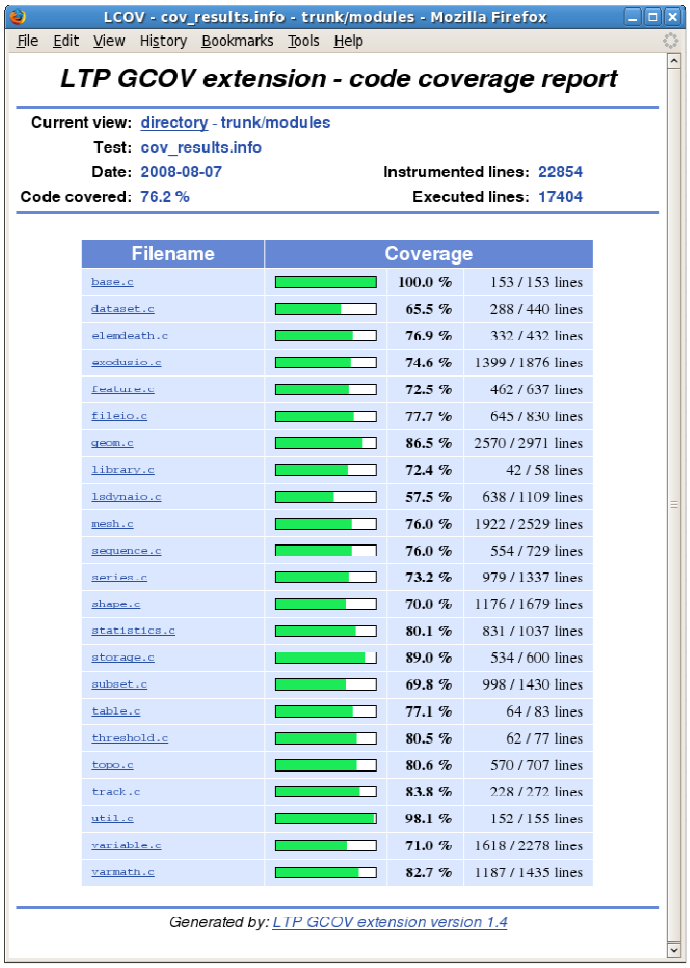


Figure 1: Code coverage estimates for FCLib's modules.

Second, we utilized the GNU profiling and coverage tools (i.e., gcov) to give an estimate of how thoroughly our tests were testing the library. In general, most of the modules in the library are covered at greater than 70% (i.e., 70% of the source code is tested). We examined the coverage results and determined that many of the untested portions of the library are either non-critical operations (e.g., printing warnings) or in redundant error checks (e.g., a function calls another function and both do checking on input data).

4.4 Nightly Testing and Documentation

While developing FCLib we utilized a stand-alone workstation to automate the testing process. This workstation used a cron job script to do the following operations:

- Download the latest version of the FCLib from the subversion repository
- Compile all of the software to a local directory
- Generate the doxygen web documentation and post the pages to a web server
- Run the unit and regression tests
- Run valgrind on the unit test to local memory leaks
- Log the results to a web page and email a copy to members of the development team

5. EXAMPLE TOOLS AND CAPABILITIES

In this section we describe some of the tools built with FCLib that demonstrate some of the more interesting capabilities and characterizations utilizing FCLib. Note that the FCLib distribution also includes a number of simpler tools that are not discussed here. These tools are quite useful in day-to-day analysis (e.g., normalizing values, bounding regions that contain maximum/minimum values of interest, quantizing statistics, etc.).

5.1 Gaplines

The gaplines toolset discovers and characterizes gaps that occur between meshes as a result of deformation of the meshes involved. The tool first determines which meshes are initially abutting (by examining the initial proximity of the vertices in the meshes) and then creates lines (initially of zero length) between their surfaces. As the meshes deform, the line lengths are updated. If they lengthen, a gap is signified. This tool makes extensive use of the Statistics, GeometricRelations, and Shape modules.

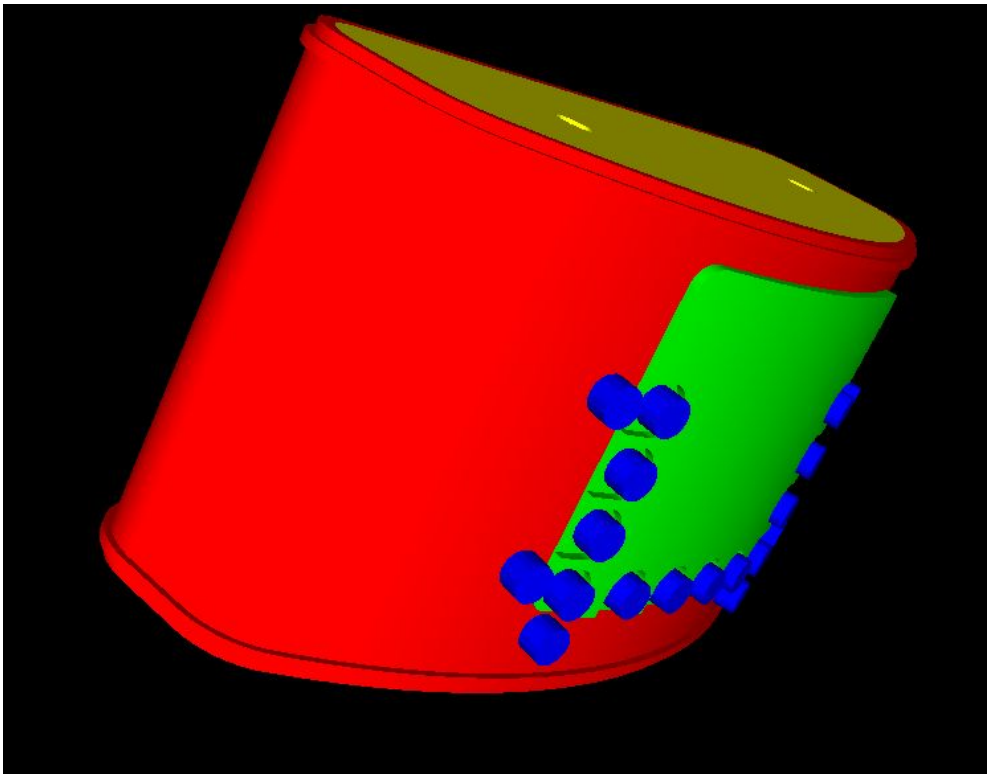


Figure 2: An item before deformation. Note that the green plate is flush with the read container.

