MESQUITE
Mesh Quality Improvement Toolkit

User's Guide

Patrick Knupp
The Sandia National Laboratories
Albuquerque NM USA

Lori Freitag-Diachin
Livermore National Laboratory
Livermore CA USA

Boyd Tidwell
Elemental Technologies Inc.
American Fork, UT USA

Last Updated: 01 May, 2012

Contents

Chapter 1

Introduction to Mesquite

1.1 Overview of Mesh Quality

Mesh quality refers to geometric properties of a mesh such as local volume, smoothness, shape, and orientation that, if not properly controlled, can adversely affect solution accuracy or computational efficiency of numerical simulations. In this section we give an overview of the role of mesh quality in the context of computer simulations of physical phenomena.

Simulation of many phenomena in the physical world involves computing numerical solutions to partial differential equations (PDE's). Commonly used approaches to computing numerical solutions such as finite volume and finite element methods require the use of approximations to the continuum operators in the PDE and a mesh or grid to subdivide the physical domain into small subregions. Together, the approximations and the mesh define a discretization. The difference between the exact solution to the PDE and the numerical solution is known as the discretization error. A *convergent* discretization means that the discretization error will asymptotically approach zero as the characteristic mesh size "h" approaches zero. Decreasing mesh size to reduce discretization error to nearly zero is often impractical in realistic simulations due to limited computing resources. One way to increase the accuracy of simulations with the same computer resources is to *adapt* the mesh to the domain and to the numerical solution. In adaptive refinement, the local mesh volume (or size) is made smaller in locations where the local discretization error is large and is made larger in locations where the error is small. In local h-refinement, mesh volume is made smaller by locally subdividing the mesh. In r-refinement, mesh volume is made smaller by moving mesh nodes closer together. Geometric adaptation can also be important in improving simulation accuracy. In regions of high domain curvature one adapts the mesh to the domain geometry by creating locally smaller mesh sizes. We see, then, that local mesh size (or volume) is a critical parameter in determining the accuracy of a simulation.

Aside from local mesh size, several other geometric mesh properties can affect solution accuracy. These include mesh smoothness, local mesh angles, aspect ratio, and orientation. For example, in some discretization methods there will be a loss of accuracy if the mesh is not smooth. In other cases, aspect ratios and orientation must be carefully adapted to the solution in order to maintain a certain level of accuracy. Simulations using meshes or domains that evolve in time (such as in ALE simulations) usually require that initially good geometric mesh properties be retained throughout the simulation time period. It is thus often important to control other geometric mesh properties in addition to local mesh size within an adaptive simulation.

In addition to solution accuracy, geometric mesh properties can also affect the amount of computer time required to obtain the numerical solution. Simulation codes usually employ iterative solvers to solve systems of equations and thus obtain numerical solutions to PDE's. The rate at which these solvers converge is determined by the spectral radius of a certain matrix. The spectral radius of the matrix is affected by, among other things, geometric properties of the mesh. Poor mesh quality can thus adversely impact solution efficiency.

Adaptive meshing techniques require an initial mesh to begin the adaptation procedure. Poor quality of the initial mesh (relative to the adapted mesh) can be difficult to overcome or, at least, reduce the efficiency of the adaptive procedure. For example, if the initial mesh contains locally inverted elements, these can often be fixed before the adaptive procedure begins. As another example, if it is known à priori that small angles will be needed on the boundary of the domain to obtain reasonable simulation accuracy, one should try to first create the small angles in the initial mesh to improve the efficiency of the subsequent adaptive meshing procedure.

Many simulations, particularly those in industry, are performed in a non-adaptive setting. That is to say, an initial mesh is generated and used throughout the calculation. The mesh is not changed as the solution is computed. Mesh quality remains important for such calculations. First, for complicated geometric domains it is often difficult to obtain good initial mesh quality. This is particularly true for non-simplicial meshes but can be true for simplicial meshes as well. A common requirement is that the mesh be smooth. Many simulation codes will not run to completion if the initial mesh contains a local volume which is negative. These must be eliminated before a simulation can begin. Analysts performing non-adaptive calculations often have considerable experience in using a variety of meshes on their problem and have a good à priori idea of what constitutes good mesh quality for a given problem. They thus desire to control the usual geometric mesh properties of the non-adapted mesh carefully.

1.2 How Mesh Quality Is Improved

Mesh quality can and should be considered during many stages of the mesh generation process from defeaturing CAD models to creation and adaptation of the mesh. Thus, for example, certain non-essential features of a CAD model, if eliminated, would go a long way to improving the quality of the mesh, depending upon the meshing scheme. Other critical meshing parameters which can affect mesh quality include geometric domain partitions, interval size and count, interaction of meshes within large assemblies of parts, biasing requirements, corner picking, etc. Choices made during the mesh generation phase of an analysis may have a large impact on initial mesh quality. Mesh quality can thus be improved by changing the way in which the domain is meshed.

Once the meshing stage is completed, one can improve mesh quality by techniques such as vertex movement and local topology modification. In vertex movement schemes, one seeks to reposition existing mesh vertices to achieve better quality. If vertex movement is undertaken within an adaptive setting, it is commonly referred to as r-refinement. Classic examples of vertex movement methods include Laplace smoothing [10] and Winslow smoothing [23]. It is helpful, in vertex movement schemes, to first be able to measure mesh quality so that one can explain in what sense one has improved it. Given a *metric* to measure mesh quality, one can formulate a numerical optimization problem which guides vertex movement to find the optimal mesh and thus improve its quality. Numerical optimization methods recently developed for unstructured meshes include [6, 13, 9, 8, 11, 12, 4].

A large number of mesh quality metrics have been devised to measure mesh quality. Many of these metrics are independent of any solution properties and are thus not useful in adaptive meshing. However, there are a number of weighted quality metrics which can be tied to the numerical solution via error indicators or other information for adaptive meshing.

Another way to improve mesh quality is to use local topological modification methods in which mesh vertices or elements are locally created and/or destroyed. These methods are very successful when applied to simplicial meshes, often within an adaptive context. Local topology modification is less effective on non-simplicial meshes.

Mesh quality improvement remains an important on-going research area. There remain, for example, open questions with regard to metrics which can be used in adaptive settings, theoretical questions on problem formulation, and how to obtain improved meshes quickly. An important subset of Mesquite capabilities is based on a mathematical theory that we are developing which we call the Target-matrix

paradigm (TMP). The basic idea is similar to that from Harmonic mappings, as applied to mesh generation: use only a few very soundly formulated quality metrics and adapt the mesh to a wide variety of specialized purposes via specification of the mapping on the target manifold. However, TMP is formulated as a discrete optimization problem, which allows direct control over important properties such as invertibility which must hold even if the asympototic limit is not reached. The mathematics behind the Target-matrix paradigm can be found in [14, 21, 5, 15, 17, 16, 18].

Although mesh quality improvement algorithms have been widely implemented in both meshing and applications codes, it has always been difficult to improve the quality of a mesh created in one software package using an improvement algorithm which has been implemented in another. This difficulty and others have inspired the creation of the Mesquite software library. This library is described in the next section.

1.3 Mesquite Goals

Mesquite (Mesh Quality Improvement Toolkit) is designed to provide a stand-alone, portable, comprehensive suite of mesh quality improvement algorithms and components that can be used to construct custom quality improvement algorithms. The design is flexible so that the algorithms can be applied to many different mesh element types and orders and referenced to both isotropic and anisotropic ideal elements. Mesquite provides a robust and effective mesh improvement toolkit that allows both meshing researchers application scientists to benefit from the latest developments in mesh quality control and improvement.

Mesquite design goals are derived from a mathematical framework and are focused on providing a versatile, comprehensive, inter-operable, robust, and efficient library of mesh quality improvement algorithms that can be used by the non-expert and extended and customized by experts. In this section we highlight the current status of Mesquite in several of our design goal areas.

Versatile. Mesquite works on structured, unstructured, and hybrid meshes in both two and three dimensions. The design permits improvements to meshes composed of triangular, tetrahedral, quadrilateral, hexahedral, prismatic (wedge) and pyramidal elements. Support for general polyhedral elements may be added at a future time. It currently incorporates only methods for node movement; plans for topology modification and hybrid improvement strategies lie in the future. Node movement strategies include both local patch-based iteration schemes for one or a few free vertices and global objective functions which improve all vertices simultaneously. Mesquite will be applicable to both adaptive and nonadaptive meshing and to both low- and high-order discretization schemes, but currently works with non-adaptive meshes containing linear elements.

Comprehensive. Mesquite will address a large variety of mesh quality improvement goals including mesh volume control (sizing, invertibility), mesh angles, aspect ratios, and orientation. Specific goals include mesh untangling, mesh smoothing, shape improvement, anisotropic smoothing, mesh rezoning for ALE, mesh alignment, and deforming mesh algorithms. These goals can be pursued in both adaptive and non-adaptive settings. The software is customizable, enabling users to insert their own quality metrics, objective functions, and algorithms and also provides mechanisms for creating combined approaches that use one or more improvement algorithms.

Inter-operable. To ensure that Mesquite is inter-operable with a large number of mesh generation packages, Mesquite defines a generic interface for accessing application mesh and domain data. Additoinally, Mesquite provides an adapter to interface with the common interfaces for mesh and geometry query currently under developed by the ITAPS center. These interfaces provide uniform access to mesh geometry and topology and will be implemented by all ITAPS center software including several DOE-supported mesh generation packages. We are working with the ITAPS interface design team to ensure that Mesquite has efficient access to mesh and geometry information through strategies such as information caching and agglomeration. We are also participating in the design of interfaces needed to support

topological changes generated by mesh swapping and flipping algorithms and to constrain vertices to the surface of a geometrical model.

Efficient. The outer layers of Mesquite use object-oriented design in C++ while the inner kernels use optimizable coding constructs such as arrays and inlined functions. To ensure efficient use of computationally intensive optimization algorithms, we employ inexpensive smoothers, such as Laplacian smoothing, as "preconditioners" for the more expensive optimization techniques. In addition, mesh culling algorithms can be used to smooth only those areas of the mesh that require improvement. Considerable attention has been devoted to understanding and implementing a variety of termination criteria that can be used to control the computational cost of the optimization algorithms.

Robust. Sound software engineering principles and robust numerical algorithms are employed in Mesquite. A comprehensive suite of test problems and a unit testing framework have been developed to verify the correct execution of the code.

Mesquite is not intended to be a mesh generation tool. It can serve as a post-processor to a mesh generation procedure, a mesh pre-processor to a non-adaptive simulation code, or as an algorithm for in-core adaptive mesh quality improvement. As a software library, Mesquite is intended to be linked to either a meshing code or to a simulation code.

1.4 Mesquite Concepts

Mesquite software design is based on a mathematical framework that improves mesh quality by solving an optimization problem to guide the movement of mesh vertices. The user inputs a mesh or submesh consisting of vertices, elements, and the relationships between them. The quality of each vertex or element in the mesh is described by a local quality metric that is a function of a subset of the mesh vertices. The global quality of the mesh is formed by taking the global norm or the average of the local mesh qualities. The global quality is thus a function of the positions of all the mesh vertices. If this function can be used in a well-posed minimization problem (e.g., it is bounded below and has one or more local minimums), mesh vertices are moved by Mesquite toward the vertex positions of the optimal mesh, thus improving the quality according to the criterion defined by the local quality metric. By changing the local quality metric one can achieve a variety of mesh quality improvement goals such as mesh untangling, shape improvement, and size adaptation.

Users of Mesquite should have in mind a goal or set of goals which define the quality of the mesh which is to be improved. The goal determines which quality metric or metrics one will use in the optimization problem. Other user inputs will include an objective function template which describes the norm or average they wish to use in defining the global mesh quality. For example, an L-infinity norm will tend to improve the worst-case local quality while an L-2 norm will improve the RMS quality of the global mesh. Once the global quality (objective function) is defined, the user can select a numerical optimization scheme (solver) within Mesquite such as a steepest descent, conjugate gradient, or feasible Newton method. A variety of termination criteria can be selected singly or in combination to tell the solver when to halt. These are useful in controlling the trade-off between the accuracy of the minimization procedure vs. how much CPU is consumed. There is also an important flag that determines whether the optimization problem will be solved via a succession of optimizations on local patches followed by a complete pass over the global mesh or if it will be solved using a global patch in which all mesh vertices are moved simultaneously. Advantages and disadvantages of each of these approaches is currently under study.

Sometimes hybrid mesh optimization schemes are useful, for example, in first untangling a mesh and then improving the shape of its elements. For sequences of optimization problems Mesquite uses the concept of an instruction queue. The queue determines the order in which the optimization problems are solved, using the output from the previous optimization step as the input to the next optimization step. The queue defines a master quality improver that defines the ultimate mesh quality improvement goal.

The queue can also be used to include steps to assess mesh quality say before and after each optimization step within the queue. The quality assessor measures various aspects of quality in the mesh and may include other quality metrics besides the one used to define the optimization problem.

Optimization problems can be solved directly by minimizing the objective function or indirectly by positioning mesh vertices at a stationary point of the global objective function. Stationary points are defined by setting the gradient of the objective function to zero. The indirect method is akin to iteratively solving a system of linear (or nonlinear) equations. Currently, such systems are solved in Mesquite and other mesh quality software by using the local patch method that is akin to a Gauss-Seidel iteration. The prime example of this in Mesquite is Laplace smoothing. In the future we may include methods for solving global systems of equations in Mesquite to obtain solutions more quickly. In the past, some mesh smoothing algorithms have been formulated as a local iterative method that cannot be derived by setting the gradient of an objective function to zero. Such methods are frowned upon in Mesquite since one cannot state what mesh quality metric is improved. However, if such methods are included in future versions of Mesquite, they will be done in a manner similar to the local Laplace smoothing algorithm in Mesquite.

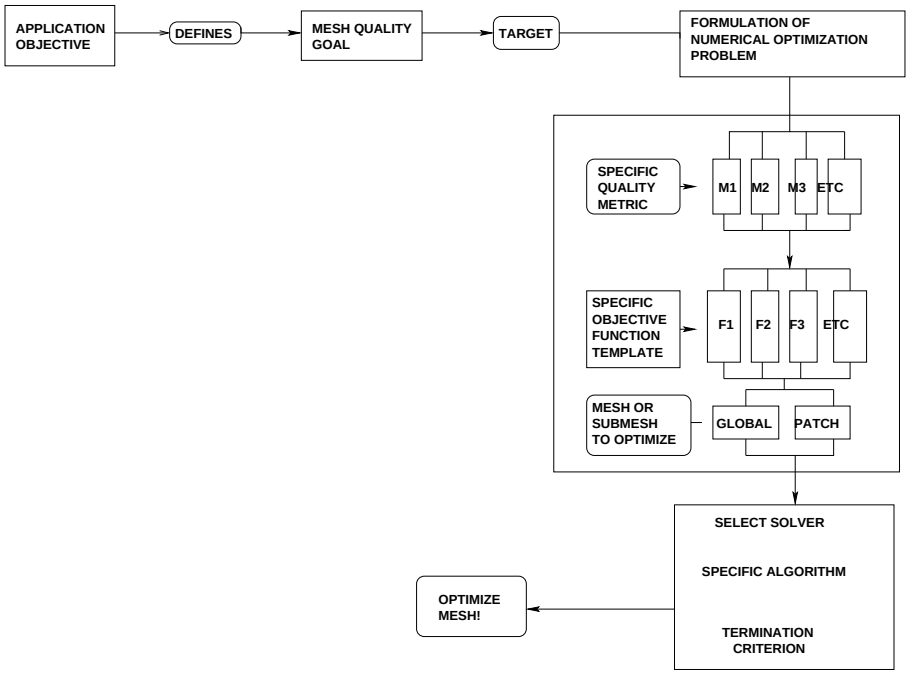

Figure 1.1: *The Mesquite Paradigm*

1.5 How to use this User's Manual

This user's manual

- provides an introduction to mesh quality and basic Mesquite concepts (Chapter 1),
- instructs novice users on how to download and install Mesquite (Chapter 2),
- provides a tutorial on Mesquite's simplified user's interface and Mesquite's detailed API (Chapter 3).
- describes how to load a mesh in Mesquite via files (Chapter 4), and

- describes Mesquite interactions with domain geometry (Chapter 5), and
- describes Mesquite Wrappers (Chapter 6),

Consult the doxygen documentation for the API reference as well as details on the software. There are two sets of doxygen documentations available:

- The developer doxygen doc is located in mesquite/doc/developer/. From that directory, you must run 'doxygen Mesquite.dox'.
- The user doxygen doc (API doc) is located in mesquite/doc/user/doxygen. From that directory, you must run 'doxygen Mesquite-user.dox'.

The doxygen command will generate two directories: an html directory containing the file index.html that you can open with your web browser, and a latex directory containing a Makefile that will generate a dvi file.

