User Guide for `libDDG`

Keenan Crane

October 3, 2011

Contents

1 Overview

"Premature optimization is the root of all evil."

Donald Knuth

`libDDG` is a collection of C++ classes designed for use in the Discrete Differential Geometry (DDG) course at the California Institute of Technology and the University of Göttingen. Emphasis is placed on *pedagogy*, *readability*, and *simplicity* – the goal is to help students quickly prototype and experiment with geometry processing algorithms, and experiment with the design of mesh data structures. *Note that libDDG is* ***not*** *production-ready code!* In some cases performance and generality have been compromised for the sake of simplicity and legibility – students learning differential geometry have enough to worry about without decrypting template metaprograms.

The primary function of `libDDG` is to build and solve linear algebra problems based on polygonal meshes. `libDDG` also includes some facilities for visualization via OpenGL. The basic idea is that students are given "bare bones" data structures and slowly build up additional functionality over the duration of the course. Certain tasks (such implementing a high-performance, general purpose linear solver) are not the focus of the course – these components are instead supported by external libraries.

2 Acknowledgements

Thanks to Peter Schröder for feedback on library syntax and to Clarisse Weischedel for early beta testing and build support on Linux.

3 Getting Started

To build `libDDG` you will first need to install and/or build several dependencies: BLAS and SuiteSparse for linear algebra, GLUT and OpenGL for visualization. `libDDG` is designed to be used in a POSIX environment (Linux, BSD, Mac OS X, etc.) using the GNU toolchain. On Windows your best option may be to use MinGW or Cygwin. (If you are determined to use VisualStudio, see Evgenii Rudnyi's article on building SuiteSparse.)

3.1 Installing BLAS

Most platforms include a BLAS implementation by default. If you do not appear to have BLAS on your system (or do not know where it lives), take a look at

`http://www.netlib.org/blas/faq.html`

On Windows/Cygwin, simply (re-)run the Cygwin installer and select all the LAPACK packages in the *Math* category.

3.2 Installing SuiteSparse

The exact details of SuiteSparse installation may vary from system to system, but this guide should get you started. Note that SuiteSparse is installed by default on certain distributions of Linux. On Windows/Cygwin you can simply select the SuiteSparse packages in the Cygwin installer (and skip the instructions below); however, you may find that you need to manually download the file SuiteSparseQR.hpp from the SuiteSparse home page and place it in `C:\cygdrive\usr\include\suitesparse`.

1. Download SuiteSparse from

   `http://www.cise.ufl.edu/research/sparse/SuiteSparse/current/SuiteSparse.tar.gz`

2. Check which version of METIS is currently required by SuiteSparse by checking the SuiteSparse web page. As of October 3, 2011 SuiteSparse uses METIS version 4.01, which can be downloaded from

   `http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/OLD/metis-4.0.1.tar.gz`

   Other versions of METIS can be obtained from the METIS home page.

3. Unarchive the SuiteSparse distribution.

4. Unarchive the METIS distribution and *put it in the SuiteSparse directory* – otherwise SuiteSparse will not be able to find METIS.

5. Open `SuiteSparse/README.txt` – as mentioned there you will need to edit the file `SuiteSparse/UFconfig/UFconfig.mk` to specify options for your platform.

6. Type `make` in the SuiteSparse directory.

7. Type `sudo make install` in the SuiteSparse directory (you will be prompted for your password). For some reason SuiteSparse does not always set the correct permissions on header files. If you are getting "permission denied" errors, try typing

   `sudo chmod ugo+r /usr/local/include/*.h /usr/local/include/*.hpp`.

8. Install the METIS library by typing

   `sudo cp metis-4.0/libmetis.a /usr/local/lib`

   (or the appropriate subdirectory) in the SuiteSparse root directory.

3.3 Installing OpenGL/GLUT

Most platforms include OpenGL and GLUT by default, but if you are on Windows (and not using Cygwin) you will need to install the version of GLUT available here:

http://www.xmission.com/~nate/glut.html

3.4 Building libDDG

Once all dependencies are bulit and installed, you need to specify their locations by editing the `Makefile` in the root `libddg` directory. Default options for several platforms are available but may need to be modified for your particular environment. Finally, type `make` in the root `libDDG` directory, which should build the executable `ddg`. You can check to see if the code was built successfully by running

```
./ddg data/bunny.obj
```

which should bring up a window displaying the Stanford bunny.