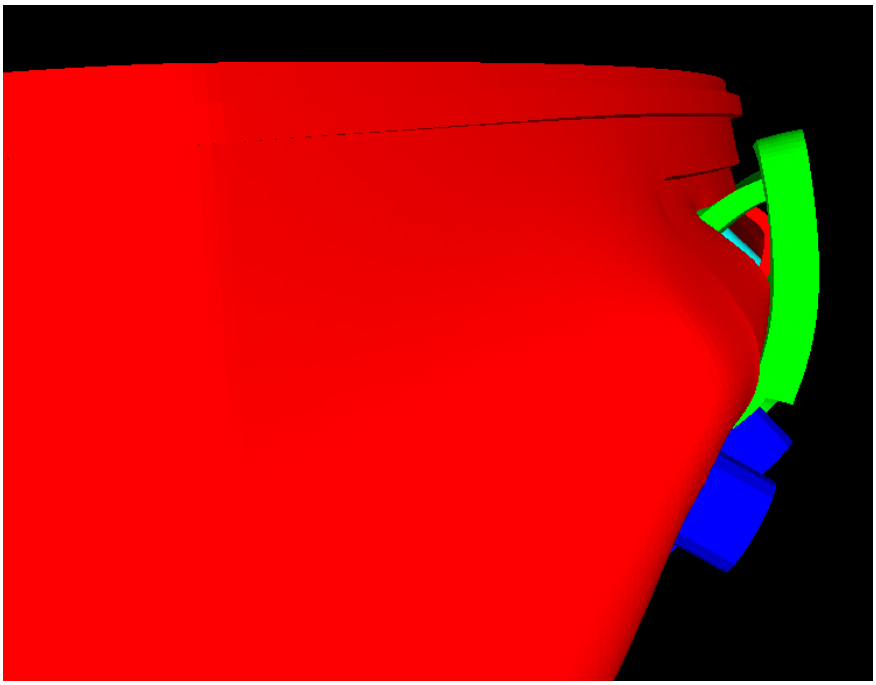


Figure 3: The item in the previous figure after deformation. The damage results a gap between the red container and its green outer plate which were initially in contact. The internals of the red container are visible through the resulting gap.

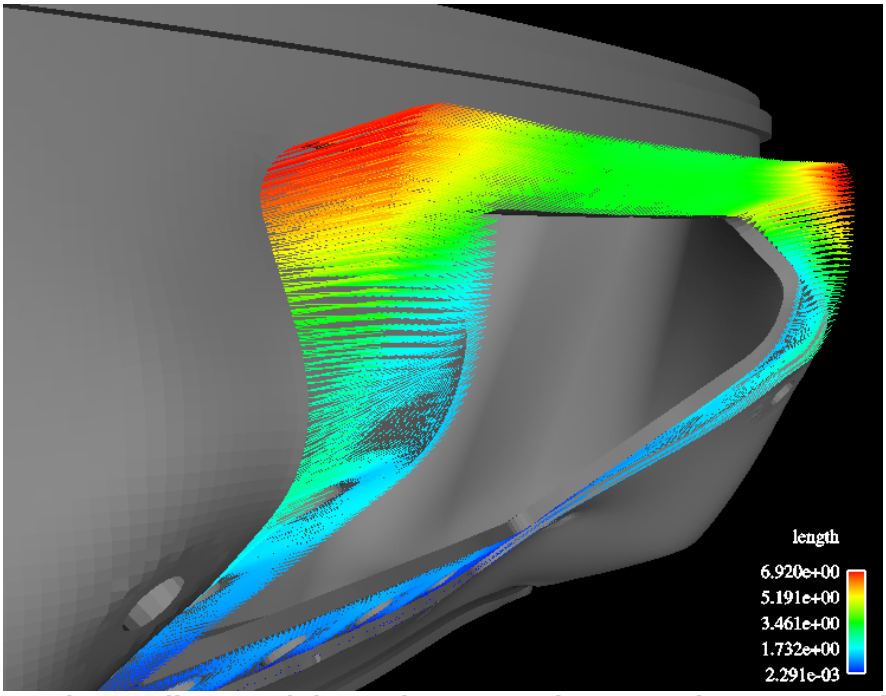


Figure 4: The Gaplines tool determines gaps that occur between meshes as a result of deformation. Gaplines are shown that result from the situation in the previous figure.