Chapter 2

Installing Mesquite

2.1 Requirements

2.1.1 Downloading Mesquite

The Mesquite distribution (in source form) may be obtained at the following URL:

`http://www.cs.sandia.gov/~web1400/1400_download.html`

2.1.2 Supported Platforms and Build Requirements

The Mesquite source code will compile in any environment comforming to the ISO/IEC 9899-1999 (C99), ISO/IEC 14882-1998 (C++98) and ISO/IEC 9945:2003 (POSIX) standards. It may also compile under many other environments.

Mesquite requires a reasonably standards-conforming C++ compiler and corresponding libraries. No additional libraries are required to build the core Mesquite library. Several optional features have additional requirements. These are listed in the next section.

Mesquite uses the GNU autotools build system. The Makefiles generated by the configure script should work on any Unix-like platform using the build tools (e.g. make) provided with that platform. The minimal requires beyond a C++ compiler are a Bourne shell (typically `/bin/sh`) implementation and a minimal corresponding command environment and an implementation of the `make` utility.

Support for building Mesquite with Microsoft Visual Studio is no longer available as of Mesquite version 2.0.

2.1.3 Optional Libraries and Utilities

- Unit tests: Mesquite provides a series of unit tests that may be used to verify the correct behavior of a build of the Mesquite library. These tests are implemented using CppUnit framework. The CppUnit framework must be installed to compile and run these test. It is available at this URL:

  `http://cppunit.sourceforge.net`

- ExodusII support: To enable support for reading or writing ExodusII files in Mesquite, the header files for the ExodusII library must be available. To link an application with a Mesquite library supporting ExodusII, the ExodusII library and possibly the NetCDF library must be available. To obtain the ExodusII library, contact:

  ```
  Marilyn K. Smith
  Research Programs Department
  Department 9103, MS 0833
  Sandia National Laboratories
  P.O.Box 5800
  ```

Albuquerque, NM 87185-0833
Phone: (505) 844-3082
FAX: (505) 844-8251
Email: mksmith@sandia.gov

The NetCDF library can be obained at the following URL:

http://www.unidata.ucar.edu/downloads/netcdf/index.jsp

2.2 Building Mesquite

After downloading and unpacking the Mesquite source, the next step is to configure and build and install the Mesquite library.

2.2.1 Compiling on Unix-like systems

This section presents the steps required to compile Mesquite with the default options. It is typically required that Mesquite be "installed" before it is used in an application. The default installation location is the system-wide `/usr/local` directory. It is more common to specify an alternate directory in which to install the Mesquite library and headers. This can be done using the `--prefix` option to the `configure` script. Additional options are available for fine-grained control of installation locations.

1. Change your working directory to the top-level Mesquite source directory (typically `mesquite-<version>/`).
2. Run the configure script with the command:

   `./configure --prefix=<installdir>`,

   replacing `<installedir>` with the location in which the finished Mesquite library is to be placed.
3. Compile Mesquite with the command: `make`
4. Optionally verify that Mesquite compiled correctly with the command: `make check`
5. Move resulting files into the destination (install) directory with the command: `make install`

If the configure step failed, please consult the following section describing some of the optional arguments to the `./configure` script.

2.2.2 Options for Unix-like systems

This section describes the options available for customizing the build system and the resulting Mesquite library. An brief description of these and other options is available with the command: `./configure --help`.

The following values may be specified as environmental variables, as arguments to the configure script using the NAME=VALUE syntax, or as arguments to `make` using the NAME=VALUE syntax. The value of these variables (if set) during the configure step will become the default for the compile step. The value of any of these variables will override the the default if specified during the compile step.

CXX The C++ compiler command

CXXFLAGS Arguments to the C++ compiler, such as those specifying debug symbols or the optimization level.

CC The C compiler command

CFLAGS Command line arguments to be used for the C compiler.

DOXYGEN The doxygen API documentation generation tool.

Most options to the configure script are either of the form

`--with-FEATURE[=ARG]` or `--enable-FEATURE[=ARG]`.

Some options may accept an additional argument following an '=' character. For each –with-FEATURE option, there is also a corresponding –without-FEATURE option. Similarly, there is a –disable-FEATURE option corresponding to each –enable-FEATURE option. The negative forms fo the options (–without-FEATURE and –disable-FEATURE) do not accept an additional argument. Only the positive form of each option is stated in the description below.

The following general build and debug options may be specified during the configure step:

–**enable-debug** Select a subset of the following options that make the most sense for developers of Mesquite.

–**enable-release** This is the default behavior unless –enable-debug is specified. It selects a subset of the following options that typically work best for using Mesquite in a production application.

–**enable-compile-optimized** Compile with the available optimizations that improve performance without any significant drawbacks (the -O2 compiler flag.)

–**enable-debug-symbols** Include debugging information in the compiled Mesquite objects (the -g compiler flag).

–**enable-debug-assertions** Include internal consistancy checks that abort when an error is detected.

–**enable-debug-output=n,m,...** Enable the output of debug and status messages to file descriptor 1 (stdout). An list of integer debug flags for which to enable output may be specified as a comma-separated list of values. The default is to enable debug flags 1 and 2 if this option is specified without any explict debug flag values.

–**enable-function-timers** Enable time-profiling of some portions of Mesquite.

–**enable-trap-fpe** Enable generation of a floating-point exception signal for arithmatic errors (e.g. division by zero.) This is an option intended for Mesquite developers. Enabling this will typically cause the application using Mesquite to abort when such an error is encountered.

–**enable-namespace** Specify an alternate namespace so as to avoid symbol conflicts between multiple versions of Mesquite. See Section 2.2.4.

The following options specify optional Mesquite components and the location of the corresponding dependencies.

–**with-cppunit=DIR** The CppUnit library is required to compile and run the tests to verify that a particular build of the Mesquite library is working correctly. If the CppUnit library is not installed in a default location where the ./configure script can find it, this option may be used to specify the location.

–**with-exodus=DIR** Enable support for reading and writing ExodusII files, and optionally specify the location where the ExodusII library and headers required for this option are installed.

–**with-netcdf=DIR** Specify the location of the NetCDF library required by the ExodusII library. The default is to look in the ExodusII directory.

2.2.3 Compiling on Microsoft Windows (CMake build)

The Mesquite source includes the necessary input files to generate Microsoft Visual Studio project files using the CMake utility. You will need to download and install the CMake utility for Windows if you have not already done so. It is available at:

`http://www.cmake.org/cmake/resources/software.html`

Using the graphical verison of the CMake utility, select the folder containing the Mesquite source and enter a folder in which you would like the CMake output and compiled code to be stored. Select the `Configure` button. You will be presented with a group of configuration options. Modify any desired options and click the `Configure` button again. Each time you change one or more configuration options, you must click the `Configure` button to update the list of available options. When you have finished changing build options, click the `Generate` button to generate Visual Studio input files and exit the CMake utility.

The build folder you specified in the CMake utility should now contain the necessary input files to build Mesquite using Microsoft Visual C++.

2.2.4 Linking Multiple Versions of Mesquite

Sometimes it is necessary to have multiple different versions of a library such as Mesquite linked into the same application. This situation typically arises when an application needs both Mesquite and some other library that depends on an older version of Mesquite. Without taking steps to avoid symbol name conflicts such a situation will often result in surprising, strange, and difficult to diagnose runtime errors.

Mesquite provides the ability to specify an alternate namespace and a standard namespace alias to assist with addressing such situations. The "namespace" `Mesquite` is typcially an alias to the true internal C++ namespace containing all Mesquite code. Applications can and should use that alias rather than the internal namespace to avoid the need to modify application code whenever the internal namespace changes.

The internal namespace can be changed with the configure option `--enable-namespace=MyNS` or the cmake option `Mesqutie_NAMESPACE`, where the value "MyNS" can be replaced with any string that is an acceptable C++ namespace label. The default namespace is `MesquiteN`, with the Mesquite major version subtitited for `N`. Specifying an alternate internal namespace results in different mangled symbol names in the compiled library, thus avoiding symbol name conflicts.

If the requested namespace is anything other than "Mesquite", then Mesquite will always provide the alias `namespace Mesquite = MESQUITE_NS;` so that application code may always use the `Mesquite` namespace.

Chapter 3

Examples

3.1 Short Tutorial

In this section, we write a driver code which calls the Mesquite library to improve the quality of a test mesh. This tutorial section is aimed at giving the user a feel for Mesquite: *this section is not where to look for detailed information.* In particular, information pertaining to loading a particular mesh format (see Chapter 4), interacting through a particular mesh interface (section 4.1), and details of defining geometric domains (see Chapter 5) are not given in this section.

First, we write a small program using Mesquite's simplified API, or wrappers, to show the fastest way to deploy Mesquite functionality to improve a mesh. The wrapper concept, as well as details about the different wrappers available, are described in section 3.1.3. Following this first example, we set up customized mesh improvement tool using Mesquite's low-level API, the details of which are described in section 3.1.4.

3.1.1 Tutorial File Template

To create and link a driver code, the Mesquite library must be installed per the instructions of section 2.2. The commands and file names specified in this section are relative to the installed `testsuite/tutorial` directory. It is assumed that that is the working diretory. This tutorial begins with the file `tutorial.cpp`, which contains the following template:

```
1.  #include "Mesquite_all_headers.hpp"
    #include <ostream>
2.  using namespace Mesquite;
    int main(int argc, char* argv[])
    {
3.    MsqError err;

      if (argc != 2) {
        std::cerr << "Expected mesh file names as single argument."
                  << std::endl;
        exit (EXIT_FAILURE);
      }

      // new code starts here
4.    //...

      return 0;
    }
```

The lines labeled 1-3 highlight three basic aspects of using Mesquite;

1. For convenience, Mesquite provides the header file

`include/Mesquite_all_headers.hpp`

which includes all Mesquite headers. Although this is the easiest way to handle the include directives, it may slow down compilation of the application.

2. All Mesquite classes are part of the `Mesquite` namespace.

3. The `MsqError` class defines an object type used to communicate Mesquite errors to the application. The calling application must pass an instance of the `MsqError` class or an instance of a subclass of `MsqError` to many Mesquite functions. The state of the error object may be checked by casting the instance ot a Boolean or using it in a Boolean context. The state is cleared by calling the `clear` method.

4. In the sections that follow, we guide the user through the steps necessary to smooth a mesh using Mesquite. All new lines of code to be added to the template file start in this position and are added in the order in which they are discussed.

The code above takes a mesh file name as a command line argument and performs no action. We can compile it in the (`examples/`) directory with the command:

```
make -f tutorial.make
```

3.1.2 Loading a Test Mesh

Our next step is to load one of the test meshes distributed with Mesquite. These meshes are distributed in the VTK unstructured mesh format, the details of which are given in [22, 3]. This format was chosen because of its readability and ease of use. In this tutorial we use the simplest mechanism for loadling a mesh into Mesquite; different options are described in Chapter 4. In particular, to load a VTK test mesh in Mesquite, instantiate the Mesquite mesh database object, `MeshImpl`, and use the `read_vtk` member function by adding the following lines to the file template described in 3.1.1.

```
Mesquite::MeshImpl my_mesh;
my_mesh.read_vtk(argv[1], err);
if (err)
{
  std::cout << err << std::endl;
  return 1;
}
```

If the mesh read in contains more than one type of element, Mesquite will automatically handle the mixed elements with no additional effort required.

Mesquite also provides a function to write a mesh file in VTK format, given a `MeshImpl` object:

```
my_mesh.write_vtk("original_mesh.vtk",err);
```

Mesquite deals automatically with all types of supported elements (triangles, quadrilaterals, tetrahedra, hexahedra, wedges, and pyramids), and also hybrid meshes consisting of mixed element types. Some meshes require geometry information as well. When improving a surface mesh, Mesquite must be provided information about surface(s) the mesh is constrained to lie on and the association between mesh entities and entities of the geometric domain (surfaces, curves, etc.) Because Mesquite is inherently a 3D code, all 2D meshes must specify some geometry constraints. The details for general geometric surfaces are explained in Chapter 5. In this section, we show how to define the geometry of a 2D planar mesh, specified by a point (x, y, z) and a normal. For example, the following defines an xy-plane shifted five units in the z-direction:

```
Vector3D normal(0,0,1);
Vector3D point(0,0,5);
PlanarDomain my_mesh_plane(normal, point);
```

3.1.3 Improving the Mesh with a Wrapper Class

The simplest way to use a Mesquite mesh quality improvement procedure is to instantiate one of the wrapper classes described in Chapter 6. Here, we will instantiate the `ShapeImprovement` wrapper and use it to improve the Mesh we created earlier. Mesquite can optimize the mesh without further input from the user by utilizing preset, default values. If some customization is desired, the wrapper classes also allow users to set the most important parameters of the underlying algorithms and metrics (see Chapter 6 for details).

```
Mesquite::ShapeImprover mesh_quality_algorithm;
mesh_quality_algorithm.run_instructions(&my_mesh,
                                        &my_mesh_plane,  err);
//Should check the error object after the instruction is ran
// to see whether the instructions were all successful.
if (err)
{
    std::cout << err << std::endl;
    return 1;
}
```

Once the algorithm has been executed using the `run_instructions` member function of the wrapper class, the improved mesh can be written to a new file:

```
my_mesh.write_vtk("smoothed_mesh.vtk",err);
```

This completes the code necessary for the simple wrapper example. Once the code has successfully compiled by typing the `make` command given in section 3.1.1, run it from the tutorial directory `mesquite/testSuite/tutorial/` with a mesh file name as a command line argument by typing

```
./tutorial ../../meshFiles/2D/VTK/square_quad_10_rand.vtk
```

The code creates the files original_mesh.vtk and improved_mesh.vtk in the current directory. These two meshes, the original and the optimized, are shown in figure 3.1. The text output of the code, shown below, reports the inverse mean ratio quality metric statistics for the mesh at three stages: the original mesh, the mesh at an intermediate step of the optimization, and the final mesh. The optimized mesh consists of square quadrilaterals which have an inverse mean ratio value of 1.0.

```
************** QualityAssessor Summary **************

  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

             metric    minimum    average        rms    maximum
 Inverse Mean Ratio    1.01013    1.16655     1.1738    1.79134

************** QualityAssessor Summary **************

  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

             metric    minimum    average        rms    maximum
 Inverse Mean Ratio    1.01013    1.16655     1.1738    1.79134

************** QualityAssessor Summary **************

  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

             metric    minimum    average        rms    maximum
 Inverse Mean Ratio          1          1          1          1
```

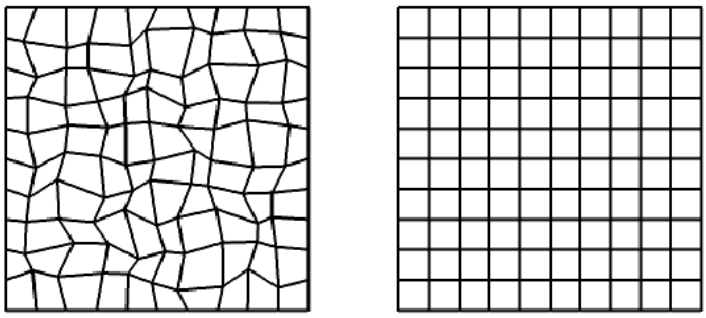

Figure 3.1: square_quad_10_rand.vtk mesh. The original mesh is on the left, the mesh smoothed with `ShapeImprover` is shown on the right.

3.1.4 Improving the Mesh with the Low Level API

If the user requires in-depth control over the mesh quality improvement process, the use of lower-level Mesquite classes provides an extensive amount of flexibility. In particular, the user can specify the quality metric, objective function template, and optimization algorithm by instantiating particular instances of each. For each, various options such as numerical or analytical gradient and Hessian evaluations or the patch size can be selected. Furthermore, the user can fine tune the optimization algorithm performance by creating and setting the parameters of the termination criteria.

Once these core objects have been created and customized, the user creates an instruction queue and adds one or more quality improvers and quality assessors to it. The mesh optimization process is initiated with the `run_instructions` method on the instruction queue class.

