Anatomy of a Mesquite 2.0 Quality Improver

Jason Kraftcheck

October 8, 2007

Contents

List of Figures

1 Introduction

1.1 Intent

After the Mesquite 1.1 release and before the release of Mesquite version 2.0 (approx. 2006/2007 fiscal year), a variety of enhancements were made to the Mesquite solver design. These include:

- Replacing the element-based and vertex-based quality metric dichotomy with the ability to evaluate quality metrics at any local mesh feature.
- The introduction of Jacobian-matrix-based target metrics and the evaluation of those metrics at a variety of sample points within a mesh element.
- Block coordinate descent solvers.
- Initial work towards smoothing of quadratic finite elements.

The intent of this document is to describe the current design and API for the portions of Mesquite affected by the above changes. It is the intent of the author that this document serve both as a description of the design changes for current developers (both Mesquite developers and those developers using Mesquite in other applications) and as documentation for the discussed portions of Mesquite for future developers.

1.2 Organization

A Mesquite quality improver is assembled from instances of the C++ classes representing various components, such as objectve function templates and quality metrics.. Examples of components in Mesquite include the `FeasibleNewton` solver, the `JacobianMetric`, and the `PMeanPTemplate`. This document is organized beginning with the top-most component, the quality improver, followed by the components required by the quality improver, followed by the subcomponents of those components, and so on.

This document also contains several subsections titled *Future Work*. These sections are the musings of the author and should not be interpreted as a formal or official plan for future Mesquite changes.

1.3 UML

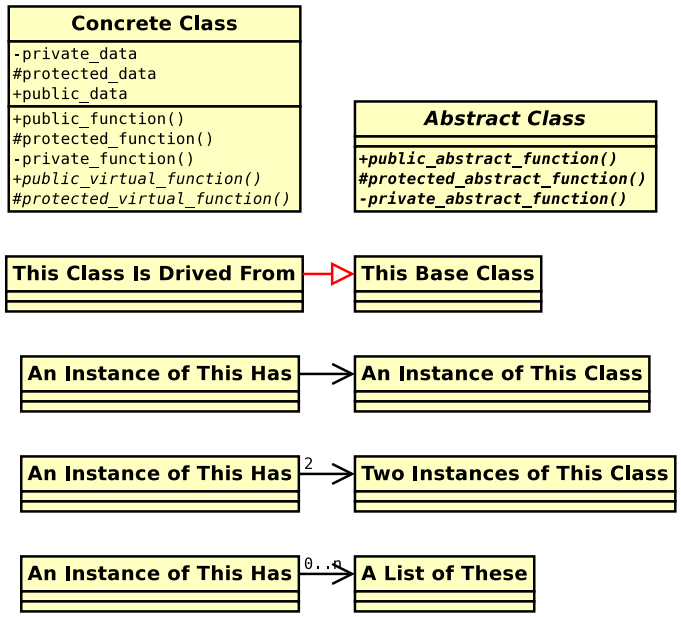

Figure 1: UML class diagram elements and coloring scheme.

UML diagrams are used throughout this document. The UML action diagrams should be self-evident even for those not familiar with UML. However, the UML class diagrams may not be as self-explanatory. Figure 1 shows the subset of UML class diagram notation used and in this document.

The "has-a" relationship, shown with an open-headed arrow in UML diagrams, indicates a loose single-directional association between two objects, typcially implemented by one object having a pointer to another. For Mesquite components, this relation usually indicates an association a developer must create during component construction. Strong associations such as aggregation and composition are not shown in the diagrams in this document. The "derived" or "implements" relationship, shown with an triangle arrow head in UML diagrams, is implemented using class inheritance in C++. The class at the base of the arrow is derived from the class at the head of the arrow. In the context of assembling Mesquite solvers, the base class is often a type of class that may be required by some other component and the derived class is one of several instances of that type that can be used to fulfill the requirement.

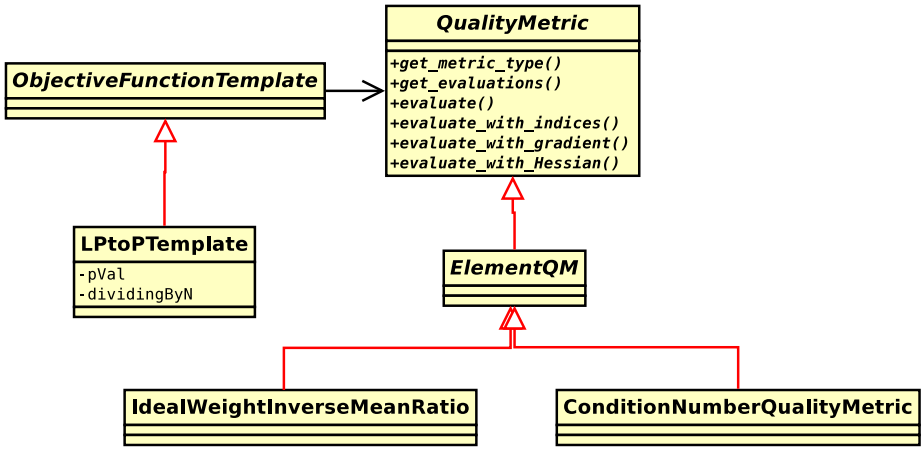

Figure 2: UML Example

Consider the simple example in Figure 2. All implementations of the `ObjectiveFunctionTemplate` abstract class, such as `LPtoPTemplate`, require an instance of a `QualityMetric`. This is indicated by the black arrow from `ObjectiveFunctionTemplate` to `QualityMetric`. To create an instance of `LPtoPTemplate` for use in a solver, one would first have to create a `QualityMetric` instance. However, as indicated by the italics in the class name "QualityMetric" in the diagram, `QualityMetric` is an abstract class or interface. One must create an instance of a class that implements the `QualityMetric` interface. Examples of such classes in Figure 2 are `IdealWeightInverseMeanRatio` and `ConditionNumberQualityMetric`. These are concrete classes (the names are not italicized), so instances of them can be created. The path of red triangle-headed arrows from each of these classes to `QualityMetric` indicate that both classes provide the `QualityMetric` interface. In summary, one can conclude from Figure 2 that a quality metric such as `IdealWeightInverseMeanRatio` or `ConditionNumberQualityMetric` must be created and used in the creation of a `LPtoPTemplate` instance.

1.4 `MsqError`

Almost every function in the Mesquite API accepts an instance of the `MsqError` class and uses that instance to flag the occurance of any errors. For brevity, this argument is not shown or discussed for any function in the API. The reader may assume an implicit final argument of type `MsqError&` for almost every function shown or discussed in this document, the exceptions being those functions that cannot fail.

The `MsqError` class can be treated as a Boolean, where a true state indicates an error. It can also be sent to a C++ output stream to print the error code, error message, and call stack (trace of nested function calls beginning with

the topmost API call down to the function at which the error occured.) An application will typically use an `MsqError` as follows:

```
if (err) {
  std::cout << err << std::endl;
  return FAILURE;
}
```

The `MsqError` class also provides functions to progamatically extract data from such as the error message, error code, and call stack lines.

Mesquite also provides several macros to assist developers in using the `MsqError` class within Mesquite. The `MSQ_SETERR` macro is used to flag an initial error condition. The following examples show typical uses of this macro:

```
  // literal error message and error code
MSQ_SETERR(err)( "My error message", MsqError::UNKNOWN_ERROR );
  // error code and printf-style formatted error message
MSQ_SETERR(err)( MsqError::INVALID_ARG, "Argument 'foo' = %d", foo );
  // error code and default message for that error code
MSQ_SETERR(err)( MsqError::OUT_OF_MEMORY );
```

The `MSQ_CHKERR` macro evalutes to true if an error has been flagged, and false otherwise. Further, if an error has been flagged, it appends the location of the `MSQ_CHKERR` invokation to the call stack maintained in the `MsqError` instance. This is the mechanism by which Mesquite generates the call-stack data. The following is an example of how this macro is typically used:

```
  if (MSQ_CHKERR(err))
    return FAILURE;
```

The macro may also seen used similar to the following example:

```
  return !MSQ_CHKERR(err) && result_bool;
```

This statement will result in a return value of `false` if either an error has been flaged or `result_bool` is false. The order of the tests is important in this example. The `MSQ_CHKERR` macro must occur first so that the call stack is updated, regardless of the value of `result_bool`.

`MSQ_ERRRTN` and `MSQ_ERRZERO` are convenience macros for developers. They are defined as follows:

```
#define MSQ_ERRRTN(ERR)  if (MSQ_CHKERR(ERR)) return
#define MSQ_ERRZERO(ERR) if (MSQ_CHKERR(ERR)) return 0
```

2 Mesh Quality Improvers

2.1 Overview

All quality improvers in the current Mesquite implementation are subclasses of the `VertexMover` class. That is, all quality improvers work by optimizing mesh vertex locations. They do not make changes to the mesh topology.

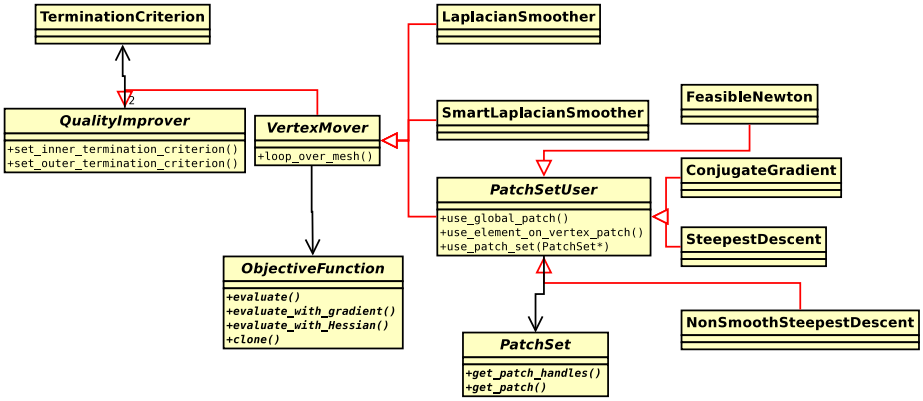

Figure 3: QualityImprover class diagram.

2.2 Patches and the Optimization Loop

2.2.1 Patches

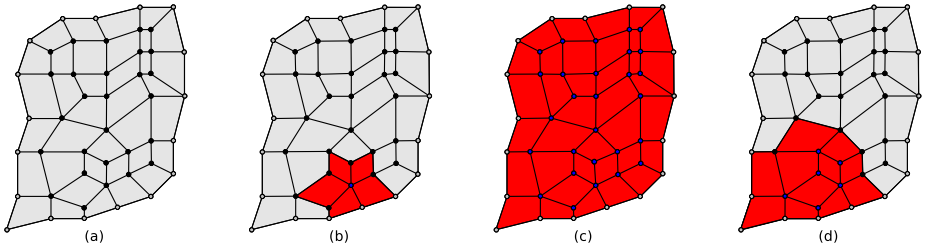

Figure 4: Miscellaneous patch configurations.

Optimization of mesh in Mesquite is performed on *patches*. Given a mesh defined by a set of vertices V and elements E, a patch is composed of an arbitrary $W \subset V$, $F = f \in E, Cl(f) \cup W \neq \emptyset$. That is, a patch is defined to be some subset of the vertices to optimize and the elements for which the quality is affected by the position of these verices. In containing elements, the patch implicitly contains all vertices in the closure of those elements. Any such implicit vertices that are not also in W ($x \in Cl(E), x \notin W$) are referred to as *fixed* (or less accurately as *boundary*) vertices. This property should not be confused with the global *fixed* property of vertices $v \in V$. The latter is an input constraint on the mesh, while the former is a property of the temporary patch data structure, not the actual subset of the mesh represented by the patch. Patches may overlap, may be disjoint, etc.

Vertices are marked as fixed in a patch because the positions of those vertices will not be optimized as part of the optimization of the current patch. Any vertex that is marked as fixed in the input mesh will always be marked as

fixed in any patch it occurs in because the coordinates of such a vertex are by definition never optimized. A vertex may also be marked as fixed in a patch for other reasons, for exaple if it is not in the set of vertices to optimize with the patch but is in the closure of the elements adjacent to said vertices. The *fixed* state of a vertex in a patch is a property of the patch. Such a vertex may not be fixed in other patches, and will not be fixed in at least one other patch unless it is marked as fixed by the application as part of the input mesh. Global patches are the only case where the fixed state in the patch is the same as the fixed state in the input mesh: all non-fixed vertices in the input mesh are not marked as fixed in a global patch.

Figure 4 illustrates different patch determination schemes. The left-most mesh (a) is an input mesh supplied by the application, with the fixed vertices colored in grey and the free vertices in black. A single-vertex patch in the same mesh is shown in (b), with the patch elements in red and the free patch vertex in blue. A global patch is shown in (c) and an appliciation-defined patch in (d).