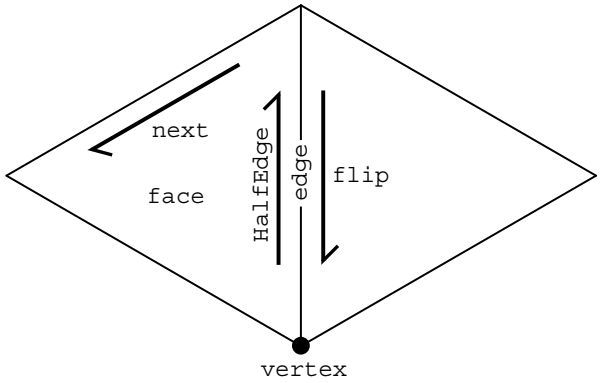
4 Basic Routines and Data Structures

This section gives a high-level introduction to the major classes in `libDDG` – for more detailed documentation of individual methods see Section 5.

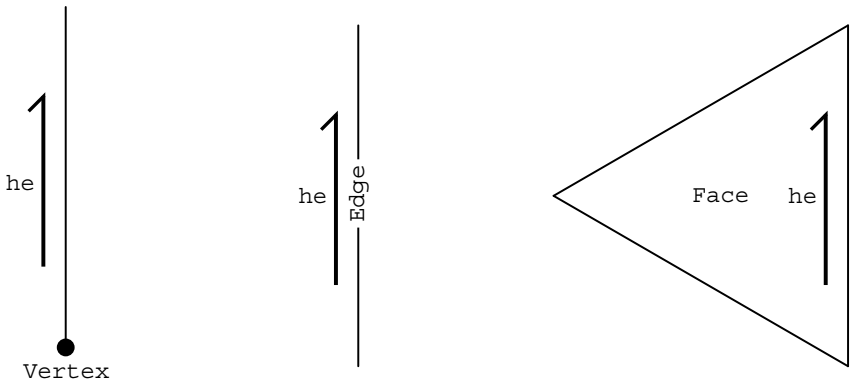
4.1 Geometry

The representation for geometry in `libDDG` is a bare-bones *halfedge* data structure, which encodes a mesh with polygonal faces. The idea is that students will extend this data structure over the course of several assignments. Mesh connectivity is determined by a collection of *halfedges*, which encode the incidence relationships among mesh elements (vertices, edges, and faces):

```
class HalfEdge
{
   public:
      HalfEdgeIter next;
      HalfEdgeIter flip;
      VertexIter vertex;
      EdgeIter edge;
      FaceIter face;
};
```



Other mesh elements do not encode connectivity information – they are used simply as a place to store mesh attributes (normals, colors, etc.). However, each mesh element stores a reference to one of its halfedges:



The primary benefit of the halfedge data structure is that it provides a lot of flexibility for iterating over different mesh regions. For instance, to iterate over all the vertices of a face `f` we might write the following loop:

```
HalfEdgeIter he = f->he;
do
{
   // do something with he->vertex

   he = he->next;
}
while( he != f->he );
```

Alternatively, the following loop iterates over all vertices adjacent to a given vertex `v`:

```
HalfEdgeIter he = v->he;
do
{
   // do something with he->flip->vertex

   he = he->flip->next;
}
while( he != v->he );
```

(Most loops will have this same sort of "do-while" structure.) Note that *by construction* the halfedge data structure can represent only orientable surfaces where every edge is manifold, i.e., every edge is contained in at most two faces. Non-manifold vertices are possible, however.

4.1.1 Mesh Attributes

One challenging question in the design of mesh data structures is: how do you associate data with mesh elements? Users need to store a variety of different mesh attributes depending on the particular geometry processing problem, but for some reason (possibly related to the available programming paradigms in C++) managing different meshes with different attributes can add considerable complexity to the code. There are several approaches, but to date it is not clear that one of these approaches is the clear "winner" – CGAL, OpenMesh, and Mesquite demonstrate several possibilities; Seiger and Botsch provide a survey of these data structures and propose an attractive alternative [1]. `libDDG` remains

largely agnostic to this issue, providing only a bare-bones data structure that challenges students to explore these design issues for themselves.