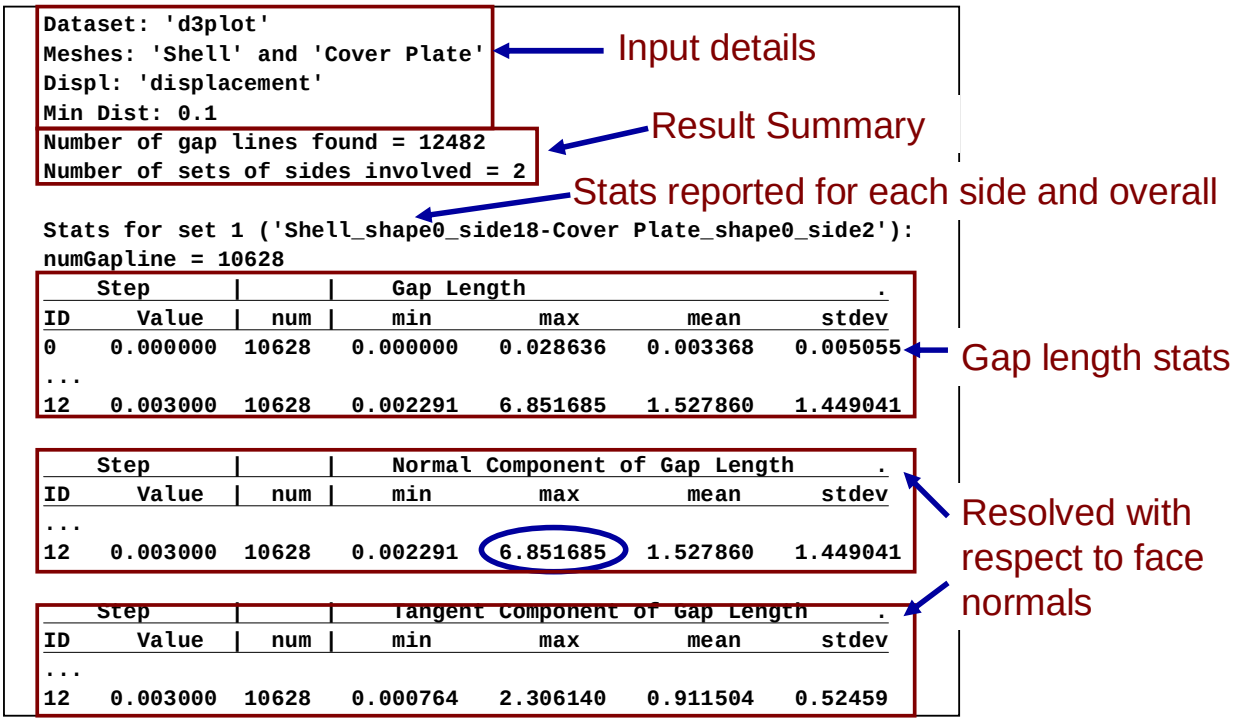


Figure 5: Partial output of the Gaplines tool for the situation shown in the figures. Characteristic information for each gap is provided, including size and location information.

5.2 Tears

The Tears tool is used for characterizing tears, which are defined as volumes of dead elements. In order to accommodate uncertainties in dead elements regions and tears that cross meshes, this tool will optionally combine tears within a given proximity to one another.

Characterizations of tears include determination of the number of dead elements in a tear, the volume of the tear, and a characteristic tear length, defined as the largest distance between any two vertices that define the surface of the dead region.

In addition, for simple shapes, characterization of the types, subtypes, and classes of tear are given, defined as:

Tear types:

- BREAK– breaks the shape into more than one pieces
- TUNNEL – intersects the shape in more than one place
- PIT – intersects the shape in a single place.

Subtypes:

- SINGLESIDE– intersects only a single side, but may be multiple time
- NONADJSIDE – intersects at least two non-adjacent sides
- ADJSIDES – intersects only adjacent sides (but may be in multiple places);

Class:

- MAJOR – intersects at least one major side fulfilling the thin shape assumption
- MINOR – intersects no major sides

These capabilities utilize the Statistics, Threshold, BoundingBox, DeadElement, and Shape capabilities of FCLib.