2.2.2 Optimization Loop

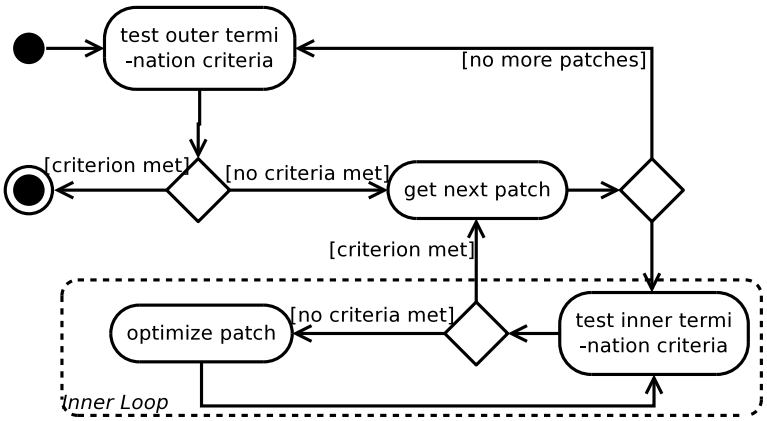

Figure 5: Simplified logical flow of solver.

Optimization is subdivided into an outer and an inner loop. The inner loop is an iterative improvement of the mesh in a patch. The outer loop is an iteration over all of the patches in the mesh. Figure 5 is a simplified UML action diagram for the optimization loop.

For a simple Laplacian smoother, the patch is always a single free vertex and its adjacent elements. The inner loop is a single iteration to update the location of that vertex. The outer loop is a repeated iteration over each vertex in the mesh.

If a patch containing the entire mesh is used, an optimization is said to be a *global* solution. In such a configuration, all optimization is done in the inner

loop. The outer loop may be considered a single iteration over the set of one patch.

2.2.3 Solvers

Many of the quality improvers in Mesquite work by minimizing the value of an objective function (see Section 3), where the objective function is a function of the mesh vertex coordinates. The term *solver* is often used to refer to the portion of the code in the concrete subclass of `VertexMover` that implements the inner loop of the optimization for quality improvers that optimize an explicit objective-function based (the code that implements the function-minimization algorithm.)

The `FeasibleNewton` and `ConjugateGradient` implementations are examples of quality improvers that contain solvers.

2.2.4 PatchSetUser and PatchSet

A `PatchSetUser` is an optimizer for which the application may specify how the mesh is decomposed into patches. `PatchSetUser` provides two pre-defined patch schemes. The `use_global_patch()` method will result in a single patch containing the entire mesh. The `use_element_on_vertex_patch()` method will result in a patch for every free vertex in the mesh, containing only the free vertex and its adjacent elements. An alternate scheme for subdividing the mesh into patches may be specified by providing a custom implementation of the `PatchSet` interface.

The `PatchSet` interface defines two methods: `get_patch_handles` and `get_patch`. The `get_patch_handles` method returns a list of handles (or identifiers), one for each potential patch in a mesh. The `get_patch` method returns the free vertices and elements in a patch, given one of the handles passed back from the `get_patch_handles` method.

The `GlobalPatch` class is the implementation of the `PatchSet` interface that provides a single patch for the entire mesh. The `VertexPatches` class provides the Laplacian-like decomposition of the mesh into a patch for every free vertex. The `ElementPatches` class is used internally in places other than the main optimization loop, such as initializing BCD data and in the `QualityAssessor` class. It decomposes the mesh into single-element patches with no free vertices.

2.2.5 Laplacian Quality Improvers

As shown in Figure 3, the Laplacian quality improvers do not implement the `PatchSetUser` interface, and therefore do not allow the selection of a patch definition scheme. Laplacian smoothers always operate on an element-on-vertex patch with a single free vertex.

2.2.6 Nash Game vs. Block Coordinate Descent

Quality improvers that have an explicit `ObjectiveFunction` may be used with the block coordinate descent algorithm rather than the default Nash game algorithm if the `ObjectiveFunction` implementation supports this mode of operation. In a Nash game optimization (the default), the objective function that is optimized by the inner loop is evaluated over only the patch being optimized. In a block coordinate descent algorithm, the objective function to be optimized in the inner loop is evaluated over the entire mesh. Only the influence of the current patch vertices on the global objective function is considered during the optimization of each patch.

The selection of Nash vs. BCD algorithms is currently done via the constructors of individual `QualityImprover` implementations. In the future this setting will most likely be moved either up to the `VertexMover` class because it doesn't affect the behavior of the code contained in the concrete solvers or down to the individual `ObjectiveFunction` implementations, as the core functionality for BCD is provided by individual `ObjectiveFunction` implementations and it is a characteristic of the individual objective functions that BCD-type algorithms are possible.

2.2.7 What's New

Many of the features described in this section are new to the current version of Mesquite. Mesquite 1.1 was capable of only single-vertex Nash or global optimizations. It supported neither BCD algorithms nor alternate patch-definition schemes.

The `PatchSetUser` and `PatchSet` classes were added after the 1.1 release. In Mesquite 1.1, the API for selecting patch type (single-vertex or global) was provided in the `QualityImprover` interface, allowing nonsensical code such as requesting a global patch for a Laplacian smoother. The logic that is now contained in `VertexPatches` and `GlobalPatch` was interspersed with the `PatchData` code for retreiving the defined patch from the `Mesh` instance.

The new `PatchSet` class also allows for easy reordering of patches during the optimization loop. As the code now works with an enumerated list of patches, that list can be reordered without impacting other code.

2.3 TerminationCriterion

The `TerminationCriterion` class provides representation of various criteria that may be used to terminate either the inner or outer loop of the quality improvement process. A separate instance of the `TerminationCriterion` class is used for each of the inner and outer loops. Outer termination criteria are not useful for an optimization using a global patch. Similarly, inner termination criteria are useless for Laplacian smoothing. The termination criterion used for these cases is an iteration count of one. In the future, Mesquite may be further re-factored to remove these sources of potential confusion from the API.

2.4 Current Issues and Future Work

There is considerable room for improvement in the current organization of quality improvers. Issues with the current design include:

- Laplacian smoothers have a meaningless inner termination criterion. The user may request an inner termination criteria, but that request is silently ignored. The inner termination criterion is always effectively a iteration count of one. However, vertex culling criteria, which are currently associated with the inner `TerminationCriterion` instance, are applicable to Laplacian smoothers.

- Global-patch optimizations have a useless and potentially harmful outer termination criterion. The inner and outer termination criteria are logically equivalent, but iterations in the outer optimization loop are more expensive than inner iterations.

- Laplacian smoothers have an associated ObjectiveFunction instance that is not (necessarily) optimized by the smoother.

- For best performance, the evaluation of an outer criterion based on objective function value should depend on the choice between Nash and BCD. The outer criterion objective function value is assumed to evaluated over the entire active mesh, regardless of the choice between Nash and BCD. The default method for obtaining the value is to evaluate the objective function over the entire mesh. This solution works for both Nash and BCD, but is grossly inefficient for BCD, where the value over the entire mesh is already known.

- The previous point is *not* true for a termination criterion based on the gradient of the objective function. The solver does not know the gradient for the objective function over the entire mesh, only the terms of the gradient that are with respect to free vertices in the current patch.

- For a Nash solution, does it make sense to have an outer termination criterion based on an objective function value for the entire mesh? If so, does it make sense to limit it to the same objective function that is evaluated over the patch in the inner loop? Pat: "Termination criterion for Nash should not include a global OF value, but only the value of the OF on the patch."

- If inner and outer termination criteria are separated, the implementation may be inefficient. However, the different termination criteria groups are logically associated and used by different objects. Laplacian and global optimizations have only one: either inner or outer. Other optimizations have two, but the outer is used by the `VertexMover` class while the inner is used by the solver (e.g. `FeasibleNewton`.)

- The LaplacianSmoother class should be generalized as a RelaxationSmoother, allowing for other relaxation schemes depending on metric values or other criteria.

3 ObjectiveFunction

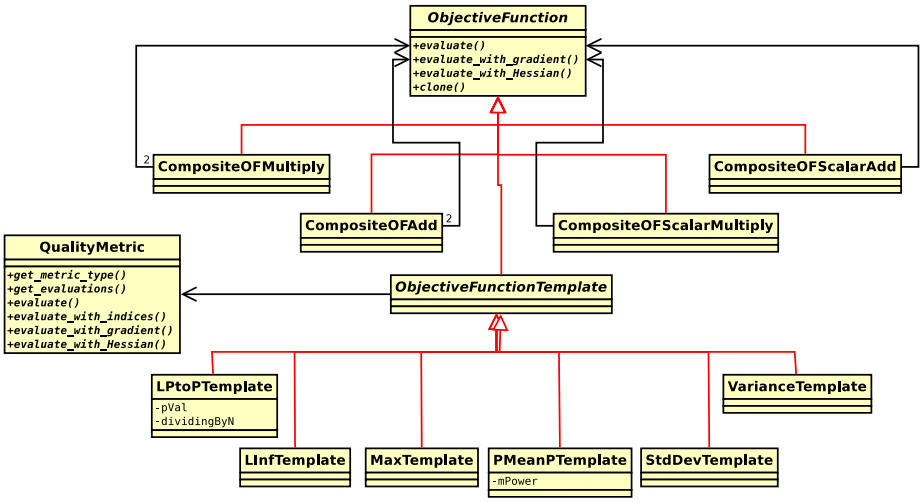

Figure 6: `ObjectiveFunction` class diagram.

An objective function is a scalar function of all the vertex coordinates in the active mesh. The mesh vertex locations are optimized so as to minimize the objective function value.

The objective functions implemented in Mesquite can be divided into two general categories: template objective functions and composite objective functions. Template objective functions have an associated `QualityMetric` instance and typically implement some kind of averaging scheme. Composite objective functions modify the values of one or two other objective functions, such as scaling the value or summing two values.

Most solvers in Mesquite also require the gradient of the objective function (the partial derivatives of the objective function with respect to the coordinate values of the free vertices in the patch.)

Some quality improvers (currently `FeasibleNewton`) need to know the Hessian (second partial derivatives) of the objective function. Only an `ObjectiveFunction` implementation capable of providing Hessian data may be used with such a solver. Mesquite provides numerical approximation of objective function gradient values, and of quality metric gradient and Hessian values, but not objective function Hessian values. Further, Mesquite is capable of working only with Hessians of objective functions that have sparse Hessian matrices. The

Hessian matrix may only contain non-zero terms for vertex pairs that share at least one element. This limitation is explicit in the implementation of the `MsqHessian` class, and is implicit in other areas of the code (such as portions of the `ObjectiveFunction` interface relating to use in a BCD algorithm.) Some `ObjectiveFunction` implementations such as `CompositeOFMultiply` (the product of two other objective function values) and `StdDevTemplate` have a dense Hessian and therefore cannot be used with solvers requiring a Hessian.

3.1 Objective Function Support for Block Coordinate Descent

The `enum EvalType` was added to allow `ObjectiveFunction` implementations to support block coordinate descent algorithms efficiently. Many objective function templates perform averaging of quality metric values. These objective functions can be calculated as a running total for the working mesh. This running total cane be updated for changes to the local patch without needing to recalculate quality metric values for other parts of the mesh. Consider the example of an objective function that is the algebraic mean of an element quality metric $\mu(v \in Cl(e_i))$:

$$M = \{e_1 \cdots e_n\}$$

$$f(M) = \frac{1}{n} \sum_{i=1}^{n} \mu(v \in Cl(e_i))$$

which can be rewritten as:

$$f(M) = \frac{1}{n} g(M)$$

$$g(M) = \sum_{i=1}^{n} \mu(v \in Cl(e_i))$$

The values of n and $g(M_0)$ can be evaluated by making a single pass over all the elements of the mesh before the optimization begins. The value of n remains constant for the duration of the optimization (Mesquite does not modify the mesh topology). Given some patch of the mesh: $P \subset M$, $g(M)$ can be updated for changes P with the rather simple arithmetic:

$$g(M) = g(M_0) - g(P_0) + g(P)$$

That is, the value of $g(P_0)$ is calculated at the beginning of the inner optimizer loop. The updated value of g (and therefore f) can then be calculated for a modified version of the patch by calculating $g(P)$ only.

3.1.1 Objective Function Derivatives for Block Coordinate Descent

`ObjectiveFunction` implementations supporting BCD algorithms are also expected to provide gradient and possibly Hessian data. In a BCD algorithm, the implementation is expected to supply only the terms of the gradient that are

with respect to the free vertices in the current patch. Similarly, only the Hessian terms that are with respect to only free vertices in the current patch need be provided.

Given that a patch is guaranteed to contain all elements adjacent to the free vertices (in any context where the gradient or Hessian is required), it is assumed that any objective function capable of meeting the sparseness requirement for the Hessian described in Section 3 will be able to provide gradient and Hessian values for a patch in a BCD context without needing to know any data other than the contents of the current patch and the scalar objective function value for the entire mesh.