For very simple projects, however, it is usually sufficient to just add mesh attributes directly to element classes. For instance, to associate a color with each vertex one could modify the `Vertex` class to look like this:

```
class Vertex
{
   public:
      HalfEdgeIter he;

      Vector color; // new attribute
};
```

4.1.2 Boundaries

Surfaces with boundary are handled by treating each boundary loop as an additional face in the mesh. To determine whether a face `f` is a boundary face, simply call the routine `f->isBoundary()`. For instance, to iterate over all *non*-boundary faces one might write

```
for( FaceIter f = faces.begin(); f != faces.end(); f++ )
{
   if( !f->isBoundary() )
   {
      // do some stuff
   }
}
```

4.2 Linear Algebra

At the most basic level, `libDDG` provides a wrapper around the SuiteSparse library which can be used to build and solve sparse linear systems. For instance, to solve the system $Ax = b$ with

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

and

$$b = \begin{bmatrix} 5 \\ 6 \end{bmatrix},$$

we can simply write

```
SparseMatrix A( 2, 2 );
A(0,0) = 1; A(0,1) = 3;
A(1,0) = 2; A(1,1) = 4;

DenseMatrix b( 2, 1 );
b(0) = 5;
b(1) = 6;

DenseMatrix x( 2, 1 );
solve( A, x, b );
```

The solution will be stored in the dense vector `x`.

4.2.1 Automatic Indexing

The most meticulous part of geometry processing is often establishing a correspondence between mesh elements and variables in a linear system. The usual idea is that each element is given a unique integer ID which is used as a row or column index in a matrix. In many cases, however, there is no good reason for the programmer to think about a linear system in terms of matrices, indices, and so forth. In these situations, `libDDG` provides an abstraction that automatically handles variable indexing and indeed the entire process of solving a linear system. This abstraction is especially useful for debugging and rapid

prototyping (though it should be noted that better performance can often be achieved by building matrices "by hand.")

For example, suppose one wants to solve the linear system

$$\begin{aligned} 3(x + y/2) + z &= 4 \\ z - 8 &= y + x/9 \\ (x + y)/5 + (y + z)/6 &= 7 \end{aligned}$$

for the values x, y, and z. In `libDDG` this system can be solved by simply writing

```
LinearSystem sys;
Variable x, y, z;

sys.push_back(     3*(x+y/2) + z == 4     );
sys.push_back(               z-8 == y+x/9 );
sys.push_back( (x+y)/5 + (y+z)/6 == 7     );

sys.solve();
```

There are a few things to note here. First, we did not need to put the equations into some kind of special form (e.g., we did not have to move all terms to the left-hand side or expand the expression in terms of primary variables). This kind of flexibility makes it easy to verify *visually* that we typed in the right set of equations. Second, note that the method `LinearSystem::solve()` automatically copies the solution of the system back into our variables `x`, `y`, and `z`. For instance, if we now wanted to display the solution we could simply write

```
cout << "x: " << *x << endl;
cout << "y: " << *y << endl;
cout << "z: " << *z << endl;
```

Here the dereference operator (`*`) is used to access the numerical value stored in each variable (whereas variables are typically treated as purely symbolic objects).

4.2.2 Fixed Variables

In many cases it is useful to be able to *fix* certain variables (i.e., hold their values constant) while solving a linear system – for instance, one might want to

prescribe boundary conditions in a finite element problem. In `libDDG` , variables can be fixed by setting the flag `Variable::fixed`. For instance, consider the system

$$\begin{aligned} x + y &= a \\ x - y &= b \end{aligned}$$

which can be encoded via

```
LinearSystem sys;
Variable x, y, a, b;

sys.push_back( x + y == a );
sys.push_back( x - y == b );
```

Initially, all the variables (x, y, a, and b) are degrees of freedom in our system. To fix the values on the right hand, we simply write

```
a.fixed = true;
b.fixed = true;
```

Now when we call *sys.solve*(), the variables `a` and `b` will automatically be interpreted as constants. For instance, to solve a sequence of systems with different right-hand sides we could write

```
*a = 1;
*b = 2;
sys.solve();

*a = 3;
*b = 4;
sys.solve();
```

Moreover, we can dynamically change the set of fixed variables. For instance, to keep `x` and `y` fixed while solving for `a` and `b`, we would write something like