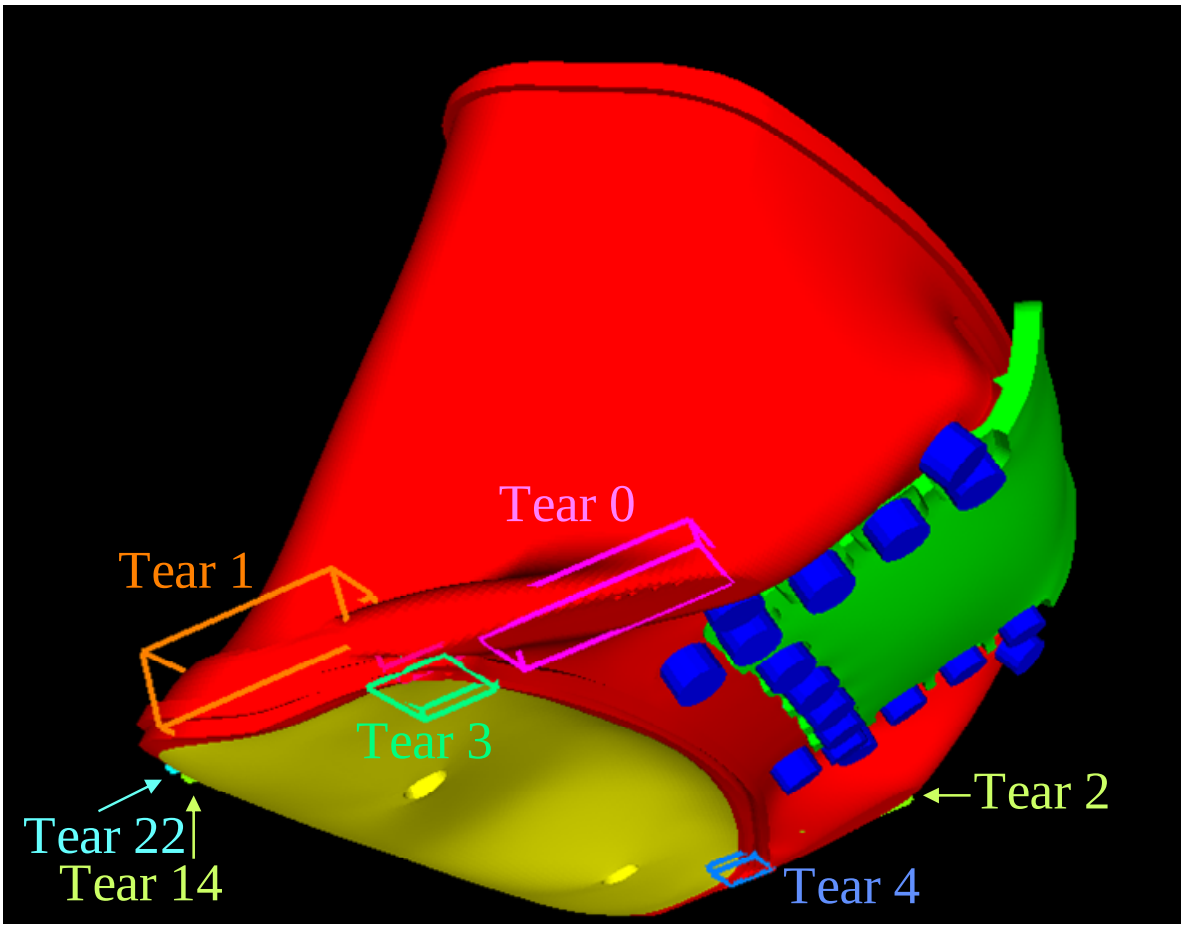


Figure 6: Tears resulting from the situation described regarding the gaps tool. The Tears tool discovers and characterizes tears, including determining bounding boxes for the tears (shown in figure).

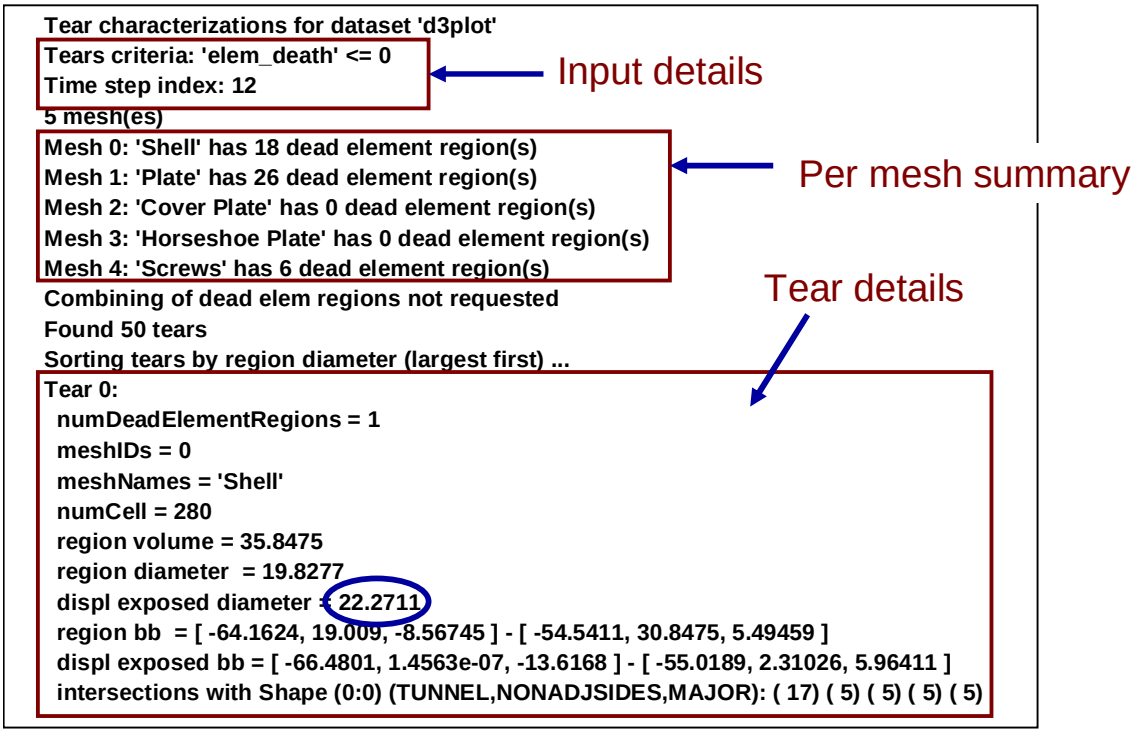


Figure 7: Partial output of the Tears tool for the situation shown in the previous figure. Characteristic information for each tear is provided, including size and location information.

5.3 ScrewBreaks

The ScrewBreaks tool was written for a specific application where the meshes included screws which held other meshes together. It is of interest to determine when a screw broke or how close a screw was to breaking. This calculation involves using the segmenting capabilities in the Threshold and Dead Element modules to determine if a dead element region increases the segments in the screw. Additionally the shape related functions are used to determine if the dead element region results in a side erosion of a screw which would also constitute breakage, through loss of contact of the remaining screw material with its neighboring meshes, though it does not result in a greater segmentation of the mesh. Finally, a closeness to breaking estimate is calculated by comparing the resultant surface area obtained by projecting the dead element regions onto the base of the screw to the absolute surface area of the base of the screw. While this is not particularly rigorous, it does roughly reflect how close the dead element region is to cutting through the screw. The screw base is determined by functions available in the Shape module. Bounding box capabilities are used in the printout in order to provide information to allow the user to distinguish the screw.

An example figure and selected output are below. The data set used is the “gen_screws.ex2” dataset in the data directory of the FCLib release. In this case the dataset consists of screws only, one in the first mesh, and two in the second. The screw in the first mesh breaks, in the second the screws erode. The breakage characterizations are identified in the output, along with the Breakage Ratio (BR) at each step and the bounding boxes for each screw. The first figure in this subsection shows the state of screws at Step 5, while the second figure shows the final state.