In this section, we provide a simple example to highlight the main steps needed for this approach. The code segment given below performs the same functionality as the wrapper class highlighted in the previous section. The comment lines provide high level documentation; the details of each class and the low-level API are not described here.

```
  // creates a mean ratio quality metric ...
IdealWeightInverseMeanRatio inverse_mean_ratio(err);

  // sets the objective function template
LPtoPTemplate obj_func(&inverse_mean_ratio, 2, err);

  // creates the optimization procedures
TrustRegion t_region(&obj_func);

  //performs optimization globally
t_region.use_global_patch();

  // creates a termination criterion and
  // add it to the optimization procedure
  // outer loop: default behavior: 1 iteration
  // inner loop: stop if gradient norm < eps
TerminationCriterion tc_inner;
tc_inner.add_absolute_gradient_L2_norm( 1e-4 );
t_region.set_inner_termination_criterion(&tc_inner);

  // creates a quality assessor
QualityAssessor m_ratio_qa(&inverse_mean_ratio);
  // creates an instruction queue
```

```
  InstructionQueue queue;
  queue.add_quality_assessor(&m_ratio_qa, err);
  queue.set_master_quality_improver(&t_region, err);
  queue.add_quality_assessor(&m_ratio_qa, err);

    // do optimization of the mesh_set
  queue.run_instructions(&my_mesh, &my_mesh_plane, err);
  if (err) {
    std::cout << err << std::endl;
    return 2;
  }
```

3.1.5 Mesh Improvement Examples

The left image in figure 3.2 shows a mesh that has been degraded by moving the disk from the right side of the square to the left while keeping the mesh topology fixed. The mesh file `mesquite/meshFiles/2D/VTK/hole_in_square.vtk` contains the information for this mesh. If you plan to run this example, note that the normal direction that defines the geometry is now $(0, 0, -1)$. This change must be made in the tutorial example code as was done in section 3.1.2, or an error message will be thrown.

```
  Vector3D normal(0,0,-1);
  Vector3D point(0,0,-5);
  PlanarDomain my_mesh_plane(normal, point);
```

We can now improve the mesh with the wrapper mentioned in 3.1.3 or the detailed API mentioned in 3.1.4. Because we changed the normal, the driver code must be recompiled; otherwise the code and executable are as before. Once the code is recompiled, type

```
./tutorial ../../meshFiles/2D/VTK/hole_in_square.vtk
```

to improve this mesh. The smoothed mesh is shown in the right image of figure 3.2. The vertex locations have been repositioned and significantly improve the quality of the mesh, as shown by the onscreen quality assessor output:

```
************** QualityAssessor(free only) Summary **************

  Evaluating quality for 140 elements.
  This mesh had 140 quadrilateral elements.
  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

            metric     minimum     average         rms     maximum    std.dev.
Inverse Mean Ratio     1.07588     85.8391     463.357     5037.46     455.336

************** QualityAssessor(free only) Summary **************

  Evaluating quality for 140 elements.
  This mesh had 140 quadrilateral elements.
  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

            metric     minimum     average         rms     maximum    std.dev.
Inverse Mean Ratio     1.01896     1.83479     1.91775     3.36336    0.557969
```

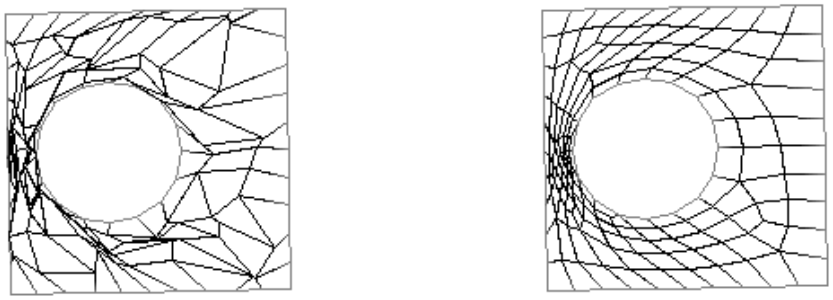

Figure 3.2: hole_in_square.vtk mesh. The original mesh is on the left, the mesh smoothed with Mesquite is shown on the right.

3.1.6 Regression Testing

Regression testing encompasses running unit tests as well as comparing results data against "blessed" or "gold" data. An example of comparing results of a smoothed mesh against a gold version is in `mesquite/testSuite/parallel_smooth_laplace/par_hex_smooth_laplace.cpp`. This utilizes a function in `MeshUtil`, `meshes_are_different`, to compare two `MeshImpl` objects (within a specified numerical tolerance). It is recommended that both unit testing and gold-comparison testing be included in your test code development.

Chapter 4

Getting Mesh Into Mesquite

The application must provide Mesquite with data on which to operate. The two fundamental classes of information Mesquite requires are:

- Mesh vertex coordinates and element connectivity, and
- Constraints on vertex movement.

In this chapter we will assume that the only constraint available for vertex movement is to flag the vertices as fixed. More advanced constraints such as vertices following geometric curves or surfaces are discussed in the following chapter.

The mesh data expected as input to Mesquite is a set of vertices and elements. Each vertex has associated with it a fixed flag, a "byte", and x, y, and z coordinate values. The fixed flag is used as input only. It indicates whether or not the corresponding vertex position should be fixed (i.e., coordinates not allowed to change) during the optimization. The "byte" is one byte of Mesquite-specific working data associated with each vertex (currently only used for vertex culling.) The coordinate values for each vertex serve as both input and output: as input they are the initial positions of the vertices and as output they are the optimized positions.

Each element of the input mesh has associated with it a type and a list of vertices. The type is one of the values defined in `Mesquite::EntityTopology` (`Mesquite.hpp`). It species the topology (type of shape) of the element. Currently supported element types are triangles, quadrilaterals, tetrahedra, hexahedra, triangular wedges, and pyramids. The list of vertices (commonly referred to as the "connectivity") define the geometry (location and variation of shape) for the element. The vertices are expected to be in a pre-defined order specific to the element topology. Mesquite uses the canonical ordering defined in the ExodusII specification[19].

For some more advanced capabilities, Mesquite may require the ability to attach arbitrary pieces of data (called "tags") to mesh elements or vertices.

4.1 The `Mesquite::Mesh` Interface

The `Mesquite::Mesh` class (`MeshInterface.hpp`) defines the interface Mesquite uses to interact with mesh data. In C++ this means that the class defines a variety of pure virtual (or abstract) functions for accessing mesh data. An application may implement a subclass of `Mesquite::Mesh`, providing implementations of the virtual methods that allow Mesquite direct access to the applications in-memory mesh representation.

The `Mesquite::Mesh` interface defines functions for operations such as:

- Get a list of all mesh vertices.
- Get a list of all mesh elements.
- Get a property of a vertex (coordinates, fixed flag, etc.)
- Set a property of a vertex (coordinates, "byte", etc.)

- Get the type of an element
- Get the vertices in an element

It also defines other operations that are only used for certain optimization algorithms:

- Get the list of elements for which a specific vertex occurs in the connectivity list.
- Define a "tag" and use it to associate data with vertices or elements.

Mesh entities (vertices and elements) are referenced in the `Mesquite::Mesh` interface using 'handles'. There must be a unique handle space for all vertices, and a separate unique handle space for all elements. That is, there must be a one-to-one mapping between handle values and vertices, and a one-to-one mapping between handle values and elements. The storage type of the handles is specified by

`Mesquite::Mesh::VertexHandle` and `Mesquite::Mesh::ElementHandle`.

The handle types are of sufficient size to hold either a pointer or an index, allowing the underlying implementation of the `Mesquite::Mesh` interface to be either pointer-based or index-based. All mesh entities are referenced using handles. For example, the `Mesquite::Mesh` interface declares a method to retrieve the list of all vertices as an array of handles and a method to update the coordinates of a vertex where the vertex is specified as a handle.

It is recommended that the application invoke the Mesquite optimizer for subsets of the mesh rather than the entire mesh whenever it makes sense to do so. For example, if a mesh of two geometric volumes is to be optimized and all mesh vertices lying on geometric surfaces are constrained to be fixed (such vertices will not be moved during the optimization) then optimizing each volume separately will produce the same result as optimizing both together.

4.2 Accessing Mesh In Arrays

One common representation of mesh in applications is composed of simple coordinate and index arrays. Mesquite provides the `ArrayMesh` implementation of the `Mesquite::Mesh` interface to allow Mesquite to access such array-based mesh definitions directly. The mesh must be defined as follows:

- Vertex coordinates must be stored in an array of double-precision floating-point values. The coordinate values must be interleaved, meaning that the x, y, and z coordinate values for a single vertex are contiguous in memory.
- The mesh must be composed of a single element type.
- The element connectivity (vertices in each element) must be stored in an interleaved format as an array of long integers. The vertices in each element are specified by an integer `i`, where the location of the coordinates of the corresponding vertex is located at position `3*i` in the vertex coordinates array.
- The fixed/boundary state of the vertices must be stored in an array of integer values, where a value of 1 indicates a fixed vertex and a value of 0 indicates a free vertex. The values in this array must be in the same order as the corresponding vertex coordinates in the coordinate array.

The following is a simple example of using the ArrayMesh object to pass Mesquite a mesh containing four quadrilateral elements.

```
/** define some mesh **/
    /* vertex coordinates */
  double coords[] = { 0, 0, 0,
                      1, 0, 0,
                      2, 0, 0,
                      0, 1, 0,
                     .5,.5, 0,
                      2, 1, 0,
```

```
                        0, 2, 0,
                        1, 2, 0,
                        2, 2, 0 };
    /* quadrilateral element connectivity (vertices) */
  long quads[] = { 0, 1, 4, 3,
                   1, 2, 5, 4,
                   3, 4, 7, 6,
                   4, 5, 8, 7 };
    /* all vertices except the center one are fixed */
  int fixed[] = { 1, 1, 1,
                  1, 0, 1,
                  1, 1, 1 };

/** create an ArrayMesh to pass the above mesh into Mesquite **/

  ArrayMesh mesh(
      3,             /* 3D mesh (three coord values per vertex) */
      9,             /* nine vertices */
      coords,        /* the vertex coordinates */
      fixed,         /* the vertex fixed flags */
      4,             /* four elements */
      QUADRILATERAL, /* elements are quadrilaterals */
      quads );       /* element connectivity */

/** smooth the mesh **/

    /* Need surface to constrain 2D elements to */
  PlanarDomain domain( PlanarDomain::XY );

  MsqError err;
  ShapeImprover shape_wrapper;
  if (err) {
    std::cout << err << std::endl;
    exit (2);
  }

  shape_wrapper.run_instructions( &mesh, &domain, err );
  if (err) {
    std::cout << "Error smoothing mesh:" << std::endl
              << err << std::endl;
  }

/** Output the new location of the center vertex **/
  std::cout << "New vertex location: ( "
            << coords[12] << ", "
            << coords[13] << ", "
            << coords[14] << " )" << std::endl;
```

NOTE: When using the `ArrayMesh` interface, the application is responsible for managing the storage of the mesh data. The `ArrayMesh` does NOT copy the input mesh.

4.3 Reading Mesh From Files

Mesquite provides a concrete implementation of the `Mesquite::Mesh` named `Mesquite::MeshImpl`. This implementation is capable of reading mesh from VTK[22, 3] and optionally ExodusII files. See Sections

2.1.3 and 2.2 for more information regarding the optional support for ExodusII files.

The 'fixed' flag for vertices can be specified in VTK files by defining a SCALAR POINT_DATA attribute with values of 0 or 1, where 1 indicates that the corresponding vertex is fixed. The `Mesquite::MeshImpl` class is capable of reading and storing tag data using VTK attributes and field data. The current implementation writes version 3.0 of the VTK file format and is capable of reading any version of the file format up to 3.0.

4.4 ITAPS iMesh Interface

4.4.1 Introduction

The ITAPS Working Group has defined a standard API for exchange of mesh data between applications. The iMesh interface[20] defines a superset of the functionality required for the `Mesquite::Mesh` interface. Mesquite can access mesh through an iMesh interface using the `Mesquite::MsqIMesh` class declared in `MsqIMesh.hpp`. This class is an "adaptor": it presents the iMesh interface as the `Mesquite::Mesh` interface.

The primary advantage of this method of providing mesh data to mesquite is that it is designed for interoperability. The same API can be used to provide other tools and services access to the mesh data. And there are stand-alone mesh data base libraries that already implement this API such as MOAB[2] and FMDB[1]. It is also possible to implement the iMesh interface in Fortran.

4.4.2 Overview

A `Mesquite::MsqIMesh` instance must be provided with at least two pieces of information: The `iMesh_Instance` handle and an `iBase_EntitySetHandle`. The optional `iBase_TagHandle` for the "fixed tag" must frequently be provided as well. The `iMesh_Instance` specifies the instance of the database containing the mesh. The `iBase_EntitySetHandle` handle specifies the subset of that mesh that is to be optimized by Mesquite. If the entire mesh is to be optimized then the "root set" should be specified for this argument. The quality of all elements in this set will be used to drive the mesh optimization. All vertices adjacent to any elements in the set will be moved as a part of the optimization unless they are explicitly designated as fixed. The "fixed tag" is used to indicate such vertices. Every vertex adjacent to the input elements should be tagged with a single integer value of either zero or one for the "fixed tag". A value of one indicates that the vertex is fixed while a value of zero indicates that the vertex location is to be optimized by Mesquite.

The boundary of the mesh must always be constrained in some way for the mesh optimization to produce valid results. For a volume mesh this can be accomplished by either designating the vertices on the mesh boundary as fixed or by specifying a geometric domain (e.g. surfaces, curves, etc) that the boundary vertices are constrained to lie on. For a surface mesh some geometric domain must always be specified (e.g. a surface) and it is still necessary to specify which vertices are fixed unless the geometric domain also includes the bounding geometric curves constraining the movement of the boundary mesh vertices[1]. Geometric domains are the topic of Chapter 5. Further discussion and examples in this section will be limited to volume meshes and true 2D meshes, both with the boundary vertices designated as fixed via the "fixed tag".

Designating vertices as fixed is the responsibility of the application using Mesquite. This responsibility is left to the application (as opposed to providing some utility in Mesquite to find the "skin" of a mesh) for several reasons. An application can often obtain the set of vertices bounding a region of mesh directly through data not available to Mesquite. For example if the application has a B-rep solid model for which the mesh is a discretization then it typically can obtain the bounding vertices as the set of vertices associated with the bounding geometric entities. Further, there exist cases where the fixed vertices are more than just those on the topological boundary of the mesh. For example, consider the mesh of a conic surface that includes a vertex at the apex of the cone. Such a vertex must be designated as fixed because the lack of a valid surface normal at that point will interfere with the correct functioning of Mesquite. Such a vertex cannot be reliably identified given only the mesh. However, identifying such vertices

[1] A surface mesh that forms a topological sphere has no boundary and therefore need not have vertices designated as fixed or otherwise constrained as long as the entire geometric domain is continuous.

typically happens naturally when obtaining the set of fixed vertices from the association with bounding geometric entities. Finally, the optimal implementation of a "skinning" operation depends greatly on details of the mesh representation that Mesquite is not aware of and is not otherwise concerned with.

4.4.3 Practical Details

The `Mesquite::MsqIMesh` class caches data related to the input `iBase_EntitySetHandle` upon construction. If the contents of the referenced entity set change, or the vertices associated with elements contained in that set change, then the application should either re-create the `Mesquite::MsqIMesh` instance or notify an existing instance of the change by calling the `set_active_set` member function. Similarly, while the implementation does not at the time of this writing cache data related to the "fixed tag", it may do so in the future. For forward compatibility the application should consider calling the `set_fixed_tag` method of `Mesquite::MsqIMesh` to notify the instance that the value of the tag may have changed for some mesh vertices.

The current version of Mesquite uses the following functions from the iMesh interface:

- `iMesh_getRootSet`
- `iMesh_getGeometricDimension`
- `iMesh_getEntities`
- `iMesh_getNumOfType`
- `iMesh_isEntContained`
- `iMesh_getEntArrTopo`
- `iMesh_getEntArrAdj`
- `iMesh_getVtxArrCoords`
- `iMesh_setVtxCoord`
- `iMesh_createTag`
- `iMesh_destroyTag`
- `iMesh_getTagName`
- `iMesh_getTagSizeBytes`
- `iMesh_getTagType`
- `iMesh_getTagHandle`
- `iMesh_getIntArrData`
- `iMesh_getIntData`
- `iMesh_getArrData`
- `iMesh_setArrData`
- `iMesh_setIntData`
- `iMesh_setIntArrData`

An implementation should provide complete implementations of all of these methods to guarantee compatibility with all possible Mesquite algorithms.