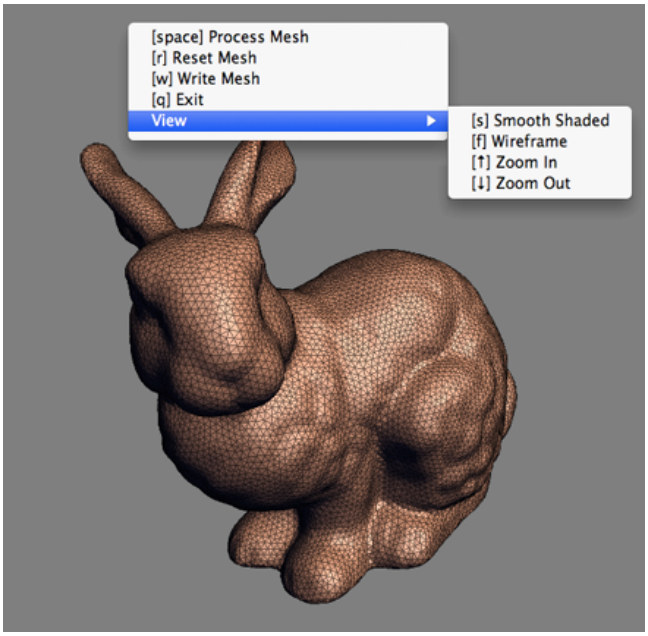
```
a.fixed = false;
b.fixed = false;

x.fixed = true;
y.fixed = true;

*x = 1.23;
*y = 4.56;

sys.solve();
```

4.3 Visualization



The `Viewer` class provides basic facilities for inspecting and interacting with a mesh. Additional functionality can be added using OpenGL and GLUT (see NeHe for a nice collection of OpenGL/GLUT tutorials). Setting up a viewer for a mesh can be done in just three lines:

```
Viewer viewer;
viewer.mesh.read( meshFilename );
viewer.init();
```

which should bring up a viewer window similar to the one pictured above. Right-clicking on this window brings up a list of commands and their keyboard equivalents.

5 Library Reference

5.1 DenseMatrix

DenseMatrix represents an m by n (real or complex) matrix where every entry – including zero-valued entries – is stored explicitly. This class is most commonly used to represent dense vectors in sparse linear systems (i.e., the right hand side and the solution vector).

A real or complex matrix is allocated via

```
DenseMatrix A( m, n );
DenseMatrix A( m, n, Complex );
```

Matrix elements are then accessed using parenthesis, e.g.,

```
A(i,j) = 1;
A(i,j) += 2;
a = A(i,j);
```

etc.

DenseMatrix is interoperable with the SuiteSparse numerical linear algebra library. In particular, dereferencing a DenseMatrix returns a cholmod_dense* which can be used by routines in SuiteSparse. For basic operations, however, you should not need to access this pointer explicitly – see the solve() method in SparseMatrix.h.

5.1.1 Class Reference

```
DenseMatrix( int m = 0, int n = 0, int xtype = Real );
```

Purpose: initialize an mxn matrix of doubles xtype is either DDG::Real or DDG::Complex

---

```
DenseMatrix( const DenseMatrix& A );
```

Purpose: copy constructor

```
const DenseMatrix& operator=( const DenseMatrix& A );
```

Purpose: copies A

```
~DenseMatrix( void );
```

Purpose: destructor

```
int nRows( void ) const;
```

Purpose: returns the number of rows

```
int nColumns( void ) const;
```

Purpose: returns the number of columns

```
int length( void ) const;
```

Purpose: returns the size of the largest dimension

```
void zero( double rVal = 0., double iVal = 0. );
```

Purpose: sets all elements to rVal+iVal*i

```
double norm( void );
```

Purpose: returns the maximum magnitude of any entry

```
double& operator()( int row, int col );
double  operator()( int row, int col ) const;
double& r( int row, int col );
double  r( int row, int col ) const;
```

Purpose: access real part of element (row,col) note: uses 0-based indexing

---

```
double& i( int row, int col );
double  i( int row, int col ) const;
```

Purpose: access imaginary part of element (row,col) note: uses 0-based indexing

---

```
double& operator()( int index );
double  operator()( int index ) const;
```

Purpose: access real part of element ind of a vector note: uses 0-based indexing

---

```
double& r( int index );
double  r( int index ) const;
```

Purpose: access real part of element ind of a vector note: uses 0-based indexing

---

```
double& i( int index );
double  i( int index ) const;
```

Purpose: access imaginary part of element ind of a vector note: uses 0-based indexing

---

```
cholmod_dense* operator*( void );
```

Purpose: returns pointer to underlying cholmod_dense data structure

---

```
const DenseMatrix& operator=( cholmod_dense* A );
```

Purpose: gets pointer to A; will deallocate A upon destruction

5.2 Edge

Edge stores attributes associated with a mesh edge. The iterator he points to one of its two associated halfedges. (See the documentation for a more in-depth discussion of the halfedge data structure.)