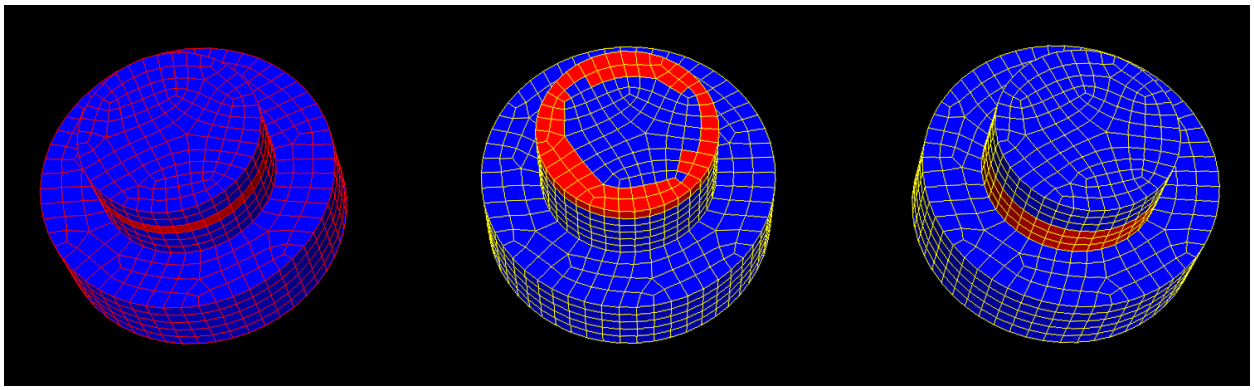


Figure 8: Timestep 5: The screws are partially damaged. The Breakage Ratio (BR) is calculated by projecting the damaged areas onto the screw base (uppermost in picture)

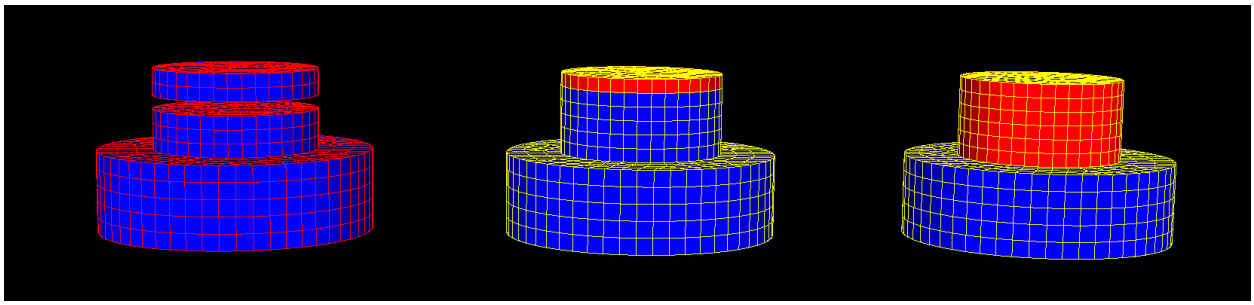


Figure 9: Final State: All screws are broken. The leftmost screw is severed. The middle and right screws are broken by surface erosion. While erosion does not result in a complete segmentation of the screws, nonetheless, the erosion results in a loss of contact of the remaining screw material with any of its neighboring meshes.

```
Screw characterizations for dataset '../data/gen_screws.ex2'
Mesh 0: 'screw-tear' has 1 screws
Screw bounding boxes at Step 0:
  Screw 0: [ -10, -9, -9.98308 ] - [ 10, 3, 9.98308 ]
Mesh screw-tear        Screw 0   Step 0   BR =  0.00 (   0.00/ 112.58)
Mesh screw-tear        Screw 0   Step 1   BR =  0.19 (  21.49/ 112.58)
…
Mesh screw-tear        Screw 0   Step 7   BR =  0.87 (  98.19/ 112.58)
Mesh screw-tear        Screw 0   Step 8   First broken
Mesh screw-tear        Broken/Total screws: 1/1
Mesh 1: 'screws-erode' has 2 screws
Screw bounding boxes at Step 0:
  Screw 0: [ 20, -9, -9.98308 ] - [ 40, 3, 9.98308 ]
  Screw 1: [ 50, -9, -9.98308 ] - [ 70, 3, 9.98308 ]
Mesh screws-erode      Screw 0   Step 0   BR =  0.00 (   0.00/ 112.58)
…
Mesh screws-erode      Screw 0   Step 5   BR =  0.67 (  75.71/ 112.58)
Mesh screws-erode      Screw 1   Step 5   BR =  0.54 (  60.53/ 112.58)
Mesh screws-erode      Screw 0   Step 6   BR =  0.76 (  85.74/ 112.58)
Mesh screws-erode      Screw 1   Step 6   BR =  0.54 (  60.53/ 112.58)
Mesh screws-erode      Screw 0   Step 7   BR =  0.86 (  96.88/ 112.58)
Mesh screws-erode      Screw 1   Step 7   First broken, side eroded
Mesh screws-erode      Screw 0   Step 8   First broken, side eroded
Mesh screws-erode      Screw 1   Step 8   *** still broken ***
Mesh screws-erode      Broken/Total screws: 2/2
All 2 Mesh(es)         Broken/Total screws: 3/3
```

Figure 10: Partial output of the ScrewBreaks tool for the situation shown in the figures. Characteristic information for each screw is provided, including breakage ratio (BR) and break type.

5.4 Feature Tracking

FClib was used to track and analyze the features corresponding to the crumpled regions of a can being crushed. The Feature Graph shows how the features interacted over time. A selected statistic (maximum stress) per feature over time is plotted. The big yellow feature at the top of the can is formed first and obtained the highest maximum stress. Other features are color-coordinated similarly. From this plot it can be seen that the maximum stress for a given feature begins to level off commensurate with the formation of a new feature.