4.4.4 Volume Example

The following example demonstrates the use of the `ShapeImprover` wrapper with an implementation of the iMesh interface. It is assumed that the application has implemented the iMesh interface to provide access to its own data or is using an existing implementation of the iMesh interface to store its mesh data. The example illustrates the setup necessary to correctly pass a subset of a mesh to mesquite and how to designate boundary verices as fixed using the "fixed tag". The input to the example function is the `iMesh_Instance` handle and an `iBase_EntitySetHandle` specifying both the elements for which to improve the quality and the free vertices. The example code uses this application-supplied designation of which vertices are fixed to initalize the "fixed tag".

```
#include <MsqError.hpp>
#include <ShapeImprover.hpp>
#include <MsqIMesh.hpp>
#include <vector>
#include <iostream>
#include <iMesh.h>

using namespace Mesquite;

/**\brief Call Mesquite ShapeImprovement wrapper for volume mesh
 *
 * Smooth mesh accessed through ITAPS APIs using Mesquite
 * ShapeImprover.
 *
 *\param mesh_instance iMesh API instance
 *\param mesh A set defined in 'mesh_instance' that contains
 *            *both* the set of elements to smooth *and* the
 *            set of interior vertices that are to be moved
 *            to improve the quality of the mesh.  This set
 *            must *not* contain vertices on the boundary of
 *            the volume mesh.
 *\return mesquite error code or imesh error code
 *        (0 for success in all cases.)
 */
int shape_improve_volume( iMesh_Instance mesh_instance,
                          iBase_EntitySetHandle mesh )
{
  MsqPrintError err(std::cerr);
  int ierr;
  iBase_EntityHandle *ptr1, *ptr2;
  int *ptr3, *ptr4;
  int i5, i6, i7, i8, i9, i10, i11;
  const int elem_dim = 3;
  const int max_vtx_per_elem = 8;

    // create adapter (should also create fixed tag)
  MsqIMesh mesh_adapter( mesh_instance, mesh, elem_dim, err );
  if (err) return err.error_code();

    // get tag for marking vertices as fixed
    // Note: we assume here that the tag has already been created.
  iBase_TagHandle fixed_tag = 0;
  iMesh_getTagHandle( mesh_instance,
                      "fixed",
```

```
                      &fixed_tag,
                      &ierr,
                      strlen("fixed") );
if (iBase_SUCCESS != ierr) return ierr;

  // get all vertices in mesh
int count, num_vtx;
iMesh_getNumOfType( mesh_instance, mesh, elem_dim, &count, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
std::vector<iBase_EntityHandle> elems(count), verts(max_vtx_per_elem*count);
std::vector<int> indices(max_vtx_per_elem*count), offsets(count+1);
ptr1 = &elems[0];
ptr2 = &verts[0];
ptr3 = &indices[0];
ptr4 = &offsets[0];
i5 = elems.size();
i7 = verts.size();
i8 = indices.size();
i10 = offsets.size();
iMesh_getAdjEntIndices( mesh_instance, mesh,
                        elem_dim, iMesh_ALL_TOPOLOGIES, iBase_VERTEX,
                        &ptr1, &i5, &i6,
                        &ptr2, &i7, &num_vtx,
                        &ptr3, &i8, &i9,
                        &ptr4, &i10, &i11, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
verts.resize( num_vtx );

  // set fixed flag on all vertices
std::vector<int> tag_data(num_vtx, 1);
iMesh_setIntArrData( mesh_instance, &verts[0], verts.size(),
                     fixed_tag, &tag_data[0], tag_data.size(), &ierr );
if (iBase_SUCCESS != ierr) return ierr;

  // clear fixed flag for vertices contained directly in set
iMesh_getNumOfType( mesh_instance, mesh, iBase_VERTEX, &count, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
verts.resize( count );
ptr1 = &verts[0];
i5 = verts.size();
iMesh_getEntities( mesh_instance, mesh, iBase_VERTEX, iMesh_ALL_TOPOLOGIES,
                   &ptr1, &i5, &i6, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
tag_data.clear();
tag_data.resize( verts.size(), 0 );
iMesh_setIntArrData( mesh_instance, &verts[0], verts.size(),
                     fixed_tag, &tag_data[0], tag_data.size(), &ierr );
if (iBase_SUCCESS != ierr) return ierr;

  // Finally, smooth the mesh
ShapeImprover smoother;
smoother.run_instructions( &mesh_adapter, err );
if (err) return err.error_code();

return 0;
```

```
}
```

4.4.5 Two-dimensional Example

This section presents an example of how to use Mesquite to optimize a 2D mesh. It is a modification of the example from the previous section with changes shown in blue. As Mesquite operates only on 3D meshes (either volume or surface), a 2D mesh is optimized by treating it as a surface mesh constrained to the XY plane.

```
#include <MsqError.hpp>
#include <ShapeImprover.hpp>
#include <MsqIMesh.hpp>
#include <PlanarDomain.hpp>
#include <vector>
#include <iostream>
#include <iMesh.h>

using namespace Mesquite;

/**\brief Call Mesquite ShapeImprovement wrapper for 2D mesh
 *
 * Smooth mesh accessed through ITAPS APIs using Mesquite
 * ShapeImprover.
 *
 *\param mesh_instance iMesh API instance
 *\param mesh A set defined in 'mesh_instance' that contains
 *            *both* the set of elements to smooth *and* the
 *            set of interior vertices that are to be moved
 *            to improve the quality of the mesh.  This set
 *            must *not* contain vertices on the boundary of
 *            the mesh.
 *\return mesquite error code or imesh error code
 *        (0 for success in all cases.)
 */
int shape_improve_2D( iMesh_Instance mesh_instance,
                      iBase_EntitySetHandle mesh )
{
  MsqPrintError err(std::cerr);
  int ierr;
  iBase_EntityHandle *ptr1, *ptr2;
  int *ptr3, *ptr4;
  int i5, i6, i7, i8, i9, i10, i11;
  const int elem_dim = 2;
  const int max_vertex_per_elem = 4;

    // create adapter (should also create fixed tag)
  MsqIMesh mesh_adapter( mesh_instance, mesh, elem_dim, err );
  if (err) return err.error_code();

    // get tag for marking vertices as fixed
    // Note: we assume here that the tag has already been created.
  iBase_TagHandle fixed_tag = 0;
  iMesh_getTagHandle( mesh_instance,
                      "fixed",
```

```
                      &fixed_tag,
                      &ierr,
                      strlen("fixed") );
if (iBase_SUCCESS != ierr) return ierr;

  // get all vertices in mesh
int count, num_vtx;
iMesh_getNumOfType( mesh_instance, mesh, elem_dim, &count, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
std::vector<iBase_EntityHandle> elems(count), verts(max_vtx_per_elem*count);
std::vector<int> indices(max_vtx_per_elem*count), offsets(count+1);
ptr1 = &elems[0];
ptr2 = &verts[0];
ptr3 = &indices[0];
ptr4 = &offsets[0];
i5 = elems.size();
i7 = verts.size();
i8 = indices.size();
i10 = offsets.size();
iMesh_getAdjEntIndices( mesh_instance, mesh,
                        elem_dim, iMesh_ALL_TOPOLOGIES, iBase_VERTEX,
                        &ptr1, &i5, &i6,
                        &ptr2, &i7, &num_vtx,
                        &ptr3, &i8, &i9,
                        &ptr4, &i10, &i11, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
verts.resize( num_vtx );

  // set fixed flag on all vertices
std::vector<int> tag_data(num_vtx, 1);
iMesh_setIntArrData( mesh_instance, &verts[0], verts.size(),
                     fixed_tag, &tag_data[0], tag_data.size(), &ierr );
if (iBase_SUCCESS != ierr) return ierr;

  // clear fixed flag for vertices contained directly in set
iMesh_getNumOfType( mesh_instance, mesh, iBase_VERTEX, &count, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
verts.resize( count );
ptr1 = &verts[0];
i5 = verts.size();
iMesh_getEntities( mesh_instance, mesh, iBase_VERTEX, iMesh_ALL_TOPOLOGIES,
                   &ptr1, &i5, &i6, &ierr );
if (iBase_SUCCESS != ierr) return ierr;
tag_data.clear();
tag_data.resize( verts.size(), 0 );
iMesh_setIntArrData( mesh_instance, &verts[0], verts.size(),
                     fixed_tag, &tag_data[0], tag_data.size(), &ierr );
if (iBase_SUCCESS != ierr) return ierr;

  // Finally, smooth the mesh
ShapeImprover smoother;
PlanarDomain xyplane(PlanarDomain::XY);
smoother.run_instructions( &mesh_adapter, &xyplane, err );
if (err) return err.error_code();
```

```
  return 0;
}
```

Chapter 5

Constraining Mesh to a Geometric Domain

Vertex positions may be constrained to a geometric domain by providing Mesquite with an optional instance of the `Mesquite::MeshDomain` interface. This interface provides two fundamental capabilities: mesh-geometry classification, and interrogation of local geometric properties. The methods defined in the `Mesquite::MeshDomain` interface combine both queries into single operation. Queries are passed a mesh entity handle (see Section 4.1), and are expected to interrogate the geometric domain that the specified mesh entity is classified to.

If Mesquite is used to optimize the mesh of a B-Rep solid model (the data model used by all modern CAD systems), then the domain is composed of geometric vertices, curves, surfaces, and volumes. Curves are bounded by end vertices, surfaces are bounded by loops (closed chains) of curves, and volumes are bounded by groups of surfaces. Mesquite expects each surface element (triangle, quadrilateral, etc.) to be associated with a 2D domain (surface). Vertices may be associated with a geometric entity that either contains adjacent mesh elements or bounds the geometric entity containing the adjacent elements. Mesquite does not use geometric volumes. A query for the closest location on the domain for a vertex or element whose classification is a geometric volume should simply return the input position.

It is possible to define an optimization problem such that mesh classification data need not be provided in a `Mesquite::MeshDomain` implementation. This is done by optimizing the mesh associated with each simple geometric component of the domain separately, with the boundary vertices flagged as fixed. The following pseudo-code illustrates such an approach for a B-Rep type geometric domain:

```
for each geometric vertex
 mark associated vertex as fixed
end-for
for each curve
 do any application-specific optimization of curve node placement
 mark associated mesh vertices as fixed
end-for
for each surface
 define Mesquite::MeshDomain for surface geometry
 invoke Mesquite to optimize surface mesh
 mark all associated mesh vertices as fixed
end-for
for each volume
 invoke Mesquite to optimize volume mesh w/o Mesquite::MeshDomain
end-for
```

5.1 The ITAPS iGeom and iRel Interfaces

Mesquite can access mesh domain data through the *iGeom* and *iRel* interface defined by the ITAPS Work Group. These interfaces provide APIs for accessing B-Rep geometric data and associating mesh with geometry (classifiation), respectively. Mesquite provides the `Mesquite::MsqIGeom` class (`MsqIGeom.hpp`) as an adaptor for interfacing with appications that present the *iGeom* and *iRel* interfaces. The use of the *iRel* interface is optional. If all the mesh vertices are constrained to a single geometric surface, it is sufficient to provide only an iGeom instance to `Mesquite::MsqIGeom`. If vertices are constrained to different geometric entities, then the *iRel* interface must be provided to `Mesquite::MsqIGeom` so Mesquite can determine which iMesh entity a given vertex is constrained to lie in.

5.2 Simple Geometric Domains

Mesquite provides several implementations of the `Mesquite::MeshDomain` interface for simple geometric primitives. All MeshDomains in Mesquite are geometric surfaces upon which meshes consisting of triangles and/or quadrilaterals can exist. Mesquite does not have any implementations of 3D geometric regions. The domains available in Mesquite include:

- `PlanarDomain`: An unbounded planar surface.
- `XYPlanarDomain`: An unbounded planar surface that exists in the XY-plane.
- `SphericalDomain`: A closed spherical surface.
- `CylinderDomain`: An unbounded cylindrical surface.
- `BoundedCylinderDomain`: A bounded cylindrical surface.
- `ConicDomain`: An unbounded cone with a circular cross-section.
- `XYRectangle`: An bounded rectangular domain in the XY-Plane.

The `PlanarDomain` is often used to map $\mathbb{R}^2$ optimization problems to $\mathbb{R}^3$. The others are used primarily for testing purposes.

Notes about Domains:

- The `BoundedCylinderDomain` provides some simplistic mesh-geometry classification capabilities. The others do not provide any classification functionality. Creating a bounded Cylinder is a two-step process. First, a cylinder is created via the constructor by specifying a radius, a vector defining the direction of the axis, and a point through which the axis passes. Second, the bounding part is specified by calling one of the two overloaded methods "create_curve()". Both versions accept a distance from the axis where the circular curve to act at the bounding box will be placed along with vertices to be considered bound to the curve. The vertices are specified by either a list or a mesh depending on which version of the method is used.
- The `ConicDomain` is not bounded at the apex. It extends infinitely in both directions.
- The `XYPlanarDomain` is the only MeshDomain type that can be used with FeasibleNewton optimization. FeasibleNewton also operates on volume meshes.
- The `XYRectangle` domain is a simple 2D domain used for free-smooth testing. The specified rectangle can be in the XY, YZ, or ZX plane. The constructor takes as input a point (x,y,z), a height and width, and a plane. A cooresponding bounding box is then created in the specified plane. The method "setup(iMesquite::Mesh* mesh, Mesquite::MsqError& err)" can then be used to determine if a particular mesh lies completely in the defined bounded rectangle. If any of the vertices of the mesh lie outside the rectangle a non-zero err value will be returned.

Chapter 6

Mesquite Wrapper Descriptions

Applications which desire to access Mesquite capabilities without delving into the low-level API can invoke wrappers to perform basic mesh quality improvement tasks that, except for a few user-defined inputs, are fully automatic. The wrappers target classic mesh optimization problems that occur repeatedly across many applications. See section 3.1.3 for an example of how to invoke a wrapper. This chapter provides a summary of the current Mesquite wrappers.

Note that the wrappers do not, by themselves, completely define the optimization problem. The user still has to set the fixed/free flags, and the values of the termination criteria.

6.1 Laplace-smoothing

Name: LaplaceWrapper
Purpose: Produce a *smooth* mesh.
Notes: This is a local patch relaxation-solver. A 'smart' Laplacian solver is also available in Mesquite, but it is not used in this wrapper.
Limitations/assumptions: No invertibility guarantee.
Input Termination Criterion: Stop after 10 global iterations.

Under the Cover:
Hardwired Parameters: None
Mesh/Element Type: Any supported type.
Global/Local: Local Patch with Culling

6.2 Shape-Improvement

Name: ShapeImprover
Purpose: Make the shape of an element as close as possible to that of the ideal/regular element shape. For example, make triangular and tetrahedral elements equilateral. The wrapper will use a non-barrier metric until the mesh contains no inverted elements. If the initial mesh was not tangled this phase will not modify the mesh. If no limit is imposed on CPU time or time remains a second phase using a barrier metric will further optimize the mesh with the guarantee that no elements will become inverted.
Notes: Will return failure if mesh contains inverted elements after first phase.
Limitations/assumptions:
Input Termination Criterion: CPU time limit (not used if input value is non-positive) or fraction of mean edge length (default is 0.005).

Under the Cover:
Metric: TMPQualityMetric(Shape/ShapeBarrier)
Objective Function: Algebraic mean of quality metric values
Mesh/Element Type: Any supported type.
Solver: Conjugate Gradient
Global/Local: Global

6.3 Untangler

Name: UntangleWrapper
Purpose: Untangle elements. Prioritizes untangling over element shape or other mesh quality measures.
Notes: A second optimization to improve element quality after untangling is often necessary.
Limitations/assumptions: There is no guarantee that the optimal mesh computed using this wrapper will, in fact, be untangled.
Input Termination Criterion: CPU time limit (not used if input value is non-positive) or fraction of mean edge length (default is 0.005). It also terminates if all elements are untangled, such that it should not modify an input mesh with no inverted elements.

Under the Cover:
Metric: TUntangleBeta or TUntangleMu(TSizeNB1) or TUntangleMu(TShapeSizeNB1)
Objective Function: Algebraic mean of quality metric values
Mesh/Element Type: Any supported type.
Solver: Steepest Descent
Global/Local: Local with culling, optionally Jacobi

6.4 Minimum Edge-Length Improvement

Name: PaverMinEdgeLengthWrapper
Purpose: Make all the edges in the mesh of equal length while moving toward the ideal shape. Intended for explicit PDE codes whose time-step limination is governed by the minimum edge-length in the mesh.
Notes: Based on Target-matrix paradigm.
Limitations/assumptions: Initial mesh must be non-inverted. User does not want to preserve or create anisotropic elements.
Input Termination Criterion: maximum iterations (default=50), maximum absolute vertex movement

Under the Cover:
Hardwired Parameters: None
Metric: Target2DShapeSizeBarrier or Target3DShapeSizeBarrier
Tradeoff Coefficient: 1.0
Objective Function: Linear Average over the Sample Points
Mesh/Element Type: Any supported type.
Solver: Trust Region
Global/Local: Global

6.5 Improve the Shapes in a Size-adapted Mesh

Name: SizeAdaptShapeWrapper
Purpose: Make the shape of an element as close as possible to that of the ideal element shape, *while preserving, as much as possible, the size of each element in the mesh*. To be used on isotropic initial meshes that are already size-adapted.
Notes: Based on Target-matrix Paradigm.
Limitations/assumptions: Initial mesh must be non-inverted. User wants to preserve sizes of elements in initial mesh and does not want to preserve or create anisotropic elements.
Input Termination Criterion: maximum iterations (default=50), maximum absolute vertex movement

Under the Cover:
Hardwired Parameters: None
Metric: Target2DShapeSizeBarrier or Target3DShapeSizeBarrier
Tradeoff Coefficient: 1.0
Objective Function: Linear Average over the Sample Points
Mesh/Element Type: Any supported type.
Solver: Trust Region
Global/Local: Global

6.6 Improve Sliver Tets in a Viscous CFD Mesh

Name: ViscousCFDTetShapeWrapper
Purpose: Improve the shape of sliver elements in the far-field of a CFD mesh while preserving an existing layer of sliver elements in the boundary layer.
Notes: Based on Target-matrix paradigm.
Limitations/assumptions: Tetrahedral meshes only.
Input Termination Criterion: Iteration Count (default=50) or Maximum Absolution Vertex Movement

Under the Cover:
Hardwired Parameters: In tradeoff coefficient model.
Metric: Target2DShape+Target2DShapeSizeOrient (or 3D) (or Barrier)
Tradeoff Coefficient: Based on element dihedral angle
Objective Function: Linear average over Sample Points
Mesh/Element Type: Tetrahedra
Solver: Trust Region
Global/Local: Global

6.7 Deforming Domain

Name: DeformingDomainWrapper
Purpose: Use initial mesh on undeformed geometric domain to guide optimization of mesh moved to deformed geometric domain.
Notes: Uses a non-barrier metric which means that the wrapper could potentially invert/tangle elements.
Limitations/assumptions: Application responsible for explicit handling of mesh on geometric curves and points. Initial mesh before moving to deformed domain must be available.
Input Termination Criterion: CPU time limit (not used if input value is non-positive) or fraction of mean edge length (default is 0.005).