5.2.1 Class Reference

```
HalfEdgeIter he;
```

Purpose: points to one of the two halfedges associated with this edge

5.3 Face

Face stores attributes associated with a mesh edge. The iterator he points to one of its associated halfedges. (See the documentation for a more in-depth discussion of the halfedge data structure.)

5.3.1 Class Reference

```
HalfEdgeIter he;
```

Purpose: points to one of the halfedges associated with this face

---

```
bool isBoundary( void ) const;
```

Purpose: returns true if this face corresponds to a boundary loop; false otherwise

---

```
Vector normal( void ) const;
```

Purpose: returns the unit normal associated with this face; normal orientation is determined by the circulation order of halfedges

5.4 HalfEdge

HalfEdge is used to define mesh connectivity. (See the documentation for a more in-depth discussion of the halfedge data structure.)

5.4.1 Class Reference

```
HalfEdgeIter next;
```

Purpose: points to the next halfedge around the current face

```
HalfEdgeIter flip;
```

Purpose: points to the other halfedge associated with this edge

---

```
VertexIter vertex;
```

Purpose: points to the vertex at the "tail" of this halfedge

```
EdgeIter edge;
```

Purpose: points to the edge associated with this halfedge

```
FaceIter face;
```

Purpose: points to the face containing this halfedge

```
bool onBoundary;
```

Purpose: true if this halfedge is contained in a boundary loop; false otherwise

---

```
Vector texcoord;
```

Purpose: texture coordinates associated with the triangle corner at the "tail" of this halfedge

5.5 LinearContext

LinearContext is the global solver context needed to interface with the SuiteSparse library. It is essentially a wrapper around cholmod_common. A single static instance of LinearContext is declared in LinearContext.cpp and is shared by all instances of DenseMatrix, SparseMatrix, and LinearSystem. In other words, you shouldn't have to instantiate LinearContext yourself unless you're doing something really fancy!

5.5.1 Class Reference

```
LinearContext( void );
```

Purpose: constructor

```
~LinearContext( void );
```

Purpose: destructor

```
operator cholmod_common*( void );
```

Purpose: allows LinearContext to be treated as a cholmod_common*

5.6 LinearEquation

LinearEquation represents an equation with an arbitrary linear polynomial on both the left- and right-hand side. It is primarily used while building a LinearSystem. For convenience, operator== is overloaded so that the user can construct a LinearEquation by writing something that looks much like the usual mathematical syntax for a linear equation. For example,

```
LinearEquation eqn = ( x + 2*y == 3*z );
```

builds the linear equation x + 2y = 3z.

5.6.1 Class Reference

`LinearPolynomial lhs;`

Purpose: left-hand side

`LinearPolynomial rhs;`

Purpose: right-hand side

5.7 LinearPolynomial

LinearPolynomial represents an affine function of the form

$$f(x_1, x_2, ..., x_n) = c_1x_1 + c_2x_2 + ... + c_nx_n + d$$

where the x_i are real-valued variables with real coefficients c_i, and d is a real constant. The variables and their coefficients are represented using instances of the Variable class. LinearPolynomial implements all the usual algebraic operations on affine functions, as well as type conversions from more elementary types (scalars, single variables, etc.).

Importantly, variables used in a LinearPolynomial should **not** be deallocated while the polynomial is still in use – LinearPolynomial stores only a reference to these variables so that the solution to a linear system can be automatically copied back into the variables.

5.7.1 Class Reference

```
LinearPolynomial( void );
```

Purpose: constructs the zero function

```
LinearPolynomial( double c );
```

Purpose: constructs the constant function with value c

```
LinearPolynomial( Variable& v );
```

Purpose: constructs a function with a single variable v

```
const LinearPolynomial& operator=( double c );
```

Purpose: assigns the constant function with value c

```
const LinearPolynomial& operator=( Variable& v );
```

Purpose: assigns a function with a single variable v

```
void operator+=( double c );
void operator-=( double c );
void operator*=( double c );
void operator/=( double c );
```

Purpose: adds, subtract, multiplies, or divides by a constant

---

```
void operator+=( Variable& v );
void operator-=( Variable& v );
```

Purpose: increments or decrements by a single variable v

```
void operator+=( const LinearPolynomial& p );
void operator-=( const LinearPolynomial& p );
```