Continuing with the algebraic mean example from above:

$$\nabla f = \frac{1}{n}\nabla g$$

$$\mathcal{H} f = \frac{1}{n}\mathcal{H} g$$

$$\nabla g = \sum_{i=1}^{n} \nabla \mu(v \in Cl(e_i))$$

$$\mathcal{H} g = \sum_{i=1}^{n} \mathcal{H} \mu(v \in Cl(e_i))$$

∇g is simply the sum of the n quality metric gradient vectors, one for each element in the mesh. As the quality metric is, by definition of an element metric, a function only of the vertices of one element the terms of the quality metric gradient containing partial derivatives with respect to the coordinates of any vertex not in the element must be zero. Therefore the terms of ∇g that correspond to the partial derivatives of any vertex v_i are zero for any quality metric evaluation over an element not adjacent to v_i. Conversely, the portion of ∇g corresponding to v_i is a function of only the quality of the elements adjacent to v_i. As the local patch is guaranteed to contain all elements adjacent to v_i if the gradient terms for v_i are requested, it is possible to compute the portion of ∇G corresponding to the free vertices in the patch without any data other than the patch contents. The same simple reasoning applies equally for the Hessian of the algebraic mean template.

3.2 Objective Function Implementations

The `ObjectiveFunctionTemplate` is a base class for those objective function implementations that are some kind of average of quality metric values. The `ObjectiveFunctionTemplate` class provides the API for associating a `QualityMetric` with an objective function and it provides a common implementation for initializing coordinate descent optimizations.

Existing template-type objective functions from version 1.1 of Mesquite have been updated for the new interface. These include: `LPtoPTemplate`,

LInfTemplate, and MaxTemplate. Of these three, support for coordinate descent optimizations was added to LPtoPTempalte only.

The new objective functions PMeanPTemplate, StdDevTemplate, and VarianceTemplate have also been added. All three of these objective function templates support block coordinate descent optimization.

The composite objective functions modify one or two existing objective functions (e.g. adding two together). All composite ObjectiveFunctions support analytical gradients and coordinate descent optimization if the underlying ObjectiveFunctions do. Similarly, all except CompositeOFMultiply support analytical Hessians as long as their underlying ObjectiveFunctions do. CompositeOFMultiply does not have a suitably sparse Hessian (see Section 3).

3.3 What's New

Since Mesquite 1.1, the ObjectiveFunction interface has been re-written to support block coordinate descent optimization. All ObjectiveFunction implementations have been updated for the interface change and all except min/max-type objective function implementations have been updated to support block coordinate descent optimization. Appendix B contains more detail on the interface-level changes to the ObjectveFunction class.

Mesquite 1.1 did not support analytical gradient or Hessian optimization in composite objective functions. This missing functionality has been added. All ObjectiveFunction implementations have also been updated for changes to the QualityMetric interface (see below.)

The PMeanPTemplate, StdDevTemplate and VarianceTemplate classes have been added subsequent to the Mesquite 1.1 release.

4 QualityMetric

The QualityMetric class provides a measure of the quality at a local mesh feature, such as an element or a vertex. Figure 7 is the class diagram for all quality metrics (except the legacy DFT metrics discussed in Section 6) in Mesquite. The basic interface for the QualityMetric class is shown in Figure 24.

4.1 Averaging Metric Values

There are two classes in Mesquite defining interfaces for averaging quality metric values: AveragingQM and PMeanPMetric. These classes are typically used for averaging corner or sample-based quality metric values at a vertex or over an element. Neither is a subclass of QualityMetric. Both are expected to be used as a base class for quality metric implementations that inherit the QualityMetric interface separately. These utility classes provide the user API for selecting the averaging scheme. They also provide utility methods for averaging met-

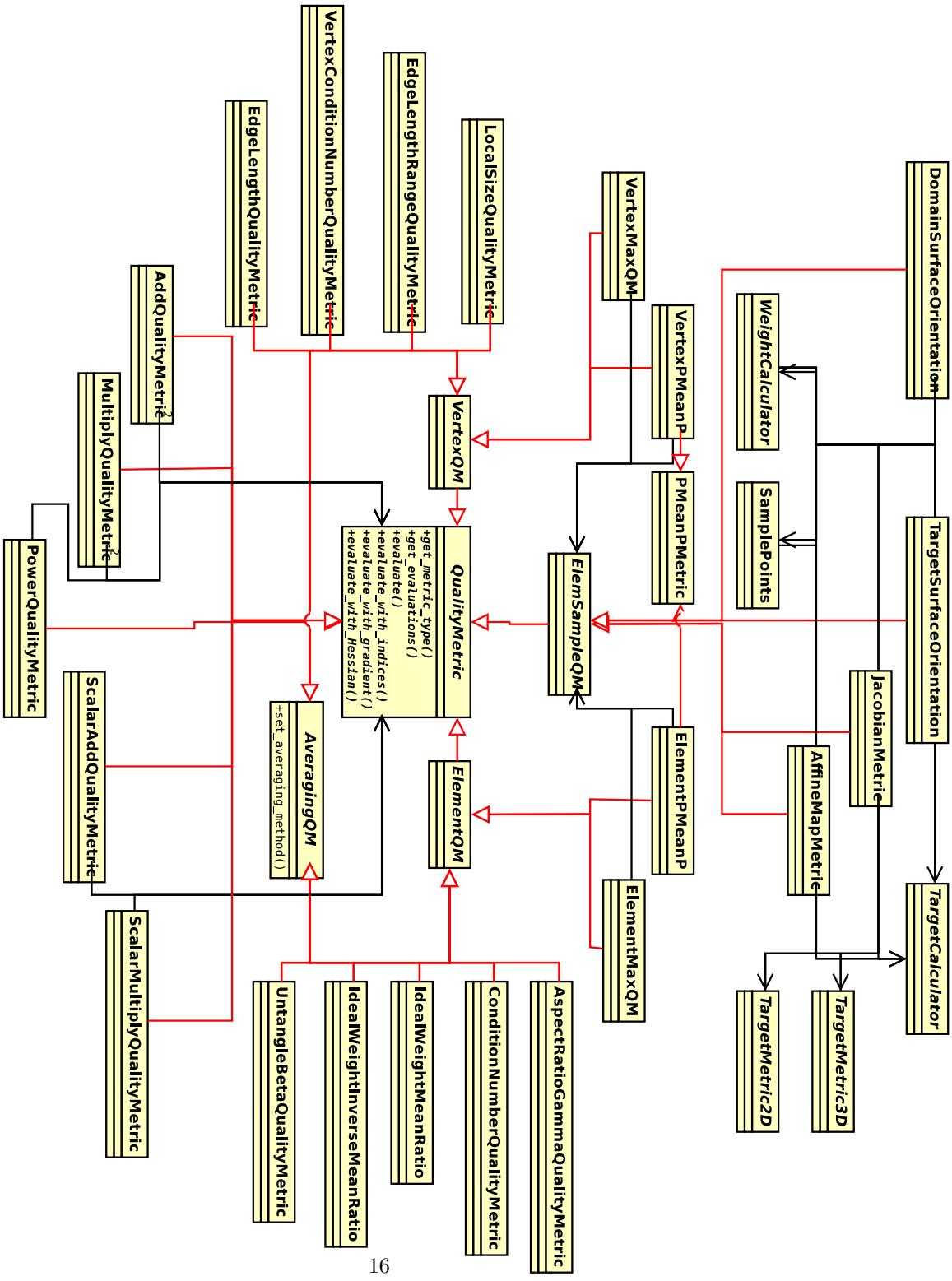

Figure 7: `QualityMetric` Classes.

ric values, gradients, and Hessians for use by the subclass implementing the `QualityMetric` interface.

The `AveragingQM` class provides the same enumeration of averaging schemes that was defined in `QualityMetric` in Mesquite 1.1. The user selects from a list of averaging schemes such as `LINEAR`, `RMS`, `SUM_SQUARED`, `MAXIMUM`, `MAX_MINUS_MIN`, etc. Many metrics that are internally corner-based for non-simplex elements implement this interface, converting such cases to element-based metrics. Most element-based quality metrics from Mesquite 1.1 are examples of this type of metric. Some metrics that are internally corner-based or edge-based implement this interface to provide a vertex-based metric by averaging the values around a vertex. Most of the vertex-based quality metrics from Mesquite 1.1 are examples of this type of metric. See the `QualityMetric` class diagram (Figure 7) for a list of metrics implementing this interface.

The `PMeanPMetric` accepts a single parameter: the P-value. The result is a power-mean of a set of values raised to the P power: $\frac{1}{n}\sum_{i=0}^{n-1}\mu_i^p$. As shown in Figure 7, this class serves as a base class for two subclasses: `ElementPMeanP` and `VertexPMeanP`. These classes require an instance of a concrete sample-based quality metric to function. They work as a type of composite metric, converting the sample-based metric to an element-based or vertex-based metric, respectively, by averaging the metric values for the appropriate sample points.

The `ElementMaxQM` and `VertexMaxQM` are similar to the `ElementPMeanP` and `VertexPMeanP` in that they convert sample-based metrics to element-based or vertex-based metrics. Unlike the `*PMeanP` metrics, these metrics return the maximum metric value for the entity.

4.2 Common Evaluation Points

The three classes: `ElementQM`, `VertexQM` and `ElemSampleQM` provide common functionality for quality metrics evaluated at the respective entity types. `ElemSampleQM` also extends the `QualityMetric` with functions to provide additional information required by classes such as `ElementPMeanP` (see Section 4.1). It is not necessary for a metric of a conceptual type to be a subclass of any of these existing classes (with the exception of the case where one wishes to use a sample-based metric with one of the averaging classes described in Section 4.1). The `ElementQM` and `VertexQM` classes provide utility methods for generating evaluation lists and other functionality that is common for all subclasses.

4.3 Composite QualityMetrics

The set of `QualityMetric` implementations that provide a modification of the output of one or more other `QualityMetric` implementations are referred to as *composite* quality metrics. Examples of composite metrics include:

`ScalarAddQualityMetric`: Offset the value of a metric by a constant amount.

`ScalarMultiplyQualityMetric`: Scale the value of a metric by a constant amount.

`PowerQualityMetric`: Raise the value of a metric to a constant power.

`AddQualityMetric`: Add two quality metric values together.

`MultiplyQualityMetric`: Multiply the values of two quality metrics.

The composite metrics that operate on multiple underling `QualityMetric` instances (`AddQualityMetric` and `MultiplyQualityMetric`) require metrics that evaluate on the same type of entity. For example, if one metric is vertex-based, the other should be also. Otherwise the code has no mechanism to determine which pairs of metric values should be combined. This is typically enforced by requiring the `get_metric_type` method return the same value and that the `get_evaluations` method returns the same list of values for the same patch (see Appendix C for an explanation of the `QualityMetric` interface.) The mechanism for encoding evaluation handles for sample-based metrics was designed such that it is possible to combine an element-based metric with a sample-based metric (Section 4.4 if the sample-based metric is evaluated at only one sample point in each element. Sample-based metrics may be combined in a composite metric if they evaluate elements at the same set of sample points.

4.4 Sample-Based Metrics

A sample-based metric is evaluated at one or more logical sample points within each element of the mesh. The sample-based metrics currently provided in Mesquite are `JacobianMetric`, `AffineMapMetric`, `DomainSurfaceOrientation`, `TargetSurfaceOrientation`, and the legacy DFT metrics (See Section 6). `AffineMapMetric` exists primarily for research purposes and will not be discussed further.

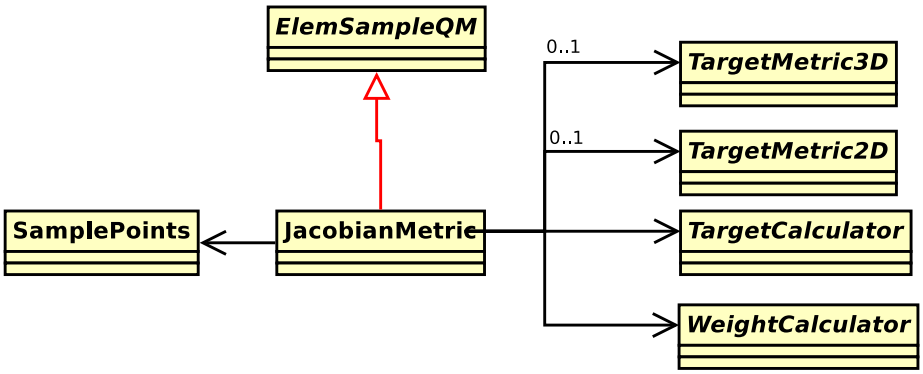

Figure 8: `JacobianMetric` Class Diagram.

As shown in Figure 8, `JacobianMetric` class combines the functionality of a `SamplePoints` instance, a `TargetMetric2D` and/or a `TargetMetric3D`, a `TargetCalculator` and a `WeightCalculator` for an `ElemSampleQM` (which is a

QualityMetric). This diagram is a subset of the larger QualityMetric class diagram (Figure 7.).

The TargetMetric2D and TargetMetric3D classes are metrics that compare a pair of 2x2 or 3x3 matrices, respectively. For surface elements (e.g. triangles or quadrilaterals), the JacobianMetric class constructs a 2x2 matrix by rotating the 3x2 Jacobian of the element into the plane of the target 3x2 matrix. The TargetCalulator instance provides the 3x3 and/or 3x2 matrices that the matrix metrics compare to the element Jacobian. The WeightCalculator provides a scalar weight at each sample point that is multiplied with the value of the target metric to produce the quality metric.