Under the Cover:
Metric: TMPQualityMetric(TShapeNB1 or TShapeSizeNB1 or TShapeSizeOrientNB1)
Objective Function: Algebraic mean of quality metric values
Mesh/Element Type: Any supported type.
Solver: Steepest Descent
Global/Local: Local with culling

Chapter 7

Optimization Strategies

7.1 The Generalized Optimization Loop

In Mesquite a generalization of the optimization strategy is used to implement a wide variety of optimization strategies. Before discussing the different types of optimization strategies that can be implemented with Mesquite we will first need to discuss the generalized strategy.

The mesh can be decomposed into subsets called *patches*. The specifics of this mesh decomposition are discussed in Section 7.2. The optimization is done by repeatedly iterating over the set of patches, optimizing each separately.

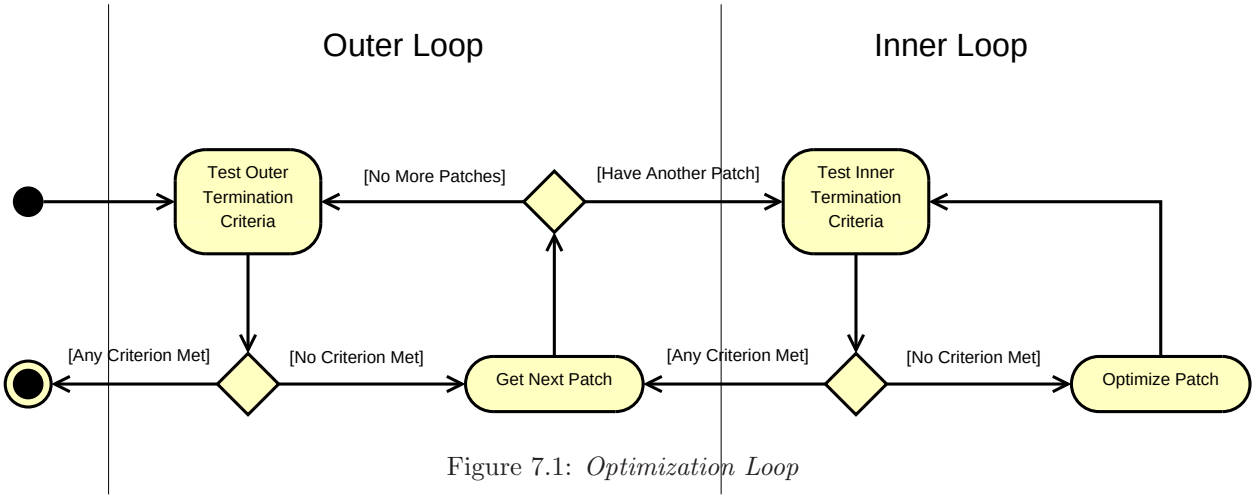

Figure 7.1: *Optimization Loop*

Figure 7.1 depicts the generalization of optimization strategies in Mesquite. The generalized optimization is composed of three loops shown as non-overlapping square cycles in the diagram. The test to exit each loop is performed at the decision points (diamonds) in the diagram. The loops are logically nested from left to right, such that the right most loop is performed within a single iteration of the loop to the left of it. The inner- and outer-most loops terminate based on user-definable termination criterion. The center loop is the iteration over the set of patches composing the mesh.

The inner-most loop (the right-most cycle in the diagram) represents the iterative optimization of the mesh contained in a single patch. This optimization is done until the *inner termination criterion* is met. Once the inner criterion is met the optimizer advances to the next patch and the inner loop is entered again to optimize that patch. Once each patch has been optimized the *outer termination criterion* is tested. If the criterion has not been met then the loop over the set of patches is repeated.

The set of outer termination criteria determine when the optimization of the entire mesh is complete. The set of inner termination criteria determine when the optimization of a single patch is complete. Both sets of criteria are tested before entering their respective loops. If a criterion is met before the loop starts then no iterations of the corresponding loop will be performed.

The outer loop(s) are implemented in the VertexMover class. The inner loop is implemented in subclasses. The LaplacianSmoother class in Mesquite provides a traditional Laplace smoother. For this class the mesh is decomposed into patches that each contain a single free vertex and the adjacent elements, one patch for each free vertex in the mesh. For Laplace smoothing the inner (per-vertex) optimization is not iterative. The inner loop always has an implicit termination criterion of a single iteration. Any other inner termination criterion will still be tested before performing the relaxation of the free vertex in the patch such that if any such criterion is met no optimization of the vertex will be performed. However, culling (Section 7.6 can have a similar effect while typically producing better results. Passes are made over the entire mesh until one of the specified outer termination criterion is met.

7.2 Patches

Mesquite can operate on a decomposition of the mesh into subsets called *patches*. Each patch is optimized individually. The overall mesh optimization is performed by repeatedly iterating over the set of patches. Mesquite provides two build-in mesh decompositions[1]: element-on-vertex patches and a global patch.

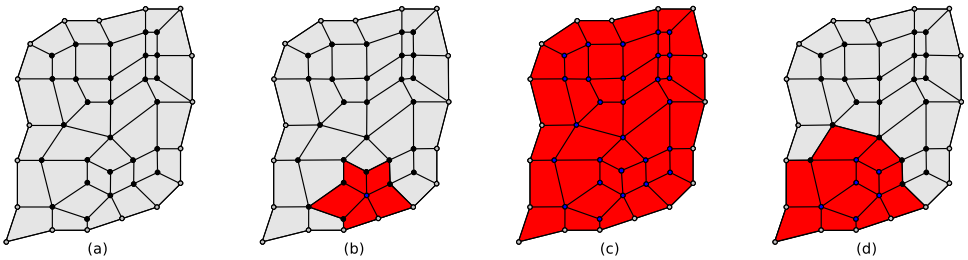

Figure 7.2: Miscellaneous patch configurations.

The global patch is a "decomposition" where the entire mesh is contained in a single patch. This is used in the global optimization strategy discussed in Section 7.3. Figure 7.2c illustrates the global patch.

The element-on-vertex decomposition subdivides the mesh into a single patch for each free vertex. Each patch includes the layer of elements adjacent to the free vertex. A element-on-vertex patch is illustrated in Figure 7.2b. This decomposition is typically used for all optimization strategies discussed in this chapter except for global optimization. Any other decomposition except global may be used for any of the optimization strategies. All of the discussed strategies other than global do not make sense for a global patch.

Any patch decomposition can be used with Mesquite. While no other decomposition strategy is provided with Mesquite, the any implementation of the PatchSet interface can be associated with any quality improver that supports it (any subclass of PatchSetUser, currently all except LaplacianSmoother). An implementation of that interface is expected to provide three things:

1. An enumeration of all the patches in the decomposition of the mesh
2. For each patch, the set of vertices to optimize
3. For each patch, the set of elements for which the quality is to be optimized (typically all elements containing the vertices to be optimized.)

A normal decomposition will be done such that each free vertex in the mesh is optimized in exactly one patch, but Mesquite does not enforce this. Having a free vertex be optimized in no patch will result in that vertex effectively being fixed for the optimization. A decomposition that optimized the same vertices in multiple patches is allowable, and should have no adverse side effects unless doing a Jacobi optimization (Section 7.7).

The listing below shows how a custom implementation of the PatchSet interface can be used with the SteepestDescent solver.

[1]Mesquite includes an additional decomposition of the mesh into single-element patches which is not suitable for use in optimization. It is used internally for quality assessment and other purposes

```
MyMeshDecomposition my_patch_set;
SteepestDescent quality_improver( &objective_function );
quality_improver.use_patch_set(&my_patch_set);
```

In any of the examples later in this chapter that call use_element_on_vertex_patch(), that call may be sustitued with a call to use_patch_set to use some decomposition other than single-vertex patches.

7.3 Global

For a global optimization an objective function that measues the quality of the mesh is minimized using a numerical solver. The coordinates of all of the free vertices in the mesh are the free variables in the optimization. This is the default mode of operation for most solver-based implementations of the QualityImprover interface.

A global optimization is simplest form of the generalized optimization loop. In this mode the mesh is "decomposed" into a single patch containing the entire mesh. The outer loops in Figure 7.1 are executed only once. The entire optimization process happens in the inner loop. For global optimization the outer termination criterion is the default of a single iteration. The inner termination criterion should be used to terminate the optimization process. Setting some other outer termination criterion is not prohibited, but will result in a much less efficient optimization process. There is no logical difference between inner and outer termination criterion, but each iteration of the outer loop begins with a clean solver state which will result in less efficient operation of the solver. Even steepest descent, the simplest solver, caculates a initial step size based on the previous iteration of the inner loop.

The listing below shows how global optimization can be selected.

```
// Create global optmizer instance
SteepestDescent improver( &objective_function );
improver.use_global_patch();

// Set only inner termnation criterion for
// global optimization
TerminationCriterion inner;
inner.add_absolute_vertex_movement( 1e-3 );
improver.set_inner_termination_criterion( &inner );

// Run optimization
InstructionQueue queue;
queue.set_master_quality_improver( &improver, err );
queue.run_instructions( &mesh, err );
```

7.4 Nash Game

```
// Create Nash optmizer instance
SteepestDescent improver( &objective_function );
improver.use_element_on_vertex_patch();

// Set inner and outer termnation criterion for
// non-global patch
TerminationCriterion inner, outer;
outer.add_absolute_vertex_movement( 1e-3 );
inner.add_iteration_limit( 2 );
improver.set_outer_termination_criterion( &outer );
improver.set_inner_termination_criterion( &inner );
```

```
// Run optimization
InstructionQueue queue;
queue.set_master_quality_improver( &improver, err );
queue.run_instructions( &mesh, err );
```

7.5 Block Coordinate Descent

```
// Create BCD optmizer instance
SteepestDescent improver( &objective_function );
improver.use_element_on_vertex_patch();
improver.do_block_coordinate_descent_optimization();

// Set inner and outer termnation criterion for
// non-global patch
TerminationCriterion inner, outer;
outer.add_relative_quality_improvement( 1e-2 );
inner.add_iteration_limit( 2 );
improver.set_outer_termination_criterion( &outer );
improver.set_inner_termination_criterion( &inner );

// Run optimization
InstructionQueue queue;
queue.set_master_quality_improver( &improver, err );
queue.run_instructions( &mesh, err );
```

7.6 Culling

```
// Create Nash optmizer with culling
SteepestDescent improver( &objective_function );
improver.use_element_on_vertex_patch();

// The culling criterion is effectively an outer
// termination criterion because optimization will
// always stop when all patches are culled.  We
// must explicitly pass an empty outer termination
// criterion to replace the default of one iteration.
// Additional outer termination criteria may also be
// specifed.
TerminationCriterion inner, outer;
inner.cull_on_absolute_vertex_movement( 1e-3 );
inner.add_iteration_limit( 2 );
improver.set_outer_termination_criterion( &outer );
improver.set_inner_termination_criterion( &inner );

// Run optimization
InstructionQueue queue;
queue.set_master_quality_improver( &improver, err );
queue.run_instructions( &mesh, err );
```

7.7 Jacobi

```
// Create Jacobi optmizer instance
SteepestDescent improver( &objective_function );
improver.use_element_on_vertex_patch();
improver.do_jacobi_optimization();

// Set inner and outer termnation criterion for
// non-global patch
TerminationCriterion inner, outer;
outer.add_absolute_vertex_movement( 1e-3 );
inner.add_iteration_limit( 2 );
improver.set_outer_termination_criterion( &outer );
improver.set_inner_termination_criterion( &inner );

// Run optimization
InstructionQueue queue;
queue.set_master_quality_improver( &improver, err );
queue.run_instructions( &mesh, err );
```

Chapter 8

Analyzing Optimizer Behavior

This chapter provides a brief overview of some of the tools provided in Mesquite for assisting with the analysis and visualization of the Mesquite optimization process. The tools discussed in this section can be used to provide additional output. External tools such as Paraview, VisIt, or GNU Plot must be used to visualize the data.

8.1 Assessing Quality

The QualityAssessor class provides a summary of the mesh quality. It can be used with Non Target-paradigm metrics (QualityMetric classes) as well as Target-paradigm metrics (TMetric classes). For simplicity, the following discussion refers to the QualityMetrics classes but the concepts apply to the TMetric classes as well. The QualityAssessor class can be used in a direct fashion as shown in the example below or via the InstructionQueue class as described in Section 3.1.4. An instance of the QualityMetric class can be specified for the QualityAssessor instance at creation to be used to assess the mesh quality. Additional QualityMetric instances can be created using the Assessor class and by adding them to the QualityAssessor instance via the "add_quality_assessment" method. If no QualityMetrics are specified, the only assessment that will be performed is a simple count of inverted elements. One or more instances of the QualityAssessor class may be inserted in the InstructionQueue at any point to print a summary of the mesh quality at that time during the optimization.

8.1.1 Stopping Assessment

A stopping assessment can be specified for each QualityAssessor instance. The "stopping assessment" directs the assessment code calculate a value using the power mean data to use that value as the return value for the loop_over_mesh call. If no power mean is specified for a QualityAssessor instance, a simple average of all metric values calculated during the assessment is returnedo from loop_over_mesh. Only one stopping assessment with its associated power mean can be specified for a particular QualityAssessor instance. There are three different ways to specify a stopping assessment: when the QualityAssessor instance is created using a constructor, when a quality assessment is added via the add_quality_assessment () method, and directly with the set_stopping_assessment() method. Since only one stopping assessment can be defined for a each instance of QualityAssessor, the last action that causes the stopping assessment to be set will be the one used for the assessment no matter how many metrics have been included.

8.1.2 Using the Quality Assessor

The QualityAssessor class provides a number of constructors. Each allows the specification of a different set parameters to control the quality assessment. The parameters are described below including default values, if any. Note that all parameters are not used in each constructor.

Parameters used by QualityAssesor constructors:

metric: QualityMetric to register for use in assessing mesh quality. Will also be used in the setting of the stopping assessment.

histogram_intervals: If non-zero, a histogram of quality metric values composed of the specified number of intervals will be generated. Default is zero.

power_mean: If non-zero, in addition to the normal summary statistics for the quality metric, an additional general power mean with the specified power will be calculated. Is used as the value set for the stopping assessment. Default is zero.

free_elements_only: When this option is TRUE, summary statistics are only computed over the set of elements which contain free vertices. If an element in the mesh does not contain a free vertex, its quality is not included in the summary. If an element in the mesh does not contain a free vertex, its quality cannot be improved by Mesquite. To compute the quality of all mesh elements, regardless of whether Mesquite can improve them, set this option to FALSE. Default is TRUE.

metric_value_tag_name: If a non-null value is specified, a tag with the specified name can be associated with quality values for individual elements or vertices if metric is an element-based or vertex-based metric. If metric is not element-based or vertex-based, this argument has no effect. The specified tag can then be associated with quality values generated for a mesh. Element-based metrics can have one tagged value per element quality value while vertex-based metrics can have one tagged value per vertex quality value. Tagged quality values are created using the methods tag_set_element_data() and tag_set_vertex_data() found in the MeshImpl class. The tagged values can be retrived using the methods tag_get_element_data() and tag_get_vertex()_data from the same class. Tags cannot be used with target metric classes. Default for metric_value_tag_name is null value.

inverted_element_tag_name: If a non-null value is specified, an integer tag with the specified name will be used to store a value of 0 for normal elements and 1 for inverted elements. Default is null value.

print_summary_to_std_out: If TRUE, summary of mesh quality will be written to std::out. If FALSE, quality assessment will be available via the get_results and get_all_results methods, but will not be printed. Default is TRUE.

output_stream IO: stream to which to write a summary of the mesh quality.

name: Name to include in output. Useful if several QualityAssessors are in use at the same time.

After the QualityAssessor instance is created, any of a number of methods can be used to set individual characteristics of the QualityAssessor object.

Once setup for the QualityAssessor object is complete, the actual assessment is preformed by calling "loop_over_mesh". After it terminates, results can be obtained using various methods supplied by the QualityAssessor class.