Purpose: increments or decrements by an affine function

```
LinearPolynomial operator-( void ) const;
```

Purpose: returns the additive inverse (i.e., negation)

```
double evaluate( void ) const;
```

Purpose: evaluates the function using the current values of its variables

---

```
std::map<Variable*,double> linearTerms;
```

Purpose: list of linear terms

```
double constantTerm;
```

Purpose: constant term

5.8 LinearSystem

LinearSystem represents a system of linear equations expressed in terms of instances of the Variable class. The main idea is to make it easy to construct and solve linear systems without explicitly think about variable indices, matrix layout, etc. (This kind of abstraction is particularly useful for debugging and rapid prototyping.) See the documentation for examples of building linear and solving systems.

Importantly, any variable used by a LinearSystem should not be deallocated while the system is still in use, because the method LinearSystem::solve() automatically copies the solution back into the variables used to define the equations. (In the future variables may become reference-counted in order to avoid this issue.)

Note that LinearSystem::solve() uses a general-purpose linear solver (namely the sparse QR factorization found in SuiteSparse) that is quite fast but may not always be your best option. To improve performance you may want to build the system explicitly using an instance of SparseMatrix and call a more specialized solver. (In the future there may be options for specifying that a LinearSystem is, e.g., symmetric and positive-definite.)

5.8.1 Class Reference

```
void clear( void );
```

Purpose: removes all equations from the system

```
void push_back( const LinearEquation& e );
```

Purpose: appends the equation e to the sytem

```
void solve( void );
```

Purpose: solves the system and automatically stores the result in the variables for an overdetermined system, computes a least-squares solution

---

```
std::vector<LinearEquation> equations;
```

Purpose: the collection of equations defining the system

5.9 Mesh

Mesh represents a polygonal surface mesh using the halfedge data structure. It is essentially a large collection of disjoint vertices, edges, and faces that are "glued together" by halfedges which encode connectivity (see the documentation for an illustration). By construction, the halfedge data structure cannot represent nonorientable surfaces or meshes with nonmanifold edges.

Mesh elements are referenced using iterators – common usage of these iterators is to either traverse an entire vector of mesh elements:

```
// visit all vertices
for( VertexIter i = vertices.begin(); i != vertices.end(); i++ )
{
   // ...
}
```

or to perform a local traversal over the neighborhood of some mesh element:

```
// visit both halfedges of edge e
HalfEdgeIter he = e->he;
do {
   // ...
   he = he->flip;
} while( he != e->he );
```

(See Types.h for an explicit definition of iterator types.)

Meshes with boundary are handled by creating an additional face for each boundary loop (the method Face::isBoundary() determines whether a given face is a boundary loop). Isolated vertices (i.e., vertiecs not contained in any edge or face) reference a dummy halfedge and can be checked via the method Vertex::isIsolated().

5.9.1 Class Reference

```
Mesh( void );
```

Purpose: constructs an empty mesh

```
Mesh( const Mesh& mesh );
```

Purpose: constructs a copy of mesh

```
const Mesh& operator=( const Mesh& mesh );
```

Purpose: copies mesh

```
int read( const std::string& filename );
```

Purpose: reads a mesh from a Wavefront OBJ file; return value is nonzero only if there was an error

---

```
int write( const std::string& filename ) const;
```

Purpose: writes a mesh to a Wavefront OBJ file; return value is nonzero only if there was an error

---

```
bool reload( void );
```

Purpose: reloads a mesh from disk using the most recent input filename

---

```
std::vector<HalfEdge> halfedges;
std::vector<Vertex>   vertices;
std::vector<Edge>     edges;
std::vector<Face>     faces;
```

Purpose: storage for mesh elements

5.10 MeshIO

MeshIO handles input/output operations for Mesh objects. Currently the only supported mesh format is Wavefront OBJ – for a format specification see
http://en.wikipedia.org/wiki/Wavefront_.obj_file
Note that vertex normals and material properties are currently ignored.