The rotation from 3x2 Jacobian to the 2x2 matrix passed to the TargetMetric2D in JacobianMetric discards 3D orientation data for surface elements. The DomainSurfaceOrientation and TargetSurfaceOrientation classes provide a means to evaluate the relative 3D orientation of surface elements. They can be combined with a JacobianMetric using a composite quality metric. The DomainSurfaceOrientation metric compares the local element normal (as provided by the Jacobian) with the normal of the geometric domain at the point closest to the location of the element sample point. The TargetSurfaceOrientation compares the normals provided by the 3x2 Jacobian matrix and the 3x2 target matrix.

4.4.1 SamplePoints

The SamplePoints class provides a mechanism for an application to specify where in each element the JacobianMetric should be evaluated. The application may specify any combination of logical sample points: corners, mid-edge, mid-face, and mid-element. These values may be specified independently for each element topology (triangle, quadrilateral, etc.)

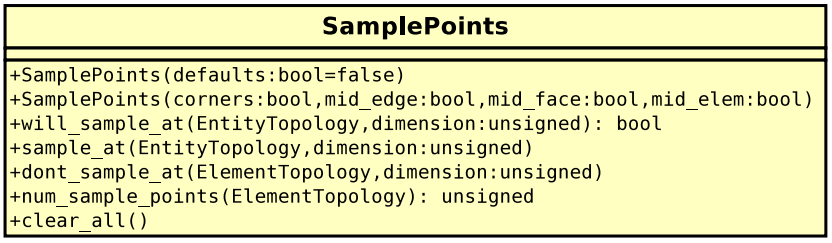

Figure 9: SamplePoints Class.

The interface for the SamplePoints class is shown in Figure 9. The first constuctor, by default, initializes an empty SamplePoints instance (no sample points selected for any element topology.) If that constructor is passed a value of true, then it will initialize with the defaults of one sample point at the center of each simplex and sample points at the corners for other element topologies.

The second constructor can be used to select a common set of logical sample points for all element topologies.

The `sample_at` method adds a logical set of points to the sample points for a specific element topology. The corresponding `dont_sample_at` method removes a set of logical sample points. The `will_sample_at` can be used to test if a given set of logical sample points will be evaluated for a specific element topology. Each of these methods accepts two arguments. The first indicates the element topology (triangle, hexahedron, etc.) The second argument, `dimension`, specifies a logical set of sample points by the portion of the element topology that they lie on: 0 for corners, 1 for edges, 2 for mid-face, and 3 for mid-element.

In Figure 8, it can be seen that the `SamplePoints` instance is associated with the subclasses of `ElemSampleQM`. It may at first seem that it would be more logical to associate the `SamplePoints` instance with the `ElemSampleQM` because all such metrics are evaluated at sample points. The reason the design is as shown is that the `SamplePoints` class provides a mechanism for the application to customize which sample points a metric is evaluated at. Some metrics that are evaluated at element sample points may only allow a pre-defined set of element sample locations for each element topology. For example, the legacy DFT metrics (not show, see Section 6) always evaluate at element corners. As another example; it may eventually be desirable to implement many of the existing element-based metrics as sample-based metrics. These metrics have one evaluation for simplicies and are evaluated at the corners for other element topologies. In both examples, the metrics are sample-based but neither allow the application to select which sample points.

4.5 Mapping Functions

A `MappingFunctionSet` must also be specified for a Jacobian-based metric to work. The `MappingFunctionSet` instance is not associated with any particular metric because a) it is assumed that if there are multiple Jacobian-based metrics in use that they will share a common `MappingFunctionSet` and b) for the planned handling of higher-order nodes a single definitive `MappingFunctionSet` must be available to determine the placement of *slave* higher-order nodes. An instance of a `MappingFunctionSet` must be passed to the top-level `InstructionQueue::run_instructions` method along with the `Mesh` and `MeshDomain` instances.

A `MappingFunctionSet` is a group of `MappingFunction` instances, one for each element topology. Each `MappingFunction` provides methods to query the value and derivatives of the mapping function. Figure 10 shows the interfaces for the `MappingFunction` and `MappingFunctionSet` classes.

A mapping function is assumed to be of the form:

$$\vec{x}(\vec{\xi}) = \sum_{i=1}^{n} N_i(\vec{\xi})\vec{x_i}$$

where $\vec{x_i}$ is a point in $\mathbf{R}^3$ (i.e. x_i, y_i, z_i), $\vec{\xi_i} = \left\{ \begin{array}{c} \xi_i \\ \eta_i \end{array} \right\}$ for surface elements and

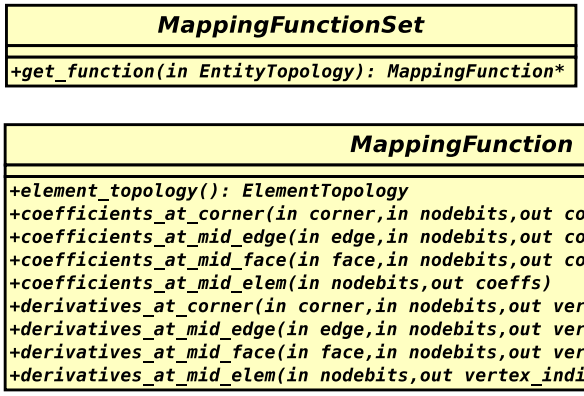

Figure 10: `MappingFunction` interface.

$\vec{\xi_i} = \left\{ \begin{array}{c} \xi_i \\ \eta_i \\ \zeta_i \end{array} \right\}$ for volume elements. For example, a linear quadrilateral element will have a mapping function of the form:

$$\vec{x}(\xi,\eta) = N_1(\xi,\eta)\vec{x_1} + N_2(\xi,\eta)\vec{x_2} + N_3(\xi,\eta)\vec{x_3} + N_4(\xi,\eta)\vec{x_4}$$

A single implementation of the mapping function interface may support multiple elements types of the same topology (e.g. both a 8-node hex and a 20-node hex.) The `nodebits` argument to each evaluation function in the interface is passed to specify which higher-order nodes are present in the element.

The `coefficients_at_*` methods return the values of the mapping function coefficients evaluated at different parameter-space coordinates (the value of the N_i terms for different pre-defined values of $\vec{\xi}$.) These methods are assumed to pass back a coefficient for each vertex in the element, in the canonical order.

The `derivatives_at_*` methods return the partial derivatives of the mapping function coefficients with respect to the parameters. These values can be used to construct the Jacobian of the mapping function and the derivative of the Jacobian with respect to the vertex coordinates. See the Doxygen comments for the `MappingFunction` class for more details.

Mesquite currently provides only one `MappingFunctionSet` implementation with a complete set of `MappingFunction` implementations (one for each element topology): `LinearFunctionSet`. The `LinearFunctionSet` provide shape functions constructed for linear elements (elements without higher-order nodes.) Mesquite also contains the `QuadLagrangeShape`, `TetLagrangeShape` and `TriLagrangeShape` classes. These classes provide shape functions constructed from Lagrange polynomials. They support any combination of higher-order nodes (e.g. a quad with a mid-edge node on only one edge.) This support for degenerate higher-order elements is provided with the intention of allowing *slave* higher-order nodes in elements. A *slave* node's location is defined by the mapping function, as op-

posed to defining the mapping function. It therefore cannot be included in the evaluation of the mapping function.

4.6 What's New

The `QualityMetric` interface has been generalized to allow metric types other than vertex-based and element-based. All existing quality metrics have been converted to this interface. This change substantially reduced duplicate code in composite metrics, numerical derivative approximation, objective function templates, etc.

Mesquite 1.1 had the functionality now provided by `AveragingQM` as part of the `QualityMetric` interface. This was moved into a separate interface so that only the metrics that support this API provide it. The work on consolidating the code for averaging metric values, gradients, and Hessians that was begun before Mesquite 1.1 was completed as a part of this change. In the future, it may make sense to transition to a scheme similar to the subclasses of `PMeanPMetric`, where the existing `AveragingQM` present themselves as corner-based metrics and a separate class is used to average the metric values. This would eliminate the need for two separate implementations of the condition number metric (`ConditionNumberQualityMetric` and `VertexConditionNumberQualityMetric`).

Complete implementations of analytical gradient and Hessian calculations have been added for all composite metrics.

The `JacobianMetric` class and all supporting functionality (`MappingFunction`, `MappingFunctionSet`, `TargetMetric2D`, `TargetMetric3D`, `TargetCalculator`, `WeightCalculator`, `SamplePoints`, etc.) is completely new. These new capabilities are discussed further in the next section.

4.7 Future Work

`VertexPMeanP` (Section 4.1) currently averages metric values at corners regardless of the sample points (Section 4.4) that the metric is to be evaluated at. This should be corrected such that any sample points that are influenced by the coordinates of the vertex are included in the average.

Some performance gain could most likely be achieved by specializing the current simple implementation of the `VertexPMeanP` metric for more common P values, similar to the `PMeanPTempalte` (Section 3.2.) The entire sample-based metric implementation is in a preliminary state. Large portions of this functionality remain unimplemented:

- There are no mapping functions for higher-order elements with hexahedral, pyramidal, or prism topologies.
- The majority of the 3D target metrics have yet to be implemented.
- None of the target metrics support analytical gradient or Hessian calculations.

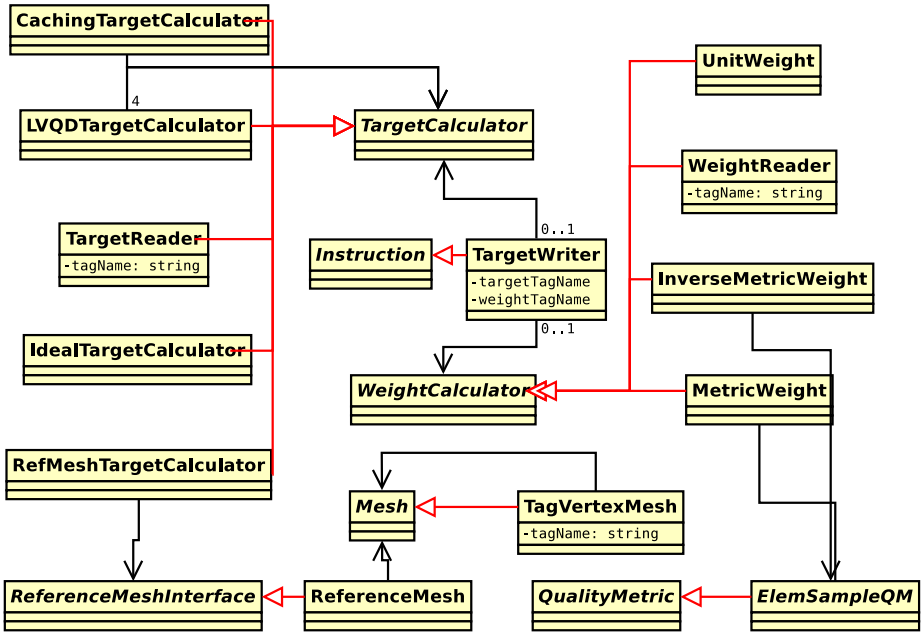

Figure 11: Target-Related and Weight-Related Classes

- The `JacobianMetric` class does not support analytical gradient or Hessian calculations.
- There is no provision for dealing with the derivatives of targets or weights that are a function of position.
- No profiling or subsequent optimization has been done for this code.

5 Target and Weight Calculators

Figure 11 is a UML diagram of the classes discussed in this section. Referring to this figure and Figure 7 may help in understanding how classes discussed in this section fit into the larger Mesquite solver API.

5.1 Target Calculators

A target matrix (W) is compared to an active matrix by a target metric. The active matrix is typically the Jacobian of the element mapping function at a sample point within the element. For example, the `JacobianMetric` class compares the Jacobian of the element to a target matrix using a `TargetMetric3D` or `TargetMetric2D` instance. See Section 4.4 for more detail about target metrics and sample-based quality metrics.

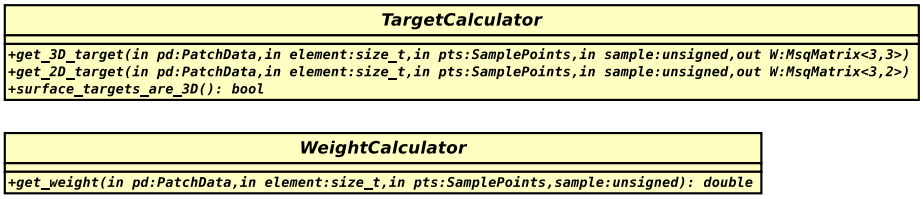

Figure 12: `TargetCalculator` and `WeightCalculator` Interfaces.

The `TargetCalculator` class defines an interface for obtaining target matrices. The `TargetCalculator` is shown in Figure 12 The two primary functions are `get_target_2D` and `get_target_3D`. Both functions are passed sufficient data to identify a specific sample point on a specific element. Both functions pass back a target matrix. For `get_target_2D`, a 3x2 matrix is returned. This matrix is compared with the 3x2 Jacobian matrix for surface elements. The `get_target_3D` function returns a 3x3 target for comparison with the Jacobian of volume elements.

The `surface_targets_are_3D` method in `TargetCalculator` allows for backwards compatibility with legacy DFT target calculators (see Section 6.) A return value of `true` indicates a legacy DFT target calculator, which generates 3x3 targets for surface elements. Many of the utility classes discussed in this section may also be used with legacy DFT code.