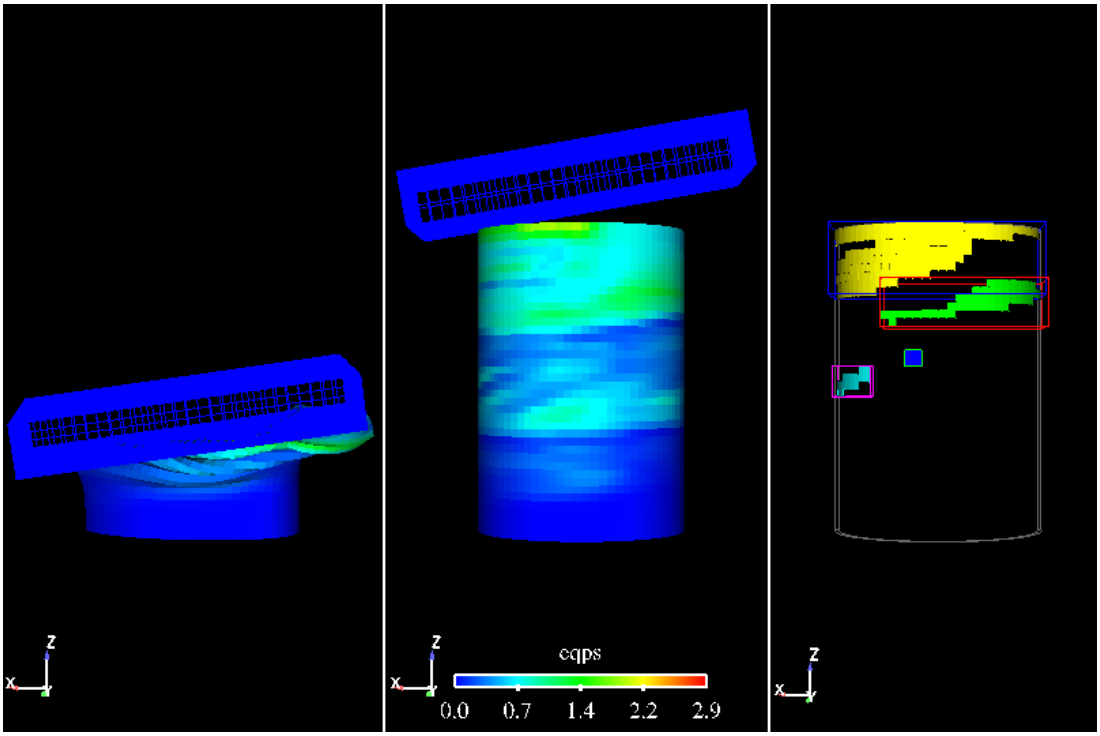


Figure 11: In the can crush example, features are located and colored in the right-most picture.

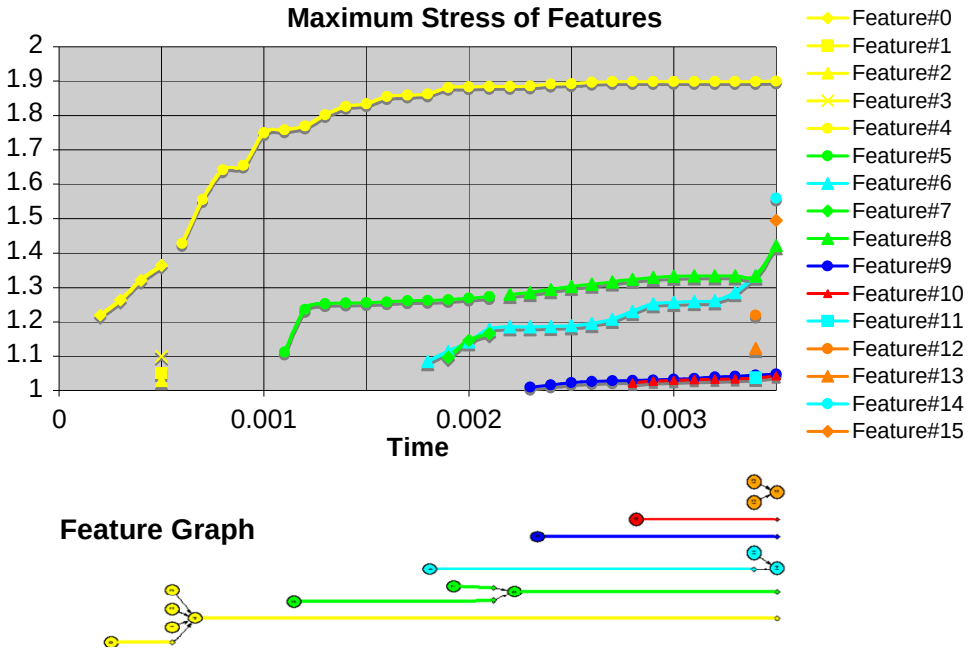


Figure 12: Maximum stress per feature over time for the can crush example. A feature graph displays the progression of different features in the dataset as time progresses. Feature colors correspond to those illustrated in the crushed can picture.

5.5 Skeleton Extraction and Manipulation

Many scientific FEM datasets employ meshes that are incredibly detailed. Given the sophistication of these meshes, it can be challenging for an analyst to be able to quickly analyze and understand the results of a simulation due to the high level of detail contained in the model. Additionally, meshing can make it challenging to compare one simulation run to another when the same object is meshed differently. Therefore it is beneficial to be able to transform meshes into simpler representations that are better suited for comparisons.

The skeleton extraction utilities in FCLib provide a basic set of tools for transforming mesh structures into tree representations that can provide insight into the geometric changes to structures in a simulation. The tools start by building a spanning tree representation of a mesh. The full spanning tree by itself can be useful for comparing two identical meshes oriented differently in one or more datasets. For example, scaling, translation, and rotation information can be obtained by comparing the coordinates of the root node in the tree and its children.

While a spanning tree representation simplifies a mesh, it is often desirable to reduce the tree to a more minimal form. The whittle tool in the skeleton extraction utilities provides multiple algorithms for reducing tree structures. These algorithms provide tradeoffs between tree quality, granularity of reduction, and the amount of time required to process data. The current algorithms include the following.