For each instance of the QualityAssessor, a summary of the results will be printed after the assessments have been completed. Display of the summary can be turned off by calling disable_printing_results() before the assessment is started. The summary will include data for each of the metric assessments run by the Assessor. The printed data includes the metric name, the minimum and maximum values, the average value, the rms (root mean square), and the standard deviation. If a power mean was specified for the assessment, an additional column will display the resultant value under a header containing the power mean value used in the calculation. All values in the QualityAssessor summary table are per mesh element. Any requested histograms are then displayed. The number of values in the histogram is dependant upon the type of metric performed. For element-based metrics, the histogram contains one value per element. For vertex-based metrics, it will contain the number of target sample points per element times the number of elements.

Below is an example of a summary and histogram for an eight element mesh for two different metrics, one that included a power mean of 1.5.

```
************** QualityAssessor(free only) Summary **************

  Evaluating quality for 8 elements.
  This mesh had 8 quadrilateral elements.
  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

       metric   minimum     average 1.5-mean   rms   maximum    std.dev.
Condition Number  1.05817     1.14257        1.1469   1.35948   0.0995044
     TSquared      1.18  2.23  2.28262      2.33533      3.77    0.693433

   TSquared histogram:
   (  1-1.3) |=====================3
   (1.3-1.6) |==============================4
   (1.6-1.9) |==============================4
   (1.9-2.2) |=============================================================================9
   (2.2-2.5) |===============2
   (2.5-2.8) |==============================4
   (2.8-3.1) |=======1
   (3.1-3.4) |=====================3
   (3.4-3.7) |=======1
   (3.7-4  ) |=======1
  metric was evaluated 32 times.
```

8.1.3 Quality Assessor Code Example

A simple example using the QualityAssessor class:

```
MsqError err;
MeshImpl meshToAssess;
PlanarDomain myDomain;
Settings mySettings;

meshToAssess.clear();

  // read in mesh
const char* filename = "meshToAssess.vtk";
meshToAssess.read_vtk( filename, err);

  // create metric instance
ConditionNumberQualityMetric metric;

  // create QualityAssessor instance accepting default values
QualityAssessor qa( &metric );

  // change some of the default parameters
qa.measure_free_samples_only( false );
qa.disable_printing_results();

  // run the QualityAssessor
qa.loop_over_mesh( &meshToAssess, &myDomain, &mySettings, err   );

  // get results
const QualityAssessor::Assessor* results = qa.get_results( &metric );
int invalid_element_count = results->get_invalid_element_count();
if ( invalid_element_count != 0 )
  std::cout << "Warning: " << invalid_element_count
                      << " invalid elements found." << std::endl;
```

8.1.4 Common-scale Histograms

When optimizating a mesh, it can be useful to display the quality before and after optimization. This is done by adding a QualityAssessor instance to an InstructionQueue, adding a quality improver instance to the InstructionQueue, and then adding the Quality Assessor instance to the InstructionQueue a second time. This allows a comparison of the mesh quality before and after optimization. Example code for doing this:

```
#include "Mesquite.hpp"
#include "MeshImpl.hpp"
#include "MsqError.hpp"
#include "InstructionQueue.hpp"
#include "TerminationCriterion.hpp"
#include "QualityAssessor.hpp"
#include "ConditionNumberQualityMetric.hpp"
#include "NonSmoothDescent.hpp"

#include "MeshImpl.hpp"
using namespace Mesquite;
```

```
int main()
{
  MsqPrintError err( std::cout );
  Mesquite::MeshImpl mesh;

    // read in mesh
  const char *file_name = "tire.vtk";\begin{lstlisting}[frame=single]
  mesh.read_vtk(file_name, err);
  if (err) return 1;

    // Create an intruction queue
  InstructionQueue queue1;

    // Create a condition number quality metric
  ConditionNumberQualityMetric cond_no;

    // Create the NonSmooth Steepest Descent procedures
  NonSmoothDescent minmax_method( &cond_no );

    // Set a termination criterion
  TerminationCriterion tc2;
  tc2.add_iteration_limit( 1 );
  minmax_method.set_outer_termination_criterion(&tc2);
    // Set up the quality assessor
  QualityAssessor quality_assessor = QualityAssessor(&cond_no);

    // assess the quality of the initial mesh
  queue1.add_quality_assessor(&quality_assessor, err);
  if (err) return 1;

    // Set the max min method to be the master quality improver
  queue1.set_master_quality_improver(&minmax_method, err);
  if (err) return 1;

    // assess the quality of the final mesh
  queue1.add_quality_assessor(&quality_assessor, err);
  if (err) return 1;

    // launches optimization on mesh_set1
  queue1.run_instructions(&mesh, err);
  if (err) return 1;

    // write out the smoothed mesh
  mesh.write_vtk("smoothed_mesh.vtk", err);
  if (err) return 1;

  return 0;
}
```

Creating Common-scale Histograms

Comparing before and after histograms can be difficult when there is a large difference in the resultant quality value range. In such cases, the common-scale histogram feature can be used to display two histograms with a common vertical interval scale and a common horizontal scale for the number of quality values that fall into each interval. Unlike the above example that reused the same QualityAssessor instance

for the before and after histograms, the common-scale histograms require two separate QualityAssessor instances. After both the before optimization and after optimization quality assessments have been performed, the method 'scale_histograms(QualityAssessor* optimal)' can be called to create a pair of common-scale histograms. The before assessment is known as the 'initial', the after assessment is known as the 'optimal'. The histogram interval for both the initial and optimal assessments must be the same for scale_histograms() to work correctly.

Below is a portion of the previous code modified to show how to create common-scale histograms.

```
    // Set up the quality assessor
  QualityAssessor initial_quality_assessor=QualityAssessor(&cond_no, 10);
  QualityAssessor optimal_quality_assessor=QualityAssessor(&c_ond_no, 10);

    // assess the quality of the initial mesh
  queue1.add_quality_assessor(&initial_quality_assessor, err);
  if (err) return 1;

    // Set the max min method to be the master quality improver
  queue1.set_master_quality_improver(&minmax_method, err);
  if (err) return 1;
\end{verbatim}
    // assess the quality of the final mesh
  queue1.add_quality_assessor(&optimal_quality_assessor, err);
  if (err) return 1;

    // launches optimization on mesh_set1
  queue1.run_instru    // Set up the quality assessor
  QualityAssessor initial_quality_assessor=QualityAssessor(&cond_no, 10);
  QualityAssessor optimal_quality_assessor=QualityAssessor(&cond_no, 10);

    // assess the quality of the initial mesh
  queue1.add_quality_assessor(&initial_quality_assessor, err);
  if (err) return 1;

    // Set the max min method to be the master quality improver
  queue1.set_master_quality_improver(&minmax_method, err);
  if (err) return 1;

    // assess the quality of the final mesh
  queue1.add_quality_assessor(&optimal_quality_assessor, err);
  if (err) return 1;

    // launches optimization on mesh_set1
  queue1.run_instructions(&mesh, err);
  if (err) return 1;

    // create common-scale histograms
  initial_quality_assessor.scale_histograms(&optimal_quality_assessor);
ctions(&mesh, err);
  if (err) return 1;

    // create common-scale histograms
  initial_quality_assessor.scale_histograms(&optimal_quality_assessor);
```

Common-scale Histograms output example

```
************** QualityAssessor(free only) Summary **************

  Evaluating quality for 8 elements.
  This mesh had 8 quadrilateral elements.
  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

          metric     minimum     average         rms     maximum    std.dev.
Condition Number     1.05817     1.14257      1.1469     1.35948   0.0995044
        TSquared        1.18        2.23     2.33533        3.77    0.693433

   TSquared histogram:
   (  1-1.3) |===================3
   (1.3-1.6) |=========================4
   (1.6-1.9) |=========================4
   (1.9-2.2) |=================================================================9
   (2.2-2.5) |============2
   (2.5-2.8) |=========================4
   (2.8-3.1) |======1
   (3.1-3.4) |===================3
   (3.4-3.7) |======1
   (3.7-4  ) |======1
  metric was evaluated 32 times.


************** QualityAssessor(free only) Summary **************

  Evaluating quality for 8 elements.
  This mesh had 8 quadrilateral elements.
  There were no inverted elements detected.
  No entities had undefined values for any computed metric.

metric  minimum     average  rms      maximum  std.dev.
TSquared 1.99733     2  2      2.00266  0.00173023

   TSquared histogram:
   ( 1.997-1.9976) |=================2
   (1.9976-1.9982) |=========================3
   (1.9982-1.9988) |=================================================================6
   (1.9988-1.9994) |=========================3
   (1.9994-2     ) |===================================4
   (     2-2.0006) |0
   (2.0006-2.0012) |=========================3
   (2.0012-2.0018) |=================================================================6
   (2.0018-2.0024) |=========================3
   (2.0024-2.003 ) |=================2
  metric was evaluated 32 times.
```

```
************** Common-scale Histograms **************

   TSquared histogram (initial mesh):
   (  1-1.3) |=====3
   (1.3-1.6) |=======4
   (1.6-1.9) |=======4
   (1.9-2.2) |================9
   (2.2-2.5) |===2
   (2.5-2.8) |=======4
   (2.8-3.1) |=1
   (3.1-3.4) |=====3
   (3.4-3.7) |=1
   (3.7-4  ) |=1
  metric was evaluated 32 times.


   TSquared histogram (optimal mesh):
   (  1-1.3) |0
   (1.3-1.6) |0
   (1.6-1.9) |0
   (1.9-2.2) |============================================================32
   (2.2-2.5) |0
   (2.5-2.8) |0
   (2.8-3.1) |0
   (3.1-3.4) |0
   (3.4-3.7) |0
   (3.7-4  ) |0
  metric was evaluated 32 times.
```

8.2 Debug Output

Mesquite contains a mechanism to send status and debug messages to an output stream (e.g. stdout or std::cout). On Unix-like systems that use a configure/make autotools system debug output is enabled using the "–enable-debug" option on the configure command. This option enables Mesquite's debug capabilities but does not enable any actual debug output messages. Output messages are controlled by flags specified using the "–enable-debug-output" option on the configure command. This two step approach is used so that in release builds the debug output feature can be disabled so that turning on debug flags in a released version has no effect.

Debug messages are grouped into logical categories identified by an integer number. For example, debug flag 1 refers to warnings, debug flag 2 is used for status information about the outer optimization loop, and debug flag 3 is used for status of the inner optimization loop. The command to turn on all three flags would be: "./configure –enable-debug-output=1,2,3". When specifying debug flags using the "–enable-debug-output", the "–enable-debug" flag is implied and need not be supplied. The CMake utility can also be used to enable debug output by setting the "Trillinos_ENABLE_DEBUG" option to "ON". As with the configure command, debug output is only enabled with no flags having been set. CMake options do not support setting of the output message flags so, when configuring Mesquite with CMake, these flags must be specified using the techniques described below.

Debug flags can be controlled through a variety of means. The --enable-debug-output configure option can be specified with a comma-separated list of integer values to specify which debug groups should be enabled by default. An application may call the MsqDebug::enable(unsigned) and MsqDebug::disable(unsigned) functions to enable or disable debug message groups. Debug message groups may also be controlled with the environmental variables MESQUITE_DEBUG and MESQUITE_NO_DEBUG. Each should have a comma-separated list of integer values as its argument. The variables enable and disable, respectively, the corresponding debug message groups.

Additional detail of the available configure command options can be found in Section 2.2.

8.3 Plotting Convergence Behavior

The Mesquite TerminationCriterion class can produce a simple table of tab-separated values for the different Mesquite termination criterion. This file can be used to plot the behavior of the optimization loop using GNU Plot, a spread sheet application, or any other suitable tool. The code listing below illustrates how this feature is activated.

```
// Create global optimizer instance
SteepestDescent improver( &objective_function );
improver.use_global_patch();

// Set only inner termination criterion for
// global optimization
TerminationCriterion inner;
inner.add_absolute_vertex_movement( 1e-3 );
inner.write_iterations( "plot.gpt" );
improver.set_inner_termination_criterion( &inner );

// Run optimization
InstructionQueue queue;
queue.set_master_quality_improver( &improver, err );
queue.run_instructions( &mesh, err );
```

For usable results the feature must be activated on the appropriate TerminationCriterion instance. For a global optimization it should be enabled for the *inner* termination criterion. For other optimization strategies (see Chapter 7) it should be enabled for the *outer* termination criterion.

The following is a sample output file:

#Iter	*CPU*	*ObjFunc*	*GradL2*	*GradInf*	*Movement*	*Inverted*
0	0	1.47419	0	0	0	0
1	0	1.147	0	0	0.657155	0
2	0	1.04779	0	0	0.402173	0
3	0	1.00572	0	0	0.357444	0
4	0	1.00006	0	0	0.150652	0
5	0	1	0	0	0.0153396	0
6	0	1	0	0	0.00015034	0
7	0	1	0	0	6.40008e-09	0

Notice that several of the columns contain only zeros. The column containing the iteration number will always contain valid values. Other values will only be included if they are calculated during the optimization loop. The objective function value will be included for any global optimization that uses an explicit objective function (currently any optimizer other than LaplacianSmoother). In the example source code above we are using the steepest descent solver with a global patch so the objective function value is also included. The other values will only be present if they are calculated for the purpose of checking termination criteria. In the example source code we specify a termination criterion based on vertex movement, so the column labeled "movement" contains the maximum distance any vertex was moved for the corresponding iteration.

Figures 8.1 shows the result of using the above data file with the following GNU Plot commands:

```
set xlabel 'iterations'
set ylabel 'objective function value'
set y2label 'maximum vertex movement'
set y2tics 0.1
plot 'plot.gpt' using 1:3 with linespoints \
     title 'objective function', \
     'plot.gpt' using 1:6 axes x1y2 with \
```

```
linespoints title 'vertex movement'
```

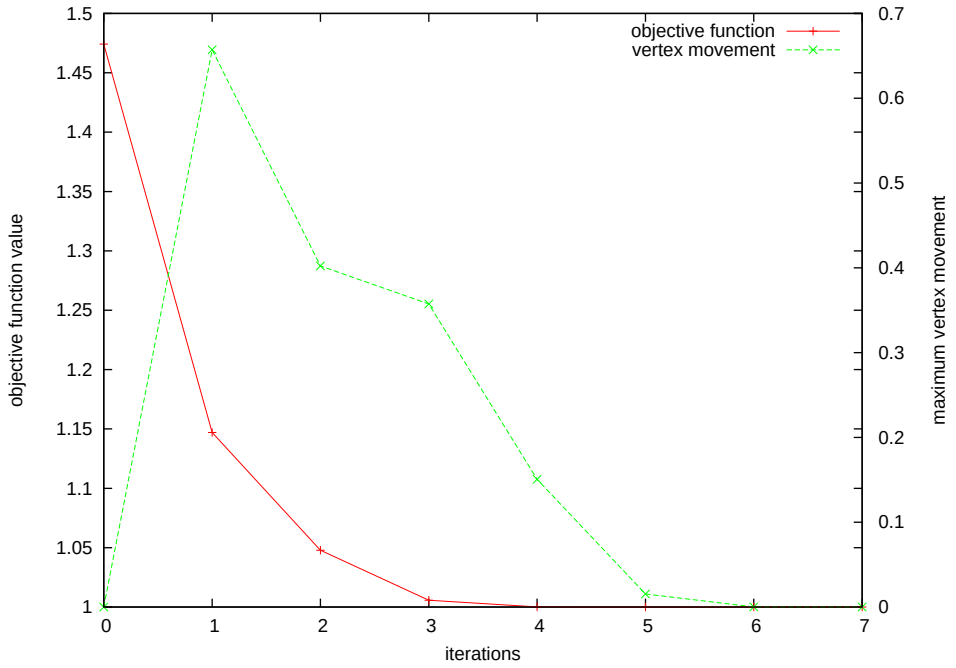

Figure 8.1: *Convergence Plot*

8.4 Viewing Meshes

VTK files read and written by the MeshImpl class are viewable in a plethora of visualization tools that use the VTK visualization library.

The Mesquite::MeshWriter namespace contains functions to export mesh in a variety of formats for visualization including:

- GNU Plot
- Visualization TookKit (VTK)
- Encapsulated PostScript (EPS)
- Scalable Vector Graphics (SVG)
- StereoLithography (STL)

The GNU plot format writes line data that can be used to plot a wireframe of the mesh (the mesh edges). Both 2D and 3D meshes can be exported in this format. A mesh can be plotted as a 2D projection with the GNU plot command:

```
plot 'filename' with linespoints
```

or as a rotatable 3D plot with the command:

```
splot 'filename' with linespoints
```

Figure 8.2 is the result of plotting the mesh contained in testSuite/higher_order/homogeneousPart.vtk with GNU plot.