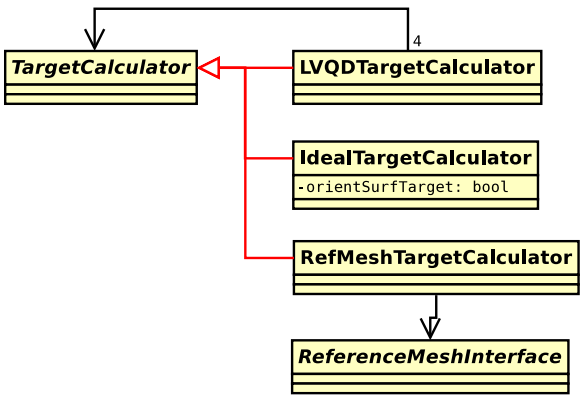

Figure 13: `TargetCalculator` Implementations.

Figure 13 shows the three true target calculators implemented in Mesquite as of this writing (as opposed to utility classes that also implement the `TargetCalculator` interface.) Reference meshes and the `RefMeshTargetCalculator` are discussed in Section 5.3.

The `IdealTargetCalculator` returns the Jacobian of an ideal element at the

sample point corresponding to the logical sample location in the active element. The default behavior for surface elements is to rotate the ideal element such that the normal of the ideal and active elements are aligned at the logical sample point before calculating the Jacobian of the ideal element. This behavior can be disabled, resulting in surface targets for ideal elements in the xy-plane.

The `LVQDTargetCalculator` combines four other target calculators using an LVQD factorization:

$$\mathbf{W} = \lambda(\mathbf{W}_\lambda) \times \mathbf{V}(\mathbf{W}_\mathbf{V}) \times \mathbf{Q}(\mathbf{W}_\mathbf{Q}) \times \mathbf{\Delta}(\mathbf{W}_\mathbf{\Delta})$$

The λ function returns a the scalar size component of the $\mathbf{W}_\lambda$ term. The $\mathbf{V}$ function returns the orientation component of the $\mathbf{W}_\mathbf{V}$ term. $\mathbf{Q}(\mathbf{W}_\mathbf{Q})$ is the shape component of $\mathbf{W}_\mathbf{Q}$. The $\mathbf{\Delta}$ function returns the aspect ratio component of $\mathbf{W}_\mathbf{\Delta}$.

5.2 Weight Calculators

A sample weight is a scalar weight that is multiplied with the result of a `TargetMetric` or other sample-based quality metric evaluation. The `WeightCalculator` class defines an interface for obtaining weights for each sample point in each element over which the metric is evaluated. The single method defined by `WeightCalculator` (see Figure 12) is `get_weight`. The `get_weight` method accepts sufficient parameters to identify an element and a sample point in that element. It returns the corresponding weight. See Section 4.4 for more detail about how sample weights are used in the Mesquite API.

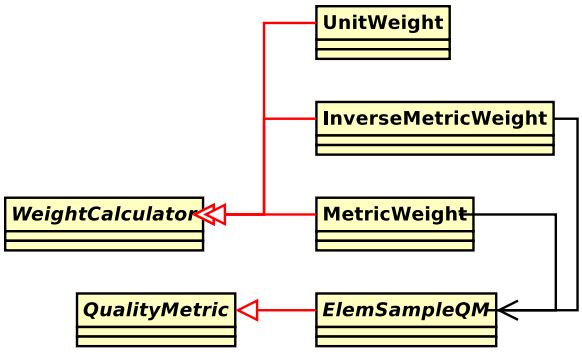

Figure 14: `WeightCalculator` Implementations.

Figure 14 shows the three concrete `WeightCalculator` implementations available at the time of this writing. The `UnitWeight` is a trivial `WeightCalculator` returning a constant unit value for all sample locations in all elements. The `MetricWeight` and `InverseMetricWeight` return the value or inverse value, respectively, of a sample-based quality metric value the corresponding location.

5.3 ReferenceMesh

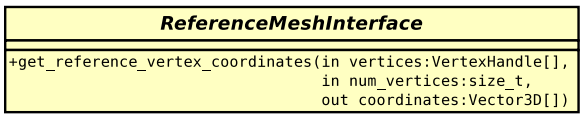

Figure 15: `ReferenceMesh Interface`

A reference mesh is another mechanism for generating targets for a `JacobianMetric`. A reference mesh is a mesh that is topologically identical to the active mesh, differing only in the location of the vertices. The target for any element in the active mesh is then the Jacobian of the corresponding element in the reference mesh. The `RefMeshTargetCalculator` class provides the implementation of this target calculation scheme.

`RefMeshTargetCalculator` uses a class implementing the interface shown in Figure 15. The single function defined by `ReferenceMeshInterface` accepts a list of handles for vertices in the *active* mesh and is expected to return the coordinates of the corresponding vertices in the reference mesh.

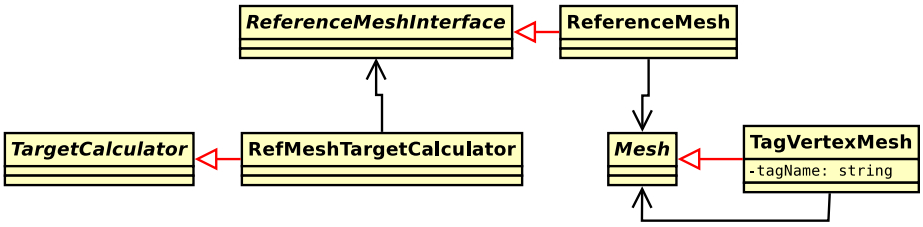

Figure 16: Reference Mesh Implementations.

Figure 16 shows the only current implementation of `ReferenceMeshInterface`: `ReferenceMesh`. The `ReferenceMesh` class accepts a pointer to a second `Mesh` instance (the active mesh being the first instance). It assumes that vertex handles in the active `Mesh` instance are `identical` to those in the reference `Mesh` instance. This assumption is unlikely to be true in general, however it is true for pairs of VTK files read by `MeshImpl` for testing purposes and for the `TagVertexMesh` scenario discussed below. A more general implementation of `ReferenceMeshInterface` would require the use of an external API such as the iTaps iRel interface or some application-specific tag convention to provide the mapping between vertices in the active and reference meshes.

The `TagVertexMesh` class shown in Figure 16 can be used to provide more compact and robust reference mesh data for deforming mesh problems. The `TagVertexMesh` class is a general purpose tool that is also useful outside of the context of target calculation (e.g. storing temporary optimization results.) It

provides the `Mesh` interface using a reference to another instance of the `Mesh` interface (decorator pattern.) All calls to the `Mesh` functions as implemented by the `TagVertexMesh` class are passed through to the corresponding functions in the underlying `Mesh` instance, except for those that query or set vertex coordinates. All vertex coordinate queries are redirected to a tag in the underlying mesh. This mechanism allows multiple coordinate tripples to be associated with each vertex in the mesh: the true coordinates and any number of alternate values stored in different tags.

To use the `TagVertexMesh` class to define a reference mesh in a deforming mesh scenario, the first step is to use the utility methods in the class to copy the vertex coordinates in the undeformed mesh to a tag. After the mesh has been deformed, it then contains the deformed vertex coordinates as the primary coordinates and the original, undeformed coordinates as data stored in a tag. When optimizing such a mesh, the unadorned `Mesh` instance is used as the active mesh and the same `Mesh` instance, wrapped inside a `TagVertexMesh`, is used as the reference mesh.

5.4 Caching Targets

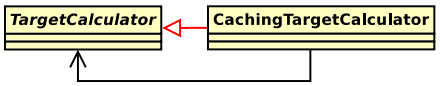

Figure 17: `CachingTargetCalculator`

The `CachingTargetCalculator` class shown in Figure 17 is a mechanism to cache target matrices on the current `PatchData`. It can be used to speed up optimizations when the target matrices are constant for the duration of the optimization.[1] Caching of targets is enabled by inserting an instance of the `CachingTargetCalculator` between the user of the true `TargetCalculator` (`JacobianMetric` in Figure 18) and the actual `TargetCalculator` instance.

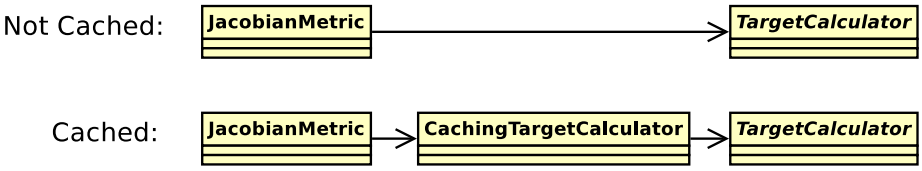

Figure 18: Caching Targets

[1] The target matrices need only be constant for the duration of the current patch (one iteration of the outer optimization loop). However, it is unlikely that a practical situation will arise where this condition is met and the targets are not constant for the duration of the optimization.

5.5 Pre-Calculating Targets and Weights

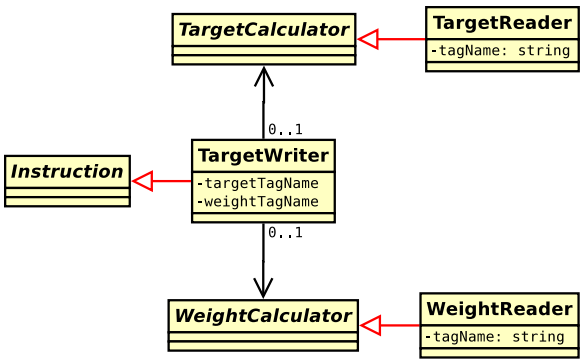

Figure 19: Classes for Pre-computing Targets and Weights.

The classes shown in Figure 19 provide a mechanism for pre-calculating targets and/or weights and storing them as tag data on a mesh. The targets are stored by associating either or both a `TargetCalculator` and a `WeightCalculator` with an instance of the `TargetWriter` class. The `TargetWriter` class implements the `Instruction` interface, and therefore is expected to be inserted directly in the `InstructionQueue`.

The targets can be calculated at the beginning of an optimization by placing the `TargetWriter` first in the `InstructionQueue` instance. Alternately, the targets can be calculated and stored for later use by running an instruction queue containing only the `TargetWriter` instance (and saving the modified mesh to a file if desired.)

The `TargetReader` and `WeightReader` classes implement the `TargetCalculator` and `WeightCalculator` interfaces, respectively. They are associated with the `JacobianMetric` or other code using the resulting data in place of the `TargetCalculator` and `WeightCalculator` instances used by the `TargetWriter`. They read the tag data stored by the `TargetWriter` instance.

5.6 What's New

While the majority of the capabilities described in this section were present in Mesquite 1.1, the implementation was substantially re-factored to allow for the following:

- Multiple `JacobianMetrics` active in a single optimization (e.g. composite metrics or composite objective functions), with different target calculators used by each metric.
- Optional caching of target matrices to allow for targets that vary during the optimization while retaining the performance when targets do not.

- LVQD factorization from targets other than identity and reference mesh Jacobians.
- 3x2 target matrices for surface elements.
- The future incorporation of targets generated from data other than a reference mesh or ideal elements.

5.7 Current Issues and Future Work

The converse of the `TargetWriter` class (or some other mechanism) should be provided for freeing the tag data generated by `TargetWriter`. There is no current mechanism for doing this, and by necessity `TargetWriter` spreads the data over multiple tags making it difficult for the application to remove the tag data manually.

The `LVQDTargetCalculator` implementation should be optimized to eliminate calculating target matrices multiple times when the same target calculator is used for more than one of the LVQD terms.

A general capability to generate target matrices by interpolating on a background mesh should be implemented. Further, as the targets generated by such a target calculator vary by spatial position rather than a logical relation, the API must be extended to account for this in the calculation of analytical gradient and Hessian data for the metric.

As all of the current sample-based metrics use a `WeightCalculator` instance, and as the instance used is in practice often `UnitWeight`, it may make sense to separate this functionality into a separate composite-type metric. This new composite metric would accept a sample-based quality metric and a weight calculator, and multiply the metric value at each sample point by the corresponding weight. This is similar in nature to the `ScalarMultiplyQualityMetric`, differing only in the fact that the scalar that each metric value is multiplied with is not necessarily the same.

Such a composite metric for incorporating the weight into the metric value would have several advantages. The need for `UnitWeight` would be eliminated. If no weighting is desired, one need simply not use the composite metric that applies the weight. The need to query the weight separately in multiple metrics that are combined in a composite (e.g. a `JacobianMetric` and a `TargetSurfaceOrientation`) is no longer necessary. The weight could be applied after the metrics are combined.

6 Legacy DFT Metrics

The experimental DFT (distance from target) metrics from Mesquite 1.1 are still present and have been updated to work within the framework of the current design. The metrics have been updated to present an `ElemSampleQM` interface, allowing them to be used with averaging metrics such as `ElementPMeanP`.

The target calculators for the legacy DFT metrics have also been updated to work within the current framework. They can be used with all of the utility classes such as target caching, pre-calculated targets, etc. Further, the `LVQDTargetCalculator` class can be used with either the new or old target calculators. The concrete target calculators for the DFT metrics have been renamed such that all class names contain the string "Corner" to distinguish them from the newer target calculators.