- **Minimum Descendents**: The minimum descendents algorithm removes the node from the graph that has the smallest number of descendents. While this algorithm is relatively fast and produces a tree with an exact number of nodes, it favors long branches and nodes close to the root of the tree.
- **Minimum Segment Change**: The minimum segment change examines all segments of all branches in the tree and removes the node that would cause the least error in the tree's distance representation (segments that connect to the node are rerouted to connect the node's parent and children). This approach is time consuming but simplifies detailed regions well.
- **Minimum Angle Change**: This algorithm examines all segments of all branches in the tree and removes the node that bridges segments that are the most aligned (i.e., the average angle the node is a part of is closest to 180°). The intention of this algorithm is to remove nodes that have the least impact on the shape of the tree.
- **Octal**: The octal algorithm attempts to remove nodes in a way that preserves spatial representations. Starting at the root node in the graph, the algorithm selects up to N nodes to keep in each of the eight Cartesian directions from the node (e.g., in the +X,+Y,+Z direction, +X,+Y,-Z direction, … -X,-Y,-Z direction). While this approach may preserve branches that disappear in the other algorithms, it does not provide the user with any granularity in the number of nodes in the final graph.

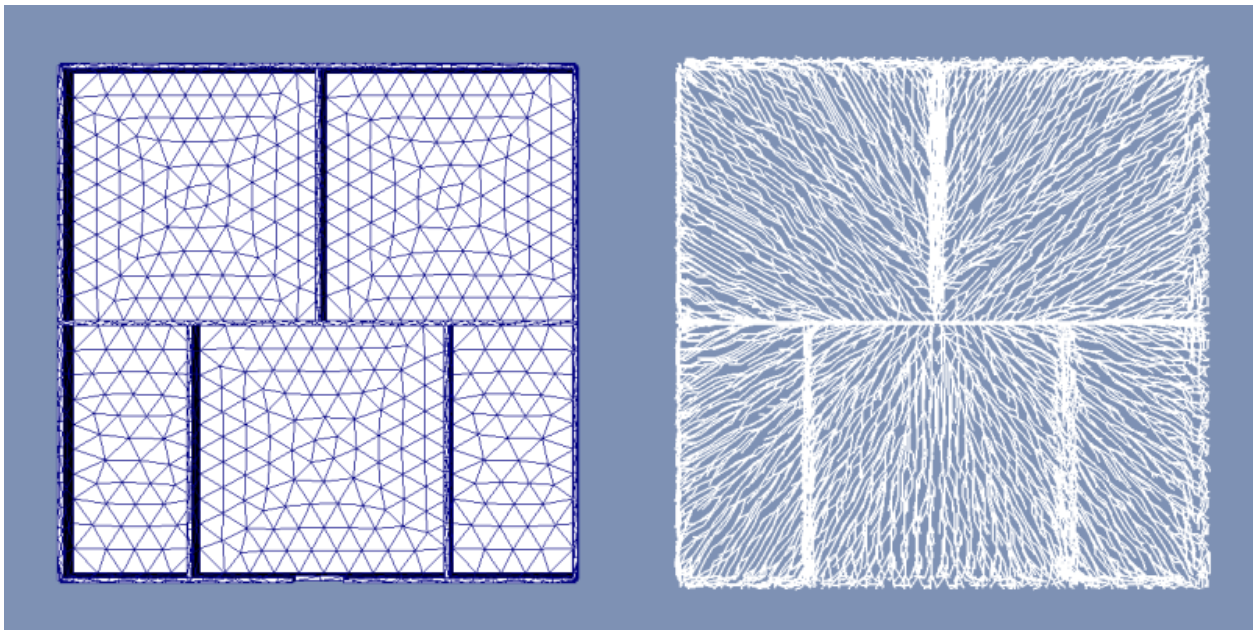


Figure 13: The skeleton utilities transform a mesh into a spanning tree structure.

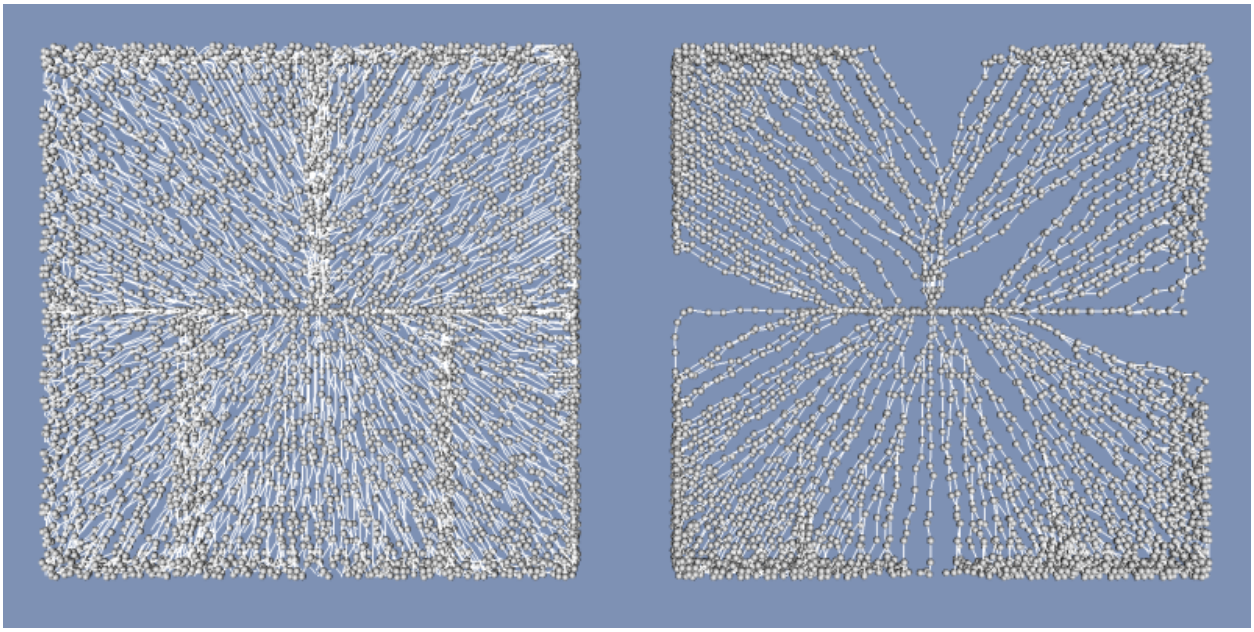


Figure 14: Spanning tree structures can be reduced to simplify the representation into a form that is easier to manage.