5.10.1 Class Reference

```
static int read( std::istream& in, Mesh& mesh );
```

Purpose: reads a mesh from a valid, open input stream in

```
static void write( std::ostream& out, const Mesh& mesh );
```

Purpose: writes a mesh to a valid, open output stream out

5.11 Quaternion

Quaternion represents an element of the quaternions, along with all the usual vectors space operations (addition, multiplication by scalars, etc.). The Hamilton product is expressed using the * operator:

```
Quaternion p, q, r;
r = q * p;
```

and conjugation is expressed using the method Quaternion::bar():

```
Quaternion q;
double normQSquared = -q.bar()*q;
```

Individual components can be accessed in several ways: the real and imaginary parts can be accessed using the methods Quaternion::re() and Quaternion::im():

```
Quaternion q;
double a = q.re();
Vector b = q.im();
```

or by index:

```
Quaternion q;
double a  = q[0];
double bi = q[1];
double bj = q[2];
double bk = q[3];
```

5.11.1 Class Reference

`Quaternion( void );`

Purpose: initializes all components to zero

`Quaternion( const Quaternion& q );`

Purpose: initializes from existing quaternion

`Quaternion( double s, double vi, double vj, double vk );`

Purpose: initializes with specified real (s) and imaginary (v) components

---

`Quaternion( double s, const Vector& v );`

Purpose: initializes with specified real (s) and imaginary (v) components

---

```
Quaternion( double s );
```

Purpose: initializes purely real quaternion with specified real (s) component

---

```
Quaternion( const Vector& v );
```

Purpose: initializes purely imaginary quaternion with specified imaginary (v) component

---

```
const Quaternion& operator=( double s );
```

Purpose: assigns a purely real quaternion with real value s

```
const Quaternion& operator=( const Vector& v );
```

Purpose: assigns a purely real quaternion with imaginary value v

---

```
double& operator[]( int index );
```

Purpose: returns reference to the specified component (0-based indexing: r, i, j, k)

```
const double& operator[]( int index ) const;
```

Purpose: returns const reference to the specified component (0-based indexing: r, i, j, k)

```
void toMatrix( double Q[4][4] ) const;
```

Purpose: builds 4x4 matrix Q representing (left) quaternion multiplication

---

```
double& re( void );
```

Purpose: returns reference to double part

```
const double& re( void ) const;
```

Purpose: returns const reference to double part

```
Vector& im( void );
```

Purpose: returns reference to imaginary part

```
const Vector& im( void ) const;
```

Purpose: returns const reference to imaginary part

```
Quaternion operator+( const Quaternion& q ) const;
```

Purpose: addition

```
Quaternion operator-( const Quaternion& q ) const;
```

Purpose: subtraction

```
Quaternion operator-( void ) const;
```

Purpose: negation

```
Quaternion operator*( double c ) const;
```

Purpose: right scalar multiplication

```
Quaternion operator/( double c ) const;
```

Purpose: scalar division

```
void operator+=( const Quaternion& q );
```

Purpose: addition / assignment

```
void operator+=( double c );
```

Purpose: addition / assignment of pure real

```
void operator-=( const Quaternion& q );
```

Purpose: subtraction / assignment

```
void operator-=( double c );
```

Purpose: subtraction / assignment of pure real

```
void operator*=( double c );
```

Purpose: scalar multiplication / assignment

```
void operator/=( double c );
```

Purpose: scalar division / assignment

```
Quaternion operator*( const Quaternion& q ) const;
```

Purpose: Hamilton product

```
void operator*=( const Quaternion& q );
```

Purpose: Hamilton product / assignment

```
Quaternion bar( void ) const;
```

Purpose: conjugation

```
Quaternion inv( void ) const;
```

Purpose: inverse

```
double norm( void ) const;
```

Purpose: returns Euclidean length

```
double norm2( void ) const;
```

Purpose: returns Euclidean length squared

```
Quaternion unit( void ) const;
```

Purpose: returns unit quaternion

```
void normalize( void );
```

Purpose: divides by Euclidean length

5.12 SparseMatrix

SparseMatrix represents an m by n (real or complex) matrix where only nonzero entries are stored explicitly. This class is most commonly used to represent the linear term in sparse linear systems (i.e., the matrix part).

A real or complex matrix is allocated via

```
SparseMatrix A( m, n );
SparseMatrix A( m, n, Complex );
```

Matrix elements are then accessed using parenthesis, e.g.,

```
A(i,j) = 1;
A(i,j) += 2;
a = A(i,j);
```

etc.

SparseMatrix is interoperable with the SuiteSparse numerical linear algebra library. In particular, dereferencing a SparseMatrix returns a cholmod_sparse* which can be used by routines in SuiteSparse. For basic operations, however, you should not need to access this pointer explicitly – see the solve() method below.

Internally SparseMatrix stores nonzero entries in a heap data structure; the amortized cost of insertion is therefore no worse than the sorting cost of putting the matrix in compressed-column order.