The multitude of concisely named target calculator subclasses defined in `ConcreteTargetCalculators.hpp` remain legacy target calculators. These should not be used with the newer Jacobian-based metrics.

7 Constructing Quality Improvers

This section walks through two example `QualityImprover` constructions, demonstrating the relationship between the UML diagrams shown in this document and the coding of actual optimizers using the Mesquite API.

7.1 Constructing a Quality Improver

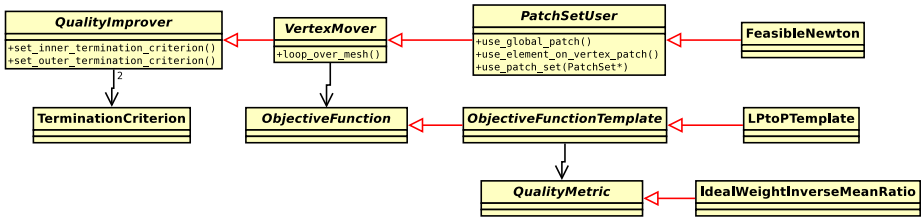

Figure 20: Example Quality Improver

Figure 20 diagrams an example `QualityImprover`. A black, one-headed arrow indicates that the application developer must provide an instance of the pointed-to class to create an instance of the class from which the arrow originates. Classes with italicized names are abstract classes (interfaces) that cannot be created directly. To create an instance of such a class, the application developer must create an instance of a concrete subclass . In the above diagram, a concrete subclass is one with a non-italicized name from which a chain of closed-headed red arrows can be traced to the desired abstract class.

In the example in Figure 20, the objective is to create an instance of the `FeasibleNewton` quality improver. The indirect base class `QualityImprover` requires two `TerminationCrierion` instances. The indirect base class `VertexMover` requires a `ObjectiveFunction` instance. These must be created in order to construct the optimizer. The `TerminationCriterion` class is concrete, so this dependency can be addressed directly.

The `ObjectiveFunction` class is abstract, so it is necessary to find a concrete subclass to instantiate. The only such class shown in Figure 20 is `LPtoPTemplate`. However, the `ObjectiveFunctionTemplate` base class of `LPtoPTemplate` requires a `QualityMetric`, another abstract class. So a concrete subclass of `QualityMetric` must be found. The only such class shown is `IdealWeightInverseMeanRatio`.

Listing 1 shows the source code required to construct the solver depicted in Figure 20. The `IdealWeightInverseMeanRatio` instance is created first at line 1. It is then used at line 2 to create a `LPtoPTemplate` instance. Next the two `TerminationCriterion` instances are created (line 5), and initialized with the desired criteria (lines 6-11). The `LPtoPTemplate` instance is used to create an instance of `FeasibleNewton` at line 13. At lines 14 and 15 the two `TerminationCriterion` instances are attached to the `FeasibleNewton` instance. Finally, at line 17 the `FeasibleNewton` sovler is instructed to use global patches with a call to the `use_global_patch` function provided by the `PatchSetUser` base class.

Listing 1: Example Quality Improver

```
1   IdealWeightInverseMeanRatio metric;
2   LPtoPTemplate obj_func( 2, &metric );
3
4   MsqError err;
5   TerminationCriterion inner_term, outer_term;
6   inner_term.add_criterion_type_with_double(
7  TerminationCriterion::VERTEX_MOVEMENT_ABSOLUTE, 0.01, err );
8   inner_term.add_criterion_type_with_int(
9  TerminationCriterion::CPU_TIME, 60, err );
10  outer_term.add_criterion_type_with_int(
11 TerminationCriterion::NUMBER_OF_ITERATES, 1, err );
12
13  FeasibleNewton solver( &obj_func );
14  solver.set_inner_termination_criterion( &inner_term );
15  solver.set_outer_termination_criterion( &outer_term );
16
17  solver.use_global_patch();
```

7.2 Constructing A Jacobian-Based Metric

Figure 21 shows an example quality improver that optimizes a jacobian-based metric. As seen in the figure, the application must provide instances of the `TargetCalculator`, `WeightCalculator`, and `SamplePoints` classes, as well as an instance of one or both of `TargetMetric2D` and `TargetMetric3D`. In the example in the figure, a `TargetMetric3D` is not provided. This means that the resulting `JacobianMetric` will fail if the mesh contains volume elements. `SamplePoints` is a concrete class, and can therefore be created directly. The `UnitWeight` class is used to satisfy the `WeightCalculator` requirement. The `Target2DShape` class is the `TargetMetric2D` that will be used by the `JacobianMetric`.

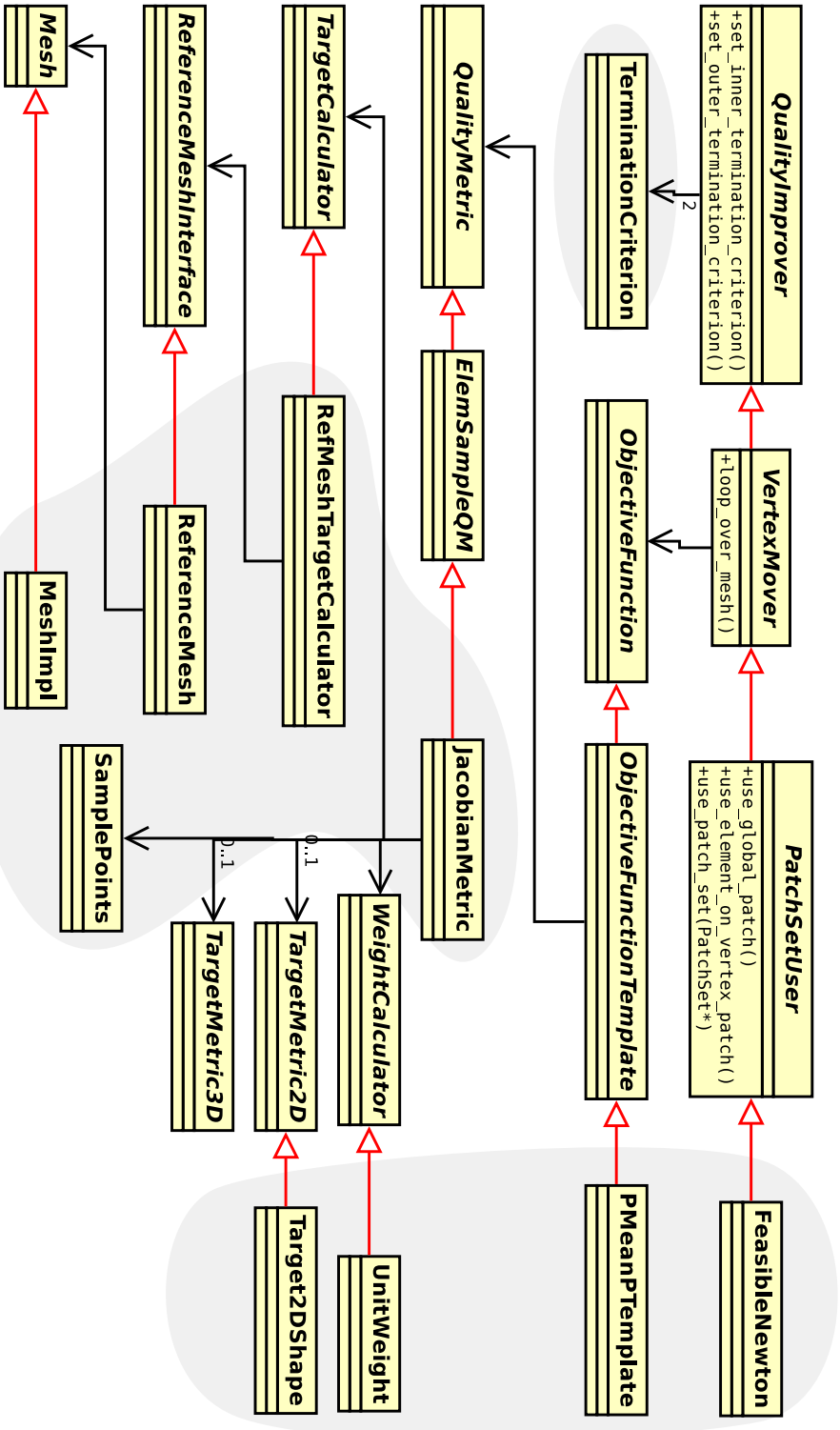

Figure 21: Example Jacobian-based Optimizer

The `TargetCalculator` used is the `RefMeshCalculator`, which requires the creation of a `ReferenceMesh` instance, which in turn requires a `Mesh` instance. Once the `JacobianMetric` has been constructed, an optimizer can be constructed by creating instances of the following: `PMeanPTemplate`, `TerminationCriterion`, and `FeasibleNewton`.

Listing 2 contains the code to create the solver depicted in Figure 21. The `SamplePoints` instance is created first. The `true` value passed to the constructor signals that it should be initialized with the default set of sample points. The 2D target metric instance is created at line 2 and the weight calculator is created at line 3. At lines 5-12 a mesh is read into a `MeshImpl` instance to serve as the reference mesh for the target calculator. The `RefMeshTargetCalculator` is created at line 15 using an instance of the `ReferenceMesh` class created on the previous line. The `JacobianMetric` is created from the `SamplePoints`, `Target2DShape`, `UnitWeight` and `RefMeshTargetCalculator` instances. The final argument to the `JacobianMetric` constuctor is `NULL` as no 3D target metric is provided. The remainder of the quality improver construction (lines 23-37) is analogous to the one described in Section 7.1. The last step (line 39) is to declare a `MappingFunctionSet` instance. This is not used during solver construction, but it must be passed to `InstructionQueue::run` for the `JacobianMetric` to function.

Listing 2: Example Quality Improver with Jacobian-Based Metric

```
1  SamplePoints samples(true);
2  Target2DShape metric_2d;
3  UnitWeight weight;
4
5  MsqError err;
6  MeshImpl ref_mesh_data;
7  ref_mesh_data.read_vtk( "refmesh.vtk", err );
8  if (err) {
9    std::cerr << "Error reading file: \"refmesh.vtk\""
10             << std::endl << err << std::endl;
11   exit( 1 );
12 }
13
14 ReferenceMesh ref_mesh( &ref_mesh_data );
15 RefMeshTargetCalculator target( &ref_mesh );
16
17 JacobianMetric metric( &samples,
18                        &target,
19                        &weight,
20                        &metric_2d,
21                        NULL );
22
23 PMeanPTemplate obj_func( 2.0, &metric );
24
25 TerminationCriterion inner_term, outer_term;
26 inner_term.add_criterion_type_with_double(
```

```
27  TerminationCriterion::VERTEX_MOVEMENT_ABSOLUTE, 0.01, err );
28    inner_term.add_criterion_type_with_int(
29  TerminationCriterion::CPU_TIME, 60, err );
30    outer_term.add_criterion_type_with_int(
31  TerminationCriterion::NUMBER_OF_ITERATES, 1, err );
32
33    FeasibleNewton solver( &obj_func );
34    solver.set_inner_termination_criterion( &inner_term );
35    solver.set_outer_termination_criterion( &outer_term );
36
37    solver.use_global_patch();
38
39    LinearFunctionSet mapping_functions;
```

7.3 Averaging A Jacobian-Based Metric Over Elements

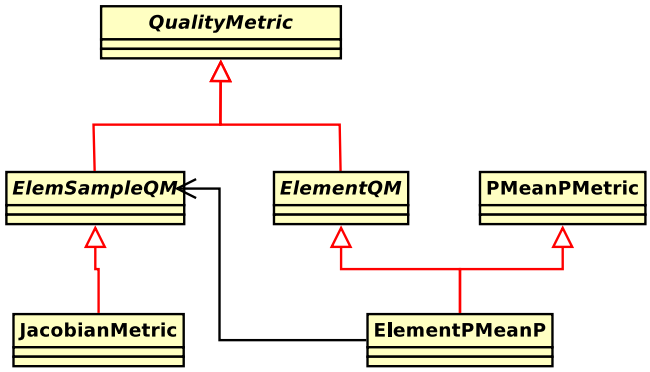

Figure 22: Relation between ElementPMeanP and JacobianMetric classes

This section describes how the setup in the previous section can be modified such that the values of the Jacobian-based metric are averaged over the sample points in each element. This change has the effect of converting the quality metric to an element-based metric. The change is effected by wrapping the `JacobianMetric` instance in an instance of the `ElementPMeanP` class. Figure 22 is a subset of the `QualityMetric` class diagram from Figure 7 showing the relation between the `JacobianMetric` and `ElementPMeanP` classes.

The source code from Listing 2 can be modified as follows to introduce element averaging. The `JacobianMetric` instance at line 17 is renamed from `metric` to `jmetric` and a new `ElementPMeanP` instance with the name `metric` is introduced at line 22. Listing 3 shows the modified source code.