5.6 Region Subsetter/Reassembler

Crucial scientific information from simulations is lost in getting high-fidelity data to the post-processing analysis. Currently, analysis is done in a post processing fashion, using data files written out at frequencies determined by checkpointing considerations. However, such frequencies are inadequate to enable high fidelity analysis. In response, we have begun initial investigations into possibilities for providing higher fidelity analysis through in situ processing of the data (locating the analysis within the application). We anticipate that the resultant impact on application runtime can be mitigated by decreased size and frequency of I/O by outputting only the regions of interest in the simulation. The intent of this work then is to explore the impact, in both analysis accuracy and application runtime, of bringing the analysis to the data, through in situ concurrent processing.

Our initial scoping of the problem involved creating the capability to dump out on a per timestep basis only the regions of interest, and the necessary information to reconstitute the regions of interest in the context of the entire problem. Initial investigations show that the output of regions of interest from an actual ALEGRA simulation result in a substantial reduction in file size.

Capabilities here required the development of the RegionSubsetter tool, which as an example application, writes out only regions satisfying a prescribed threshold. The writeout involves dumping the segmented regions, and the variable data on those regions, out to a file as individual meshes (which we call “subset meshes” since they are new meshes, born of a subset on the original mesh). In addition, the subset meshes would have an additional new variable which consists of the vertex and/or element id mappings between the original mesh and the newly formed subset mesh. This mapping, as well as a well-known naming convention for the subset meshes and the relevant timestep would be used for reassembly of the subset meshes later. This is done via a companion code, Reassembler, which given an original mesh geometry, reconstitutes the subset meshes onto the original geometry for viewing in tools such as Ensight.

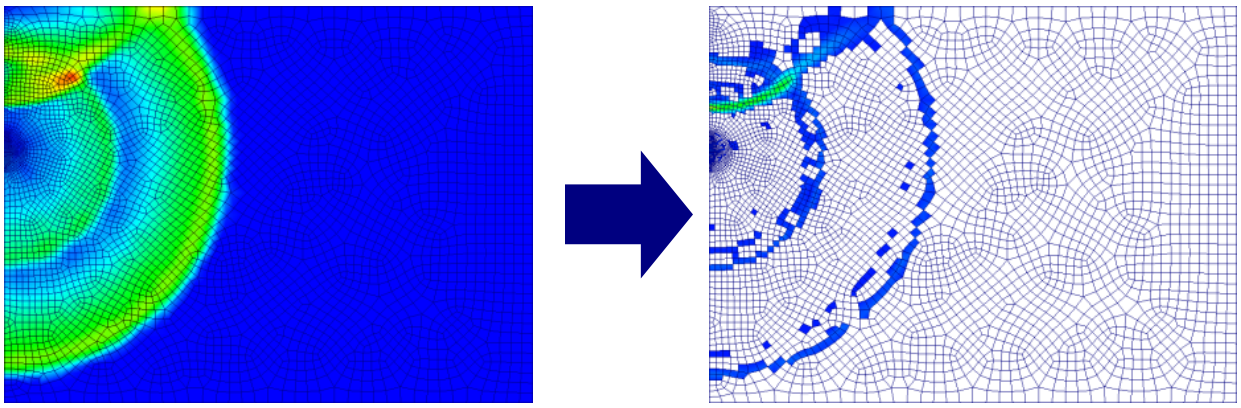


Figure 15: In this subsetter example, a large shockwave dataset is reduced to a minimal form that contains only elements that are significant to an analysis.

6. SUMMARY AND FUTURE WORK

FCLib is a powerful library that enables analysts to rapidly prototype data analysis operations. In addition to serving as a neutral interface into multiple file formats, FCLib is organized in a logical manner that allows users to interrogate data in a structured manner. The example applications demonstrate that FCLib can be used to develop command line tools that perform significant data analysis and characterization operations.

Currently, the access mode for FCLib capabilities is via command line interface. We have explored both a GUI interface and an XML interface, with specific emphasis on processing some of the well-defined and more commonly-desired characterizations, such as thresholding and simple mathematical processing.

Based on our experience with implementing FCLib, we see multiple areas where data analysis tools will need to be improved in the near future. The largest obstacle is balancing analysis performance with data set size. The transition to petascale-class science will result in datasets that are an order of magnitude larger or more than today's. Observing that processor performance is greatly outpacing disk performance, it is clear that tomorrow's data analysis applications will need to focus on efficient means of managing data in an out-of-core manner. While FCLib employs lazy loading to minimize disk access, future analysis tools will require more sophisticated data management facilities that either perform on-demand paging or employ parallel architectures for overcoming disk access overhead. These systems will likely require improvements to the programming environment in order to make them accessible for practical usage.

7. DISTRIBUTION

1	MS-9159	Philip Kegelmeyer	08962 (electronic copy)
1	MS-9152	Ann Gentile	08963 (electronic copy)
1	MS-9152	Craig Ulmer	08963 (electronic copy)
1	MS-0899	Technical Library	08944 (electronic)


Sandia National Laboratories