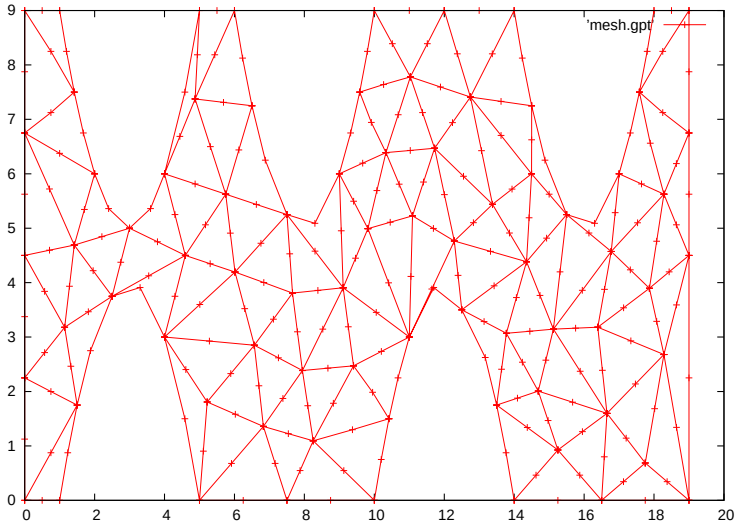

Figure 8.2: *GNU Plot of 2D Quadratic Triangles*

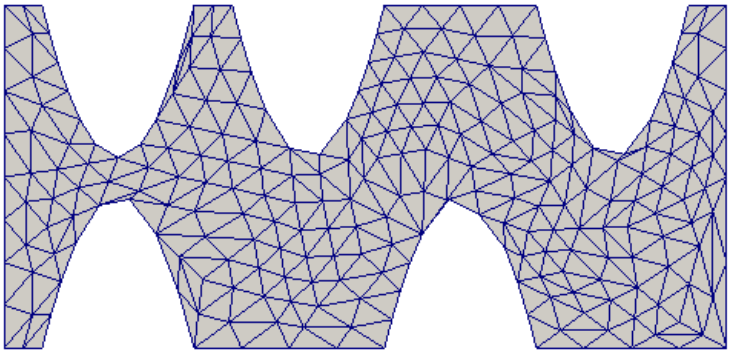

Figure 8.3: *Paraview plot of 2D Quadratic Triangles*

As mentioned in the previous section, the VTK file format can be used with a variety of visualization tools. Figure 8.3 shows a simple plot of the same mesh in the Paraview visualization tool.

Figure 8.4 shows the output of the encapsulated PostScript writer for the mesh. The EPS writer can write only 2D projections of the mesh. The caller must specify a projection when calling `MeshWriter::write_eps`. The `testSuite/higher_order/homogeneousPart.vtk` file contains quadratic triangle elements. Compare the mesh edges on the mesh boundary in this plot with the output in Figures 8.2 and 8.3. The EPS writer in Mesquite exports the quadratic edges as curves corresponding to the classic quadratic edge shape function:

$$E(u) = \frac{1}{2}u(u-1)V_1 + (1-u^2)V_2 + \frac{1}{2}u(u+1)V_3$$

The STL file format can be used to write only linear triangles. Higher-order triangular elements will be written as linear triangles. An error will be returned if the mesh contains other element types.

8.5 Exporting Mesh Quality

The `QualityAssessor` class has the ability to store mesh quality values and other characteristics as tag data on mesh elements. This data can be accessed directly by applications or written to a VTK file using the

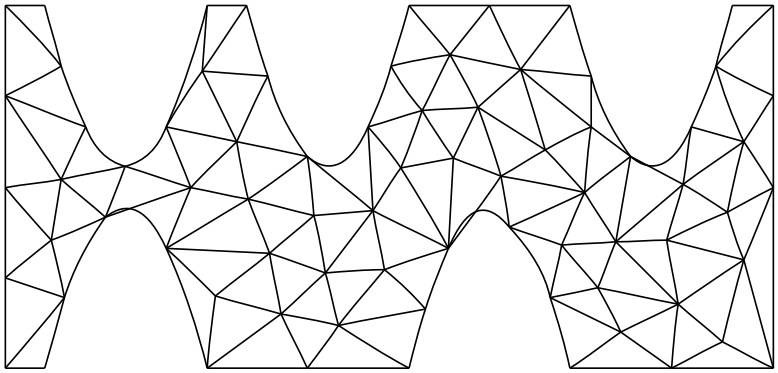

Figure 8.4: *Encapsulated PostScript of 2D Quadratic Triangles*

MeshImpl class or the applications native mesh writer (if it is capable of writing tag data.) The example code below was used to create the VTK file from which the Paraview plot in Figure 8.5 was generated.

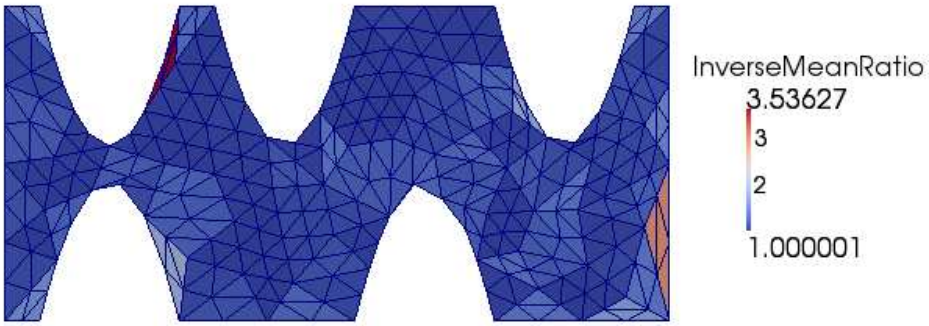

Figure 8.5: *Paraview Plot Coloring Elements by Quality Metric Value*

```
MsqError err;
MeshImpl mesh;
mesh.read_vtk( "homogeneousPart.vtk", err );

IdealWeightInverseMeanRatio metric;
QualityAssessor qa;
qa.add_quality_assessment(&metric,0,0,0,"InverseMeanRatio");

PlanarDomain plane(PlanarDomain::XY);
InstructionQueue queue;
queue.add_quality_assessor( &qa, err );
queue.run_instructions( &mesh, &plane, err );

mesh.write_vtk( "meshqual.vtk", err );
```

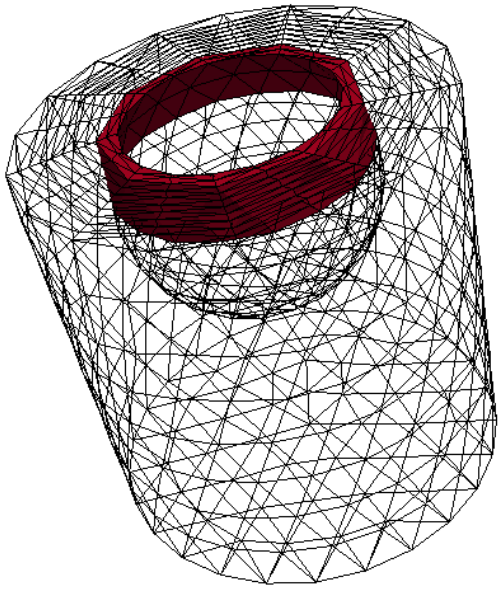

Figure 8.6: *Paraview Plot Showing Inverted Elements*

Figure 8.6 is a Paraview plot showing the inverted elements in a quadratic tetrahedral mesh. The mesh is plotted twice: once as a simple wireframe of the mesh boundary and a second time as solid mesh with a threshold filter on the inverted flag exported by Mesquite. The listing below shows how the QualityAssessor class can be instructed to flag inverted elements:

```
MsqError err;
MeshImpl mesh;
mesh.read_vtk( "sphereCylinder_1194_inv.vtk", err );

QualityAssessor qa;
qa.tag_inverted_elements("Inverted");

InstructionQueue queue;
queue.add_quality_assessor( &qa, err );
queue.run_instructions( &mesh, &plane, err );

mesh.write_vtk( "meshqual.vtk", err );
```

8.6 Mesh Optimization Visualization

The Mesquite TerminationCriterion class can write the complete mesh after each iteration as either VTK or GNU Plot data suitable for viewing as an animation. Similar to requesting plot data as described in Section 8.3, it is important to request this feature from the appropriate termination criterion instance. If doing a global optimization, the feature should be activated for the *inner* termination criterion. Otherwise the feature should almost always be activated for the *outer* termination criterion.

The command to request an animation of the mesh optimization in the VTK format is:

```
tc.write_mesh_steps( "anim", TerminationCriterion::VTK );
```

This will produce a sequence of files named "anim.1.vtk", "anim.2.vtk", etc. The files can be opened in visualization tools such as Paraview as a single set and played back as an animation. If the optimization calculates the gradient of the objective function, that data will also be included in the file as vector data on each mesh vertex. The components of the vector on each vertex are the partial derivatives of the objective function with respect to each coordinate value of the vertex. A Paraview "glyph" filter can be used to display these vector values during the animation.

The command to request an animation of the mesh optimization in a format suitable for animating in GNU plot is:

```
tc.write_mesh_steps( "anim", TerminationCriterion::GNUPLOT );
```

This will produce a sequence of files named "anim.1", "anim.2", etc. It will also export a file named "anim" that contains the necessary GNU Plot commands to display the animation.

Chapter 9

Using Mesquite in Parallel

9.1 Introduction

Large meshes are often partitioned across many parallel processors either because they are too large to fit into the memory of a single machine or in order to speed up the computation. Even if it would be possible to assemble all partitions on a single processor, smooth the mesh, and repartition the result, such an approach would be very I/O inefficient. Moreover, for larger meshes such an approach would quickly run out of memory and fail. Therefore Mesquite supports smoothing meshes in parallel.

Mesquite currently does only synchronous Nash-game or local optimizations in parallel [7]. It does not yet provide parallel solvers and therefore cannot do either block coordinate descent or truly global optimizations in parallel (minimization of an explicit, global objective function.)

For algorithms such as Laplacian smoothing that are local optimizations, optimization in parallel is essentially the same as in serial. For other optimizations that do a global minimization of an explicitly defined objective function in serial (for example `ShapeImprover`), the parallel optimization will be a Nash-game type optimization where the interior vertices (those not on the partition boundaries) will be optimized as a group. Each vertex on the partition boundary will then be optimized individually. While a global optimization in serial will typically have only one outer iteration, it is generally desirable to do multiple outer iterations in parallel so the Nash-game type optimization can reach convergence. Mesquite wrappers (see Chapter 6) that implement global optimizations in serial default to 10 outer iterations in parallel.

9.2 Distributed Mesh

The input mesh for use in parallel quality improvement must be partitioned based on vertices. That is, each vertex in the mesh must be assigned a single processor as its owner. For optimal performance, vertices should be evenly distributed amongst available processors and the vertices assigned to the same processor should compose a contiguously connected patch of mesh.

Each processor must also have access to all elements for which the position of its vertices influence the quality. For almost all algorithms in Mesquite, this is the set of all elements that contain one of the vertices. Further, each processor must also be able to access any additional vertices owned by other processors that are necessary to define those elements. The instances of such vertices on processors that do not own them are typically referred to as “ghosted” vertices. Elements for which copies exist on multiple processors may sometimes also be referred to as “ghosted” or “ghost” elements.

Figure 9.1 shows a mesh partitioned amongst three processors. The vertices owned by the three different processors are shown in three different colors: blue, red, and green. Elements are colored according to the processors for which copies of that element must be available. A copy of an element must be available on each processor owning at least one of the vertices of the element. Elements colored blue, red, or green need be visible only on the processor owning vertices of the corresponding color. The single grey element must have copies defined on all three processors because each of its vertices is owned by a different processor. The remaining elements must be defined on at least two processors.

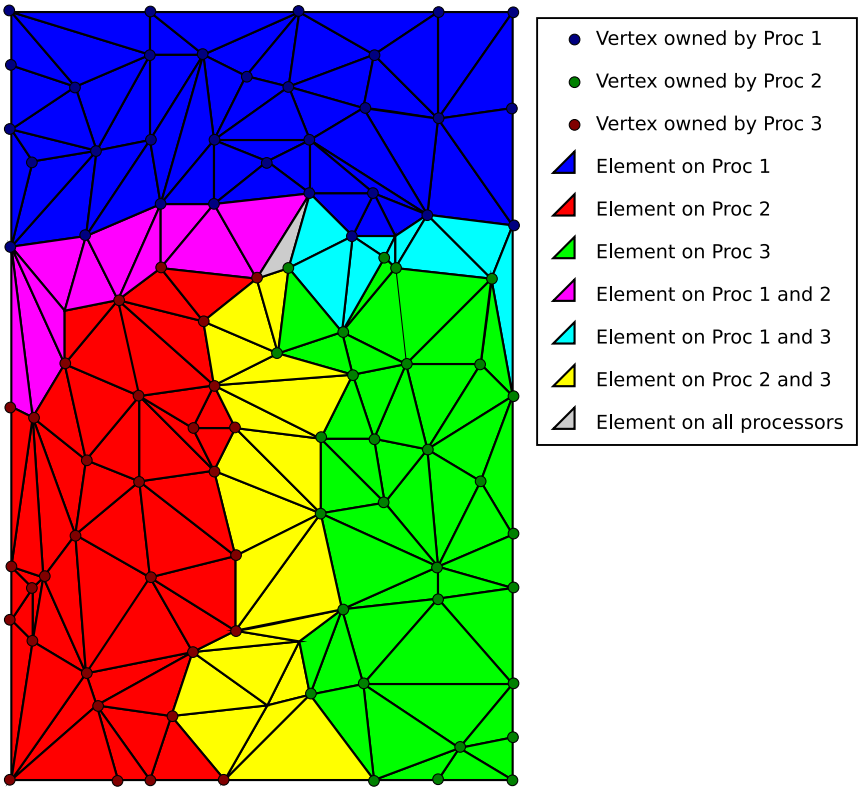

Figure 9.1: Sharing or ghosting of elements and vertices in a partitioned mesh.

For a copy of an element to be available on a processor, all of its vertices must also be available on that processor. So for all elements for which copies exist on more than one processor, the vertices contained in those elements must also exist as ghost vertices on at least one processor. That is, copies of such vertices must exist on processors other than those that are responsible for optimizing the location of that vertex. For example, copies of the yellow elements in Figure 9.1 exist on both the blue and the green processors. All blue vertices in at least one yellow element must exist as ghost vertices on the green processor and all green vertices in at least one yellow element exist as ghost copies on the blue processor. A copy of the grey element must exist on every processor. Therefore each vertex in that element exist as ghost copies on both of the other two processors that do not own it.

9.3 Input Data

Assuming the mesh exists in partitioned form the user has to provide Mesquite with three things:

- a processor ID of type `int` for every vertex that determines which processor owns a vertex and is in charge for smoothing this vertex,
- a global ID of type `size_t` for every vertex that (at least in combination with the processor ID) is globally unique,
- all necessary ghost elements and ghost nodes along the partition boundary must be provided.

The following copies of elements and vertices must exist: Elements must exist on all processors that own one or more of the vertices they reference. Vertices must exist on all processors that have some element referencing them.

The `Mesquite::ParallelMesh` class (`ParallelMeshInterface.hpp`) inherits `Mesquite::Mesh` and defines the interface Mesquite uses to interact with parallel mesh data. It contains the following additional pure virtual (or abstract) functions:

- get processor ids for given vertices,
- get global ids for given vertices,
- set and get a pointer to a `Mesquite::ParallelHelper` object.

To allow Mesquite direct access to the way you store the parallel mesh data you must inherit `Mesquite::ParallelMesh` and also implement your own get processor ID and get global ID functionality. The `Mesquite::ParallelHelper` class takes care of all the underlying communication using MPI. You will always use the `Mesquite::ParallelHelperImpl` implementation that we provide.

Alternatively you can turn any existing mesh of type `Mesquite::Mesh` into a parallel mesh of type `Mesquite::ParallelMesh` by using the `Mesquite::ParallelMeshImpl` implementation we provide. On creation it needs a pointer to an object of type `Mesquite::Mesh` and the names of two tags. It is expected that every vertex is properly tagged with the processor ID tag being of type INT and the global ID tag being of type HANDLE.

9.3.1 ParallelMesh Implementation Requirements

In addition to global and processor ID's, a tag named `LOCAL_ID`, with type `INT`, must be provided in your ParallelMesh implementation. In summary, here are the tags and their types required by Parallel Mesquite:

Concept name	Typical/required code string	Mesquite type
vertex processor owner id	`PROCESSOR_ID` (typical, implementation-dependent)	INT
vertex global unique id	`GLOBAL_ID` (typical, implementation-dependent)	HANDLE
vertex local id (internal use)	`LOCAL_ID` (required)	INT

9.4 ITAPS iMeshP Interface

The MsqIMeshP class is an alternate implementation of the `ParallelMesh` interface that can be used to provide Mesquite with callbacks to access mesh and related parallel properties. The ITAPS Working Group has defined a standard API for exchange of parallel mesh data between applications. The `Mesquite::MsqIMeshP` class declared in `MsqIMeshP.hpp` is an "adaptor": it presents the iMeshP interface as the `Mesquite::ParallelMesh` interface.