Listing 3: Averaging Sample-Based Metrics Over Elements

```
17    JacobianMetric jmetric( &samples,
```

```
                                &target,
                                &weight,
                                &metric_2d,
                                NULL );
  ElementPMeanP metric( 1.0, &jmetric );
```

A Changes to `PatchData`

Two significant changes were made to the `PatchData` class since version 1.1. The first was the addition of the `ExtraData` class and related classes. These utility classes allow other code in Mesquite to associate arbitrary data with a `PatchData` and provide the necessary functionality to ensure data is correctly updated when the logical patch represented by the `PatchData` instance changes. This functionality is currently used by `CachingTargetCalculator` to store cached target matrices.

The second change, affecting much more existing Mesquite code, was to reorder vertices in the patch such that all free vertices occur first in the array, followed by all slave vertices[2], followed by all fixed vertices. While this does simplify code that iterates over free vertices in a patch, the primary motivation was to remove terms corresponding to fixed vertices from the gradient and Hessian data from an objective function. The vertex index in the `PatchData` corresponds to the vertex index in the vector of gradient terms and to the row and column positions in the MsqHessian. By definition, a non-free vertex is not a free variable in the objective function. Partial derivatives with respect to constant terms are by definition zero, and of no interest to the optimization. By moving the free vertices to the beginning of the array in the `PatchData`, the gradient and Hessian data can be reduced to only those terms that correspond to actual variables in the objective function.

This change removes unnecessary memory allocation (improving cache efficiency), localizes the portion of the vertex coordinate array that is modified during an optimization (again, improving cache efficiency), and is a better fit for the underlying logic of the optimization.

One could argue that the number of fixed vertices is small relative to the number of free vertices, as the former is $O(n^2)$ and the latter is $O(n^3)$. However, this ignores several important factors. The first is that while the fraction of boundary vertices decreases linearly with increasing mesh size, a quite large mesh is required to make it an insignificant fraction. For example, for a regular mesh of a cube it can be shown the the ratio of boundary to interior vertices is $\frac{2}{3i}$ where i is the number of intervals. To make this ratio a still fairly large value of $\frac{1}{10}$ (ten times as many interior as boundary veritces), the mesh must contain approximately 225,000 vertices. The second relevant factor is that while Mesquite is used with very large meshes, it is typically invoked on subsets of the mesh corresponding to material boundaries. The final relavant point to consider

[2]The ability to represent slave vertices will be used in future capabilities such as smoothing elements with higher-order nodes

is that when optimizing subsets of the mesh (patches) during either a BCD or Nash game solution, all the vertices on the boundary of the patch are logically fixed. A patch containing 225k nodes is much larger than what is typically used in Mesquite. In the limit of a patch with a single free vertex, this change will reduce the gradient and Hessian data by an order of magnitude.

However, the primary motivation for this change was to add support for *slave* nodes for future handling of quadratic elements. The higher-order nodes treated as *slave* nodes will greatly outnumber the free vertices in the mesh, regardless of the mesh size.

B `ObjectiveFunction` Interface

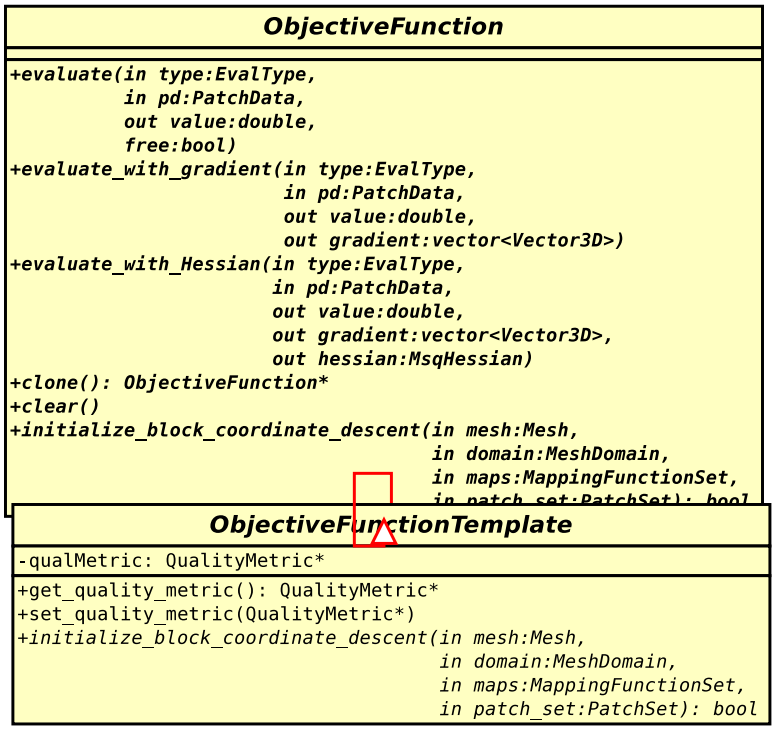

Figure 23: ObjectiveFunction Interface.

The interface to the `ObjectiveFunction` class (Figure 23) is fairly straightforward, with the exception of the arguments and functions for the handling of block coordinate descent (BCD) algorithms. The interface provides three forms of the `evaluate` method. The first returns only the scalar value of the objective function. The `evaluate_with_gradient` method returns both the value and the gradient of the objective function. The gradient is returned as a list if `Vector3D`

objects, one `Vector3D` for each free vertex in the `PatchData`. Each `Vector3D` contains the gradient terms with respect to the three coordinate values for the corresponding vertex. The `evaluate_with_Hessian` returns the scalar value, the gradient, and the Hessian of the objective function. The Hessian is returned using an `MsqHessian` object, which provides compact storage for the sparse Hessian data.

The first argument to each of the `evaluate*` methods is `EvalType`. Passing the `CALCULATE` value for this term will result in the objective function being evaluated only over the current patch. This is the mode used for Nash game and global patch optimizations. Other values, used only during BCD solutions, are discussed below. Not all `ObjectiveFunction` implementations support BCD-type algorithms. Such implementations will return an error condition if the evaluation type is anything other than `CALCULATE`.

The `clone` function is expected to return a new `ObjectiveFunction` instance that is a duplicate of the current instance, including any referenced `QualityMetrics`, accumulated BCD data, etc. The `clear` function is used to reset any accumulated BCD data.

The`ObjectiveFunction` interface declares a pure virtual method named `initialize_block_coordinate_descent`. A concrete `ObjectiveFunction` subclass that supports BCD must provide this method. The method initializes the value of the the objective function to $f(M_0)$, presumably by iterating over the mesh. A common implementation of this function is provided in `ObjectiveFunctionTemplate` for all template objective functions. Composite objective functions must provide their own implementation (typically by calling the corresponding method on their underlying `ObjectiveFunction` instance(s). The way the mesh is iterated over to establish the initial value depends on the type of the `QualityMetric`, so a common implementation for composite objective functions is not possible.

B.1 `EvalType`

The `enum EvalType` is the first argument to any of the methods used to evaluate the objective function value. This enum has five possible values: `CALCULATE`, `ACCUMULATE`, `SAVE`, `UPDATE`, `TEMPORARY`. The `CALCULATE` value requests Nash-game behavior, where the objective function is evaluated over the local patch only. The others are used for various states of a BCD optimization. Listing 4 shows an example of how these evaluation modes would be implemented for a simple algebraic mean objective function template.

`CALCULATE`

> This mode of evaluation is the equivalent of the old Nash-game behavior. The objective function is evaluated over the current patch. No accumulated data is modified or used in calculating the result value.

`ACCUMULATE`

An `ACCUMULATE` evaluation is used in the default implementation of `initialize_block_coordinate_descent` method provided by the `ObjectiveFunctionTemplate` class. The `ObjectiveFunction` implementation is expected to add the evaluation of the current patch to the total for the overall mesh, including any vertex counts, etc. That is, the value of $f(M_0)$ is constructed by repeatedly calling the `evaluate` method with patches containing non-intersecting subsets of M. The definition of a non-intersecting subset is dependent on the type of the `QualityMetric`. For example, for an element metric each patch will contain only a single element. For a vertex metric, each patch will contain a single free vertex and the adjacent elements.

An implementation does not need to support the `ACCUMULATE` mode in any of the evaluation methods other than the simple `evaluate` method. Further, an implementation need not support it at all if it provides it's own `initialize_block_coordinate_descent` implementation.

`SAVE`

A `SAVE` evaluation is passed the initial, unmodified patch (P_0) just before the inner loop of the optimization begins. The implementation is expected to save this value for later reference. The accumulated global objective function value should not be modified during this evaluation mode. The result value should be the (unmodified) accumulated value.

`UPDATE`

A `UPDATE` evaluation modifies both the accumulated objective function vacua and the saved patch value from the previous `SAVE` or `UPDATE` evaluation. During this evaluation mode, the objective function should update its accumulated value such that it is updated for a change in the passed patch to the current value from the previously saved value. Further, the saved patch value should be changed to the corresponding value for the current patch, as that is the value now accounted for in the accumulated value. The result value is the updated accumulated value.

This mode of evaluation is best explained describing the how it relates to several of the above modes in an optimization. The typical sequence of events is something like the following:

1. An accumulated value for the entire mesh is first calculated either with a call to `initialize_block_coordinate_descent` or repeated calls to the `evaluate` method in `ACCUMULATE` mode.
2. A patch is created containing some subset of the mesh that is to be modified.
3. `evaluate` is called in `SAVE` mode to inform the implementation of the initial state of the patch.

4. `evaluate` is called in `UPDATE` mode because the patch has been changed from it's previous state to the current one. The implementation is expected to update the accumulated objective function value accordingly.

5. `evaluate` is called in `UPDATE` mode again because the patch has been modified again.

`TEMPORARY`

The `TEMPORARY` evaluation mode returns the same result value to the caller as would have been returned for the `UPDATE` mode. However, *no* internal data including the previous patch state or the accumulated objective function value are modified. This mode of evaluation can be used by a solver to test how the objective function value would be affected by a change to the patch without actually effecting that change.

Listing 4: Example of `EvalType` implementation (algebraic mean).

```
bool AlgebraicMean::evaluate( EvalType eval_mode,
                              PatchData& patch,
                              double& result_value,
                              bool free,
                              MsqError& err )
{
    // get value for current patch
  int count;
  double value;
  bool rval = sum_metric_vals( patch, value, count, err );
  if (MSQ_CHKERR(err) || !rval)
    return false;

    // implement requested evaluation mode
  switch (eval_mode) {
    case CALCULATE: // Nash
      result_value = value/count;
      break;
    case ACCUMULATE: // Initialize
      accumulated_value += value;
      accumulated_count += count;
      result_value = accumulated_value/accumulated_count;
      break;
    case SAVE: // Store f(P_0)
      saved_value = value;
      saved_count = count;
      result_value = accumulated_value/accumulated_count;
      break;
    case UPDATE:
      assert(count == saved_count); // same patch
        // update accumulated value for change in patch
```

```
      accumulated_value -= saved_value;
      accumulated_value += value;
        // 'value' is now the portion of
        // accumulated_value corresponding to this patch
      saved_value = value;
      result_value = accumulated_value/accumulated_count;
      break;
    case TEMPORARY:  // result_value same as UPDATE, but
                     // accumulated_value isn't modified
      assert(count == saved_count); // same patch
      result_value = accumulated_value;
      result_value -= saved_value;
      result_value += value;
      result_value /= accumulated_count;
      break;
  } // end switch(eval_mode)
}
```

B.2 Numerical Approximation of Gradients

The evaluation modes described in section B.1 are also used by the code responsible for numerical approximation of objective function gradients. If the BCD-type evaluation modes are available, they will be used to more efficiently calculate the effect on the overall objective function due to the perturbation of a single vertex. If calling `evaluate` with the BCD evaluation modes fails, the code will fall back to the very inefficient solution requiring evaluation of the objective function over the entire mesh for every vertex coordinate perturbation.

B.3 What's New

The ability to select between numerical and analytical solutions for objective function gradients has been removed. It made the API more complicated and confusing for application developers; and it was a source of errors for developers (e.g. failure to set the default correctly for a concrete objective function implementation.) For the new interface, the `evaluate_with_gradient` is virtual, but not abstract. The default implementation provided in the `ObjectiveFunction` base class is the numerical approximation. `ObjectiveFunction` implementations that include analytical gradient calculation simply override that function. Thus the application developer always gets the analytical solution if it is available, and the numerical solution if the analytical one is not available without having to take any action. For the few cases such as testing where one may need to force numerical approximation, alternate solutions exist. See Appendix D.

Also, in the Mesquite 1.1 API, a quality metric negate flag was applied automatically for the value-only evaluation method in the `ObjectiveFunction` base class but individual subclasses were expected to incorporate it for evaluation methods returning derivatives. This type of inconsistency typically leads to bugs, and was therefore removed. Further, it was typically handled incorrectly

in the case of composite objective functions. Now, each template objective function is expected to apply the negate flag from it's quality metric, and composite objective function implementations need not be concerned with negation flags.

Support for BCD solutions did not exist in Mesquite 1.1. Both the changes to the `ObjectiveFunction` interface and the support in concrete `ObjectiveFunction` implementations is new. Further, code was added to utilize the `ObjectiveFunction` BCD API for more efficient numerical approximation of gradients.

The `ObjectiveFunctionTemplate` interface has been added, separating those metrics that have an underlying `QualityMetric` from composite objective functions and other cases where there is not necessarily a single `QualityMetric` instance. This eliminates possible non-sensical uses of the interface such as requesting the `QualityMetric` from a `CompositeOFMultiply`.