This class will use the iMeshP API to query processor identifiers and global identifiers for mesh vertices. However, the MPI-based communication routines implemented in `ParallelHelperImpl` are used rather to communicate updated vertex locations between processors, rather than the mechanism provided by the iMeshP implementation.

9.5 Examples

This section contains two different examples of simple stand-alone applications that demonstrate the use of the `LaplaceWrapper` smoother in parallel. Both examples, in being stand-alone programs, load the mesh from one or more files. When integrating Mesquite into an existing application where it is desired that Mesquite access application mesh data in memory, the initial setup will be different. It will typically involve either providing some application-specific implementation of the `Mesh` and possibly `ParallelMesh` interfaces or instances of an appliction-specific `iMeshP` and `iMesh` implementation.

9.5.1 Example: Parallel Laplacian Smooth

This example uses the `LaplaceWrapper` wrapper in parallel using the built-in `Mesh`, `ParallelMesh`, and `ParallelHelperImpl` implementations. For this example to work, the mesh must be partitioned such that the mesh for each processor is saved in a separate file named `part-%d.vtk`, with the `%d` replaced with the processor rank. Each VTK file must contain vertex attributes named `GID` and `PID` containing the global ID and owning processor rank for each vertex. Further, as this example provides no geometric domain definition, the vertices on the boundary of the mesh must be designated as "fixed" for the problem setup to be valid.

```
/* Mesquite includes */
#include <Mesquite.hpp>
#include <MeshImpl.hpp>
#include <ParallelMeshImpl.hpp>
#include <ParallelHelper.hpp>
#include <MsqError.hpp>
#include <LaplaceWrapper.hpp>

/* other includes */
#include <mpi.h>
#include <iostream>
using namespace std;

int main( int argc, char* argv[] )
{
  /* init MPI */
  int rank, nprocs;
  if (MPI_SUCCESS != MPI_Init(&argc, &argv)) {
    cerr << "MPI_Init failed." << endl;
    return 2;
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  /* create processor-specific file names */
  ostringstream in_name, out_name;
  in_name << "part-" << rank << ".vtk";
  out_name << "part-" << rank << "-smoothed.vtk";

  /* load different mesh files on each processor */
  Mesquite::MsqError err;
  Mesquite::MeshImpl mesh;
  mesh.read_vtk(in_name.str().c_str(), err);
  if (err) {cerr << err << endl; return 1;}

  /* create parallel mesh instance, specifying tags
   * containing parallel data */
  Mesquite::ParallelMeshImpl parallel_mesh(&mesh, "GID", "PID");
  Mesquite::ParallelHelperImpl helper;
  helper.set_communicator(MPI_COMM_WORLD);
  helper.set_parallel_mesh(&parallel_mesh);
  parallel_mesh.set_parallel_helper(&helper);

  /* do Laplacian smooth */
  LaplaceWrapper optimizer;
  optimizer.run_instructions(&parallel_mesh, err);
  if (err) {cerr << err << endl; return 1; }

  /* write mesh */
  mesh.write_vtk(out_name.str().c_str(),err);
  if (err) {cerr << err << endl; return 1;}

  MPI_Finalize();
  return 0;
}
```

Implementation of Example 9.5.1

In your Mesquite distribution, there is an implementation of the example code for Lapalace smoothing in parallel, in the file `mesquite/testSuite/parallel_smooth_laplace/par_hex_smooth_laplace.cpp`. This code reads in a serial or parallel-split set of VTK files and smooths the mesh, then compares the result to a "gold" copy, which is useful for regression testing (see 3.1.6).

Parallel Regression Tests

In addition to the Laplace example, see
`mesquite/testSuite/parallel_untangle_shape/par_hex_untangle_shape.cpp`
for example use of parallel mesh untangling and shape improvement, and the associated files:
`meshFiles/2D,3D/VTK/par_*`

For example, an initial, tangled quadrilateral mesh is shown in 9.2 while the result of untangling and smoothing is shown in 9.3. A similar example with hexahedra is shown in figures 9.4 and 9.5.

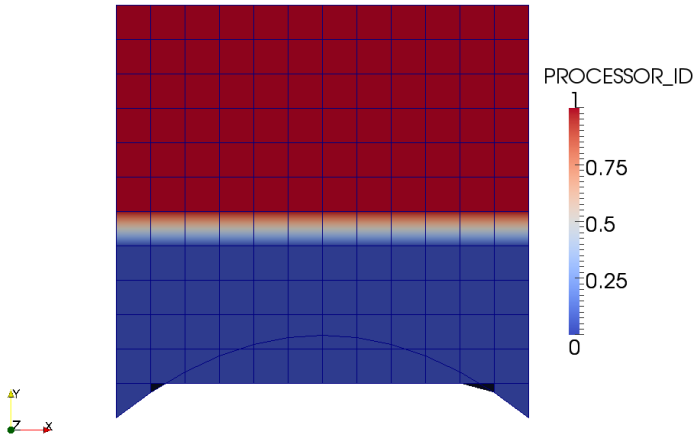

Figure 9.2: Initial, tangled quadrilateral mesh.

9.5.2 Example: Using `Mesquite::Mesquite::MsqIMeshP`

Similar to the example in Section 9.5.1, this example uses the `LaplaceWrapper` wrapper in parallel to improve element shape. However, this example assumes that either the iMeshP implementation is partitioning or that it is reading some pre-defined partitioned mesh and it relies on the iMeshP implementation to create ghost elements, assign global vertex IDs, etc.

An implementation of the iMesh and iMeshP APIs must be provided for this example to work. Mesquite can use these APIs, but does not provide them.

```
/* Mesquite includes */
#include <Mesquite.hpp>
#include <MsqIMeshP.hpp>
#include <ParallelMeshImpl.hpp>
#include <ParallelHelper.hpp>
#include <MsqError.hpp>
#include <LaplaceWrapper.hpp>
```

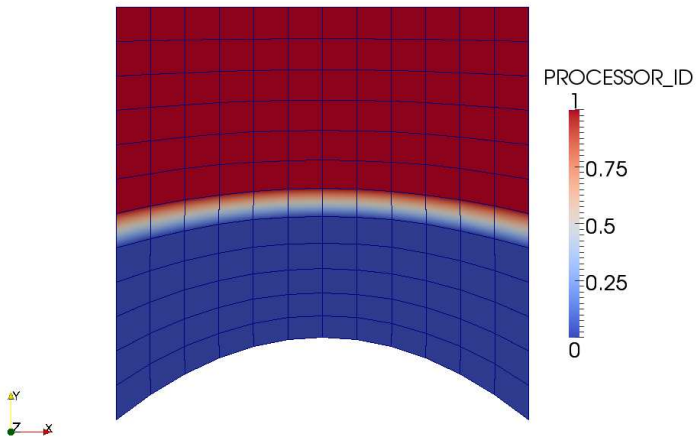

Figure 9.3: Untangled and smoothed quadrilateral mesh.

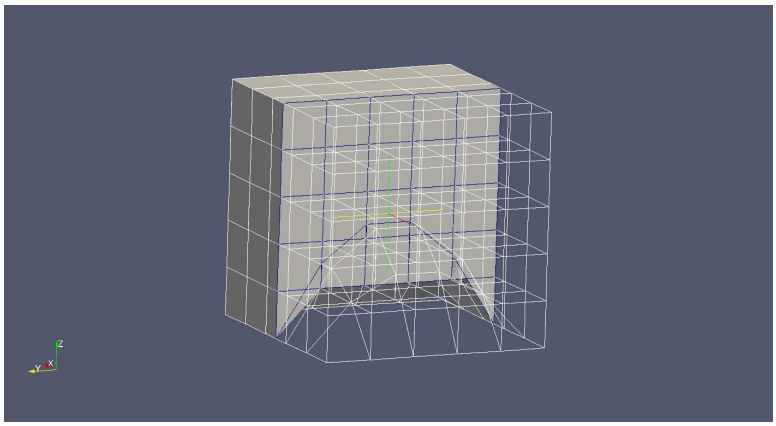

Figure 9.4: Initial, tangled hexahedra mesh.

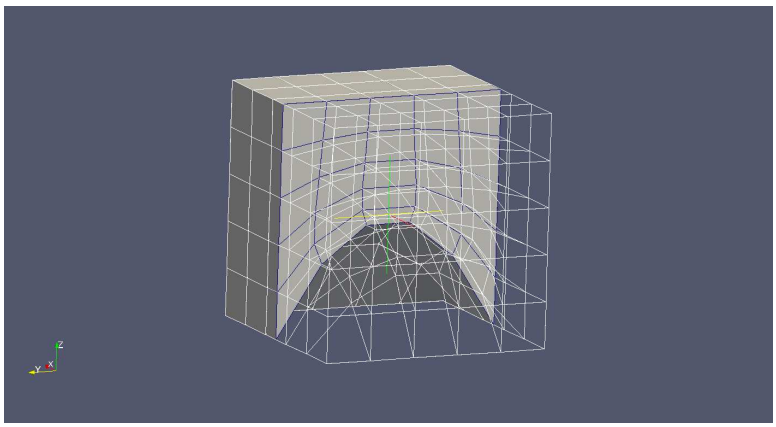

Figure 9.5: Untangled and smoothed hexahedral mesh.

```
/* other includes */
#include <mpi.h>
#include <iostream>
using namespace std;

int main( int argc, char* argv[] )
{
  const char input_file[] = "testmesh";
  const char output_file[] = "smoothmesh";

  /* init MPI */
  int rank, nprocs;
  if (MPI_SUCCESS != MPI_Init(&argc, &argv)) {
    cerr << "MPI_Init failed." << endl;
    return 2;
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  /* create a new instance of the iMesh database */
  int ierr;
  iMesh_Instance mesh;
  iMesh_newMesh(NULL, &mesh, &ierr, 0);
  if (iBase_SUCCESS != ierr) return ierr;
  iBase_EntitySetHandle root_set;
  iMesh_getRootSet(mesh, &root_set, &ierr);
  if (iBase_SUCCESS != ierr) return ierr;

  /* create a partition instance in which to read
     the partitioned mesh */
  iMeshP_PartitionHandle partition;
  iMeshP_createPartitionAll(mesh, MPI_COMM_WORLD,
                            &partition, &err);
  if (iBase_SUCCESS != ierr) return ierr;

  /* load mesh */
  iMeshP_loadAll(mesh, partition, root_set, input_file,
                 NULL, &err, strlen(input_file), 0);
  if (iBase_SUCCESS != ierr) return ierr;

  /* create 1 layer of ghost entities */
  iMeshP_createGhostEntsAll(mesh, partition, 3, 1, 1, 0, &err);
  if (iBase_SUCCESS != ierr) return ierr;

  /* create MsqIMeshP instance */
  Mesquite::MsqError err;
  Mesquite::MsqIMeshP parallel_mesh(mesh, partition, root_set,
                                    iBase_REGION, err);
  if (err) {cerr << err << endl; return 1; }

  /* do Laplacian smooth */
  LaplaceWrapper optimizer;
  optimizer.run_instructions(&parallel_mesh, err);
  if (err) {cerr << err << endl; return 1; }
```

```
  /* write mesh */
  iMeshP_saveAll(mesh, partition, root_set, output_file,
                 NULL, &ierr, strlen(output_file), 0);
  if (iBase_SUCCESS != ierr) return ierr;

  /* cleanup */
  iMeshP_destroyPartitionAll(mesh, partition, &ierr);
  if (iBase_SUCCESS != ierr) return ierr;
  iMesh_dtor(mesh, &ierr);
  if (iBase_SUCCESS != ierr) return ierr;
  MPI_Finalize();
  return 0;
}
```

Chapter 10

User Support

10.1 Mailing Lists

An open mailing for discussion of Mesquite usage questions is available at mesquite@software.sandia.gov. This list is open to all Mesquite users. Archived messages and subscription information are available on the list web page:
http://software.sandia.gov/mailman/listinfo/mesquite.

10.2 WWW Page

The Mesquite WWW page is located at
`http://www.cs.sandia.gov/optimization/knupp/Mesquite.html`.

Appendix A

The Mesquite Team

The Mesquite team is composed of members from Sandia National Laboratories (SNL), Lawrence Livermore National Laboratory (LLNL), and Elemental Technologies Inc. (ETI).

The current Mesquite developement team includes:

Lori Freitag-Diachin (Co-PI, LLNL),

Patrick Knupp (Co-PI, SNL), and

Boyd Tidwell (ETI)

Past developers and other significant contributors to the development of Mesquite include:

Michael Brewer (SNL),

Ulrich Hetmaniuk (SNL),

Jason Kraftcheck (UW).

Thomas Leurent (ANL),

Darryl Melander (SNL), and

Todd Munson (ANL).

Current Mesquite developers can be contacted via e-mail at either the (private) developers' mailing list: Mesquite-Developers@software.sandia.gov, or the (open) mailing list for all Mesquite users: Mesquite@software.sandia.gov.

Appendix B

Acknowledgments

Mesquite is supported under the DOE SciDAC Interoperable Tools and Petascale Simulation (ITAPS) project.

Bibliography

[1] FMDB: Flexible distributed mesh database. http://www.scorec.rpi.edu/FMDB/.

[2] MOAB: A mesh-oriented database. https://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB.

[3] The visualization toolkit. http://public.kitware.com/VTK/.

[4] H. Edelsbrunner and N. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the 8th ACM Symposium on Computational Geometry*, pages 43–52, 1992.

[5] Patrick Knupp Evan van der Zee. Convexity of mesquite optimization metrics using a target-matrix paradigm. Technical Report SAND2006-4975J, Sandia National Laboratories, Albuquerque, NM, 2006.

[6] L. Freitag. Users manual for Opt-MS: Local methods for simplicial mesh smoothing and untangling. Technical Report ANL/MCS-TM-239, Argonne National Laboratory, Chicago, IL, 1999.

[7] L. Freitag, M. T. Jones, and P. E. Plassmann. An efficient parallel algorithm for mesh smoothing. In *Proceedings of the Fourth International Meshing Roundtable*, pages 47–58, Albuquerque, NM, 1995. Sandia National Laboratories.

[8] L. Freitag, P. Knupp, T. Munson, and S. Shontz. A comparison of optimization software for mesh shape-quality improvement problems. In *Proceedings of the 11th International Meshing Roundtable*, pages 29–40, Ithaca, NY, 2002. Sandia National Laboratories.

[9] L. A. Freitag and P. M. Knupp. Tetrahedral element shape optimization via the Jacobian determinant and condition number. In *Proceedings of the 8th International Meshing Roundtable*, pages 247–258, Albuquerque, NM, 1999. Sandia National Laboratories.

[10] P. Hansbo. Generalized laplacian smoothing of unstructured grids. *Comm. Num. Meth. Engr.*, 11:455–464, 1995.

[11] Barry Joe. Three-dimensional triangulations from local transformations. *SIAM Journal on Scientific Computing*, 10:718–741, 1989.

[12] Barry Joe. Construction of three-dimensional improved quality triangulations using local transformations. *SIAM Journal on Scientific Computing*, 16:1292–1307, 1995.

[13] P. Knupp. Achieving finite element mesh quality via optimization of the Jacobian matrix norm and associated quantities. part i - a framework for surface mesh optimization. *Int'l. J. Numer. Meth. Engr.*, 48(3):401–420, 2000.

[14] Patrick Knupp. Formulation of a target-matrix paradigm for mesh optimization. Technical Report SAND2006-2730J, Sandia National Laboratories, Albuquerque, NM, 2006.

[15] Patrick Knupp. Analysis of 2d rotational-invariant non-barrier metrics in the target-matrix paradigm. Technical Report SAND2008-8219P, Sandia National Laboratories, Albuquerque, NM, 2008.

[16] Patrick Knupp. Label-invariant mesh quality metrics. In *Proceedings of the 18th International Meshing Roundtable*, pages 139–155, Salt Lake City, UT, 2009. Springer.

[17] Patrick Knupp. Measuring quality within mesh elements. Technical Report SAND2009-3081J, Sandia National Laboratories, Albuquerque, NM, 2009.

[18] Patrick Knupp. Target-matrix construction algorithms. Technical Report SAND2009-7003P, Sandia National Laboratories, Albuquerque, NM, 2009.

[19] Victor R. Yarberry Larry A. Schoof. Exodusii: A finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Albuquerque, NM, 1994.

[20] et.al. Lori Freitag-Diachin. The itaps imesh interface: Version 0.7 draft. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, 2007.

[21] Patrick Knupp Ulrich Hetmaniuk. Local 2d metrics for mesh optimization in the target-matrix paradigm. Technical Report SAND2006-7382J, Sandia National Laboratories, Albuquerque, NM, 2006.

[22] B. Lorensen W. Schroeder, K. Martin. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics 3rd Edition.* Kitware, Inc., 2003.

[23] A. Winslow. Numerical solution of the quasilinear poisson equations in a nonuniform triangle mesh. *J. Comp. Phys.*, 2:149–172, 1967.