C `QualityMetric` Interface

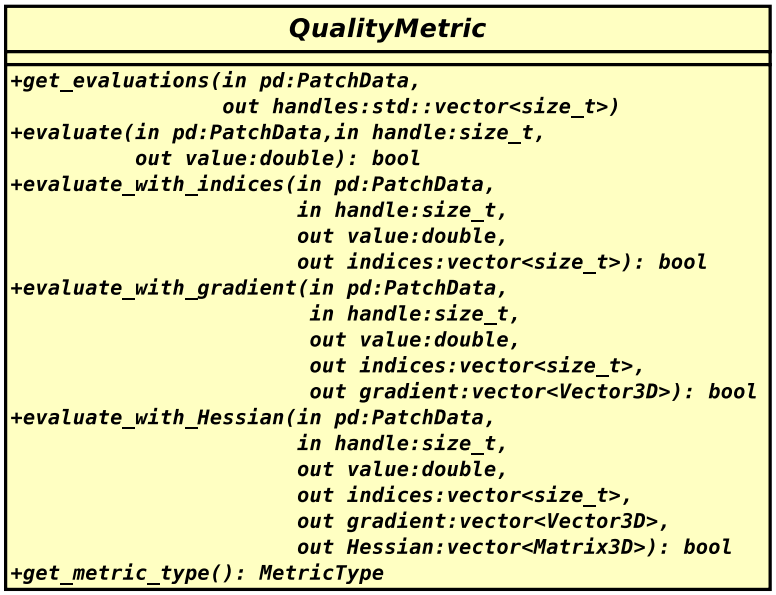

Figure 24: `QualityMetric` Interface.

The `QualityMetric` interface is abstracted such that it is applicable to all quality metrics (regardless of the mesh feature at which they are evaluated) through the use of evaluation handles. The `get_evaluations` method is passed a `PatchData` representing the current mesh patch. The method returns a list of *handles* corresponding to each local mesh feature in the patch at which the quality metric can be evaluated. For a vertex-based quality metric, the result of the function will be a handle for each vertex (presumably the vertex indices

in the PatchData). For an element-based metric, the result will be a value for each element in the patch. All the functions to evaluate the quality metric are passed a PatchData and one of the handles returned from get_evaluations, the combination of which specify a single mesh feature at which the metric can be evaluated.

Evaluation of the quality metric can return at most the following four pieces of data:

value The scalar value of the quality metric.

indices The list of free vertices for which the coordinates influence the value of the quality metric, specified as vertex indices in the corresponding PatchData.

gradient The gradient of the quality metric, as a std::vector of Vector3D objects. Each entry in the vector corresponds to the vertex specified in indices. Each value in a Vector3D is the partial derivative of the metric with respect to the corresponding vertex coordinate.

Hessian The Hessian of the quality metric as a std::vector of Matrix3D objects. A single Matrix3D object contains the second derivatives of the metric with respect to a pair of vertices. The std::vector represents the upper portion of a symmetric matrix of Matrix3D objects, in row-major order. That is, it does not contain entries for the lower portion of the matrix. The std::vector is in reduced form, but Matrix3D terms corresponding on the diagonal are complete even though those Matrix3D terms are symmetrical. No value is undefined or uninitialized.

Figure 25 shows an example of the index, gradient, and Hessian outputs for a quality metric evaluation that depends on the coordinates of free vertices at indices 6 and 7 in the PatchData.

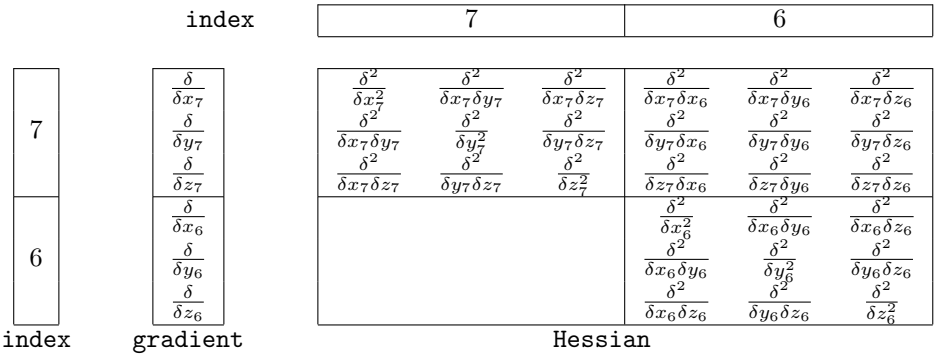

Figure 25: Quality metric evaluation arguments.

The evaluation functions provided in the QualityMetric interface pass combinations of the above values:

	value	index	gradient	Hessian
evaluate	yes			
evaluate_with_indices	yes	yes		
evaluate_with_gradient	yes	yes	yes	
evaluate_with_Hessian	yes	yes	yes	yes

The QualityMetric class provides default implementations of the evaluate_with_gradient and evaluate_with_Hessian methods that rely on the evaluate_with_indices and evaluate methods to generate numeric approximations of the derivatives by perturbing vertex coordinates. This is currently the only use of the evaluate_with_indices method in Mesquite.

The get_metric_type function returns either ELEMENT_BASED or VERTEX_BASED. This value is used to determine how to decompose the mesh into patches such that the metric is evaluated at every possible evaluation point exactly once and the patches are as small as possible. This is used to evaluate the metric over the entire mesh for operations such as quality assessment and initializing a coordinate descent optimization. A vertex metric will return VERTEX_BASED. Element-based metrics, corner-based metrics, metrics that are evaluated at sample points within an element, etc. return ELEMENT_BASED.

There is no EDGE_BASED option. This is because we assume that the mesh does not contain explicit edges, and therefore it is not possible to iterate over all of the edges directly. An edge-based metric could be implemented using a VERTEX_BASED patch type. For evaluation of the metric during normal optimization, all edges adjacent to the free vertices in the patch are evaluated. For quality assessment and coordinate descent initialization, only the edges in the patch that connect the center vertex to a second vertex with a smaller handle value are evaluated, resulting in all the edges being evaluated exactly once.

C.1 What's New

The most significant change to the QualityMetric interface was the generalization from the union of two interfaces, one for vertex metrics and one for element metrics, to a single unified interface for any metric type. In Mesquite 1.1, the QualityMetric interface contained separate evaluate_at_element and evaluate_at_vertex methods, and similar dual functions for derivatives. Which entities the metric could be evaluated for was implicit in the type: vertices for vertex-based metrics and elements for element-based metrics. The new interface uses the concept of evaluation handles to get rid of the dichotomy in evaluation functions. The metric gives the caller a list of entities at which the metric can be evaluated (vertices for vertex-based metrics and elements for element-based metrics), and the caller passes the values in that list to the evaluate* methods. This change has several advantages:

- Metric types other than element-based and vertex-based are possible.
- Calling code such as objective functions don't need to worry about testing for every possible metric type and calling the appropriate function.

- The interface no longer allows for non-sensical code such as calling `evaluate_at_vertex` on an element-based metric.
- A single common implementation of numerical derivative approximation works for all metric types.

In Mesquite 1.1 and earlier, the caller informed the `QualityMetric` of the vertices for which it was interested in derivative terms. This list was always the free vertices the metric depended on (e.g. the free vertices in an element for an element-based metric.) The information was communicated inconsistently for different functions. The caller is no longer expected to implicitly know which vertices a metric evaluation depends on. The metric is now expected to return to the caller the list of free vertices that the metric value was a function of. Not only is this necessary to make the interface metric-type-agnostic, it also simplifies the implementation of many metrics because it allows the metric to return derivative values in the order most convenient for calculation (Figure 25.)

The ability to select between numerical and analytical solutions for quality metric derivatives has been removed. It made the API more complicated and confusing for application developers; and it was a source of errors for developers (e.g. failure to set the default correctly for a concrete quality metric implementation.) For the new interface, the `evaluate_with_gradient` and `evaluate_with_Hessian` methods are virtual, but not abstract. The default implementations provided in the `QualityMetric` base class perform the numerical approximation. `QualityMetric` implementations that include analytical gradient or Hessian calculation simply override the corresponding function. Thus the application developer always gets the analytical solution if it is available, and the numerical solution if the analytical one is not available. For the few cases such as testing where one may need to force numerical approximation, alternate solutions exist. See Appendix D.

D Numerical Gradient and Hessian For Testing

The changes to to the `ObjectiveFunction` and `QualityMetric` interfaces (Sections B and C, respectively) include the removal of the ability to request numerical approximations of gradient and Hessian values when the classes provide analytical calculations. This makes the Mesquite API simpler and less error-prone for application developers, but there are a few cases such as testing where it may still be desirable to force numerical approximation of derivatives.

When the `ObjectiveFunction` or `QualityMetric` are being called directly, such as when comparing numerical and analytical values, the simplest is to invoke the numerical approximation methods in the interface class directly. This is the mechanism provided by the C++ language for bypassing polymorphism. For example, given a quality metric instance pointed to by `qm` that provides analytical gradients and Hessians, the following code will query the analytical gradient:

```
// get analytical gradient
qm->evaluate( patch, handle, value_out, err );
```

This is the normal method of calling the `evaluate` method, where the implementation of the virtual function in the leaf-most class is invoked. To obtain the numerical gradient, one must simply bypass the polymorphism of the virtual function, calling the numerical approximation implemented in the `QualityMetric` base class:

```
// get numerical gradient
qm->QualityMetric::evaluate( patch, handle, value_out, err );
```

For cases where it is not possible or advisable to modify the call to the `QualityMetric` or `ObjectiveFunction`, such as when the calls are made from within existing (non-test) Mesquite code, a wrapper class can be used. The wrapper class implements it's `evaluate` methods by calling the same function in the wrapped `QualityMetric` or `ObjectiveFunction`. For any method for which the numerical solution is to be returned, the wrapper just doesn't implement the method, causing the default numerical approximation provided by the base class to be used.

Listing 5 is an example of this technique. The class `NumericalHessianMetric` is delcared at line 3 to implement the `QualityMetric` interface. The constructor is declared to accept a pointer to the "real" `QualityMetric` that this class will serve as a wrapper for. All of the methods in the class are simply defined such that they pass the function call to the underlying, real `QualityMetric` implementation. However, as noted in the comment at line 85, the class does not provide an implementation of the `evaluate_at_Hessian` method. The class will inherit the `evaluate_at_Hessian` from `QualtiyMetric`, which is a numerical approximation.

Listing 5: Numerical `QualityMetric` Wrapper

```
1  // Define a class that uses analytical gradients
2  // (if they're available) but always uses numerical
3  // Hessians
4  class NumericalHessianMetric : public QualityMetric
5  {
6  private:
7    QualityMetric* realMetric;
8  public:
9
10   NumericalHessianMetric( QualityMetric* real_metric )
11     : realMetric(real_metric)
12     {}
13
14   MetricType get_metric_type() const
15     { return realMetric->get_metric_type(); }
16
17   msq_std::get_name() const
18     { return realMetric->get_name()
```

```
19        + " (Numerical Hessian)"; }
20
21    int get_negate_flag() const
22      { return realMetric->get_negate_flag(); }
23
24    void get_evaluations(
25                  PatchData& pd,
26                  msq_std::vector<size_t>& handles,
27                  bool free_only,
28                  MsqError& err )
29    {
30      realMetric->get_evaluations( pd,
31                                   handles,
32                                   free_only,
33                                   err );
34      MSQ_CHKERR(err);
35    }
36
37      // call realMetric to get actual value
38    bool evaluate( PatchData& pd,
39                   size_t handle,
40                   double& value,
41                   MsqError& err )
42    {
43      bool rval;
44      rval = realMetric->evaluate( pd, handle, value, err );
45      return !MSQ_CHKERR(err) && rval;
46    }
47
48      // call realMetric to get actual value
49    bool evaluate_with_indices(
50                  PatchData& pd,
51                  size_t handle,
52                  double& value,
53                  msq_std::vector<size_t>& indices,
54                  MsqError& err )
55    {
56      bool rval;
57      rval = realMetric->evaluate_with_indices( pd,
58                                                handle,
59                                                value,
60                                                indices,
61                                                err );
62      return !MSQ_CHKERR(err) && rval;
63    }
64
65      // call realMetric because we want analytical
66      // gradients if realMetric implements that.
67    bool evaluate_with_gradient(
68                  PatchData& pd,
```

```
69                  size_t handle,
70                  double& value,
71                  msq_std::vector<size_t>& indices,
72                  msq_std::vector<Vector3D>& grad,
73                  MsqError& err )
74    {
75      bool rval;
76      rval = realMetric->evaluate_with_gradient( pd,
77                                                 handle,
78                                                 value,
79                                                 indices,
80                                                 grad,
81                                                 err );
82      return !MSQ_CHKERR(err) && rval;
83    }
84
85      // NOTE: No evaluate_with_Hessian.
86      // By not providing an implementation here, the
87      // default numerical solution from QualityMetric
88      // will be inherited.  It will use analytical
89      // gradients in calculating the numerical
90      // Hessian because it will call the
91      // evaluate_with_gradient function above.
92  };
```