5.12.1 Class Reference

```
SparseMatrix( int m = 0, int n = 0, int xtype = Real );
```

Purpose: initialize an mxn matrix of doubles xtype is either DDG::Real or DDG::Complex

---

```
~SparseMatrix( void );
```

Purpose: destructor

```
void resize( int m, int n );
```

Purpose: clears and resizes to mxn matrix

```
cholmod_sparse* operator*( void );
```

Purpose: dereference operator gets pointer to underlying cholmod_sparse data structure

---

```
int nRows( void ) const;
```

Purpose: returns the number of rows

```
int nColumns( void ) const;
```

Purpose: returns the number of columns

```
int length( void ) const;
```

Purpose: returns the size of the largest dimension

```
void zero( double rVal = 0., double iVal = 0. );
```

Purpose: sets all nonzero elements to rVal+iVal*i

```
void transpose( void );
```

Purpose: replaces this matrix with its transpose

```
void horzcat( const SparseMatrix& A, const SparseMatrix& B );
```

Purpose: replaces the current matrix with [A, B]

```
void vertcat( const SparseMatrix& A, const SparseMatrix& B );
```

Purpose: replaces the current matrix with [A; B]

```
double& operator()( int row, int col );
double  operator()( int row, int col ) const;
double& r( int row, int col );
double  r( int row, int col ) const;
```

Purpose: access real part of element (row,col) note: uses 0-based indexing

---

```
double& i( int row, int col );
double  i( int row, int col ) const;
```

Purpose: access imaginary part of element (row,col) note: uses 0-based indexing

---

```
typedef std::pair<int,int> EntryIndex;
```

Purpose: convenience type for an entry index; note that we store column THEN row, which makes it easier to build compressed column format

---

```
typedef std::pair<double,double> EntryValue;
```

Purpose: convenience type for a complex entry value (real,imaginary)

```
typedef std::map<EntryIndex,EntryValue> EntryMap;
typedef EntryMap::iterator       iterator;
typedef EntryMap::const_iterator const_iterator;
```

Purpose: convenience type for storing and accessing entries

```
iterator begin( void );
const_iterator begin( void ) const;
iterator   end( void );
const_iterator   end( void ) const;
```

Purpose: return iterators to first and last nonzero entries

5.13 Variable

Variable represents a variable that can be used to define a (linear or nonlinear) system of equations. Its main feature is that it can be used both as an abstract variable (e.g., when used to define an equation) but can also store a definite numerical value. For instance, suppose we define a linear polynomial

```
Variable x, y;
LinearPolynomial p = x + 2*y;
```

Now by assigning different numerical values to x and y, we can evaluate the polynomial at different points:

```
*x = 1;
*y = 2;
cout << p.evaluate() << endl;


*x = 3;
*y = 4;
cout << p.evaluate() << endl;
```

In general the dereference operator (*) accesses the numerical value. Variables can also be named in order to aid with debugging. For instance,

```
Variable x( "x" );
Variable y( "y" );
Polynomial p = x + 2*y;
cout << p << endl;
```

will print out something like `x+2*y`.

The "fixed" flag in a variable refers to whether it is held constant while solving a system of equations – see the documentation for further discussion.

5.13.1 Class Reference

```
Variable( double value = 0., bool fixed = false );
```

Purpose: initialize a variable which has value zero and is not fixed by default

---

```
Variable( std::string name, double value = 0., bool fixed = false );
```

Purpose: initialize a named variable which has value zero and is not fixed by default

---

```
double& operator*( void );
```

Purpose: returns a reference to the numerical value

```
const double& operator*( void ) const;
```

Purpose: returns a const reference to the numerical value

```
std::string name;
```

Purpose: names the variable (for display output)

```
double value;
```

Purpose: numerical value

```
bool fixed;
```

Purpose: true if a variable is held constant while solving a system of equations

5.14 Vector

Vector represents an element of Euclidean 3-space, along with all the usual vectors space operations (addition, multiplication by scalars, etc.). The inner product (i.e., scalar or dot product) is expressed using the global method dot():

```
Vector u, v;
double cosTheta = dot( u, v );
```

and the cross product is expressed using the global method cross():

```
Vector u, v, w;
w = cross( u, v );
```

Individual components can be accessed in two ways: either directly via the members x, y, and z:

```
Vector v;
cout << v.x << endl;
cout << v.y << endl;
cout << v.z << endl;
```

or by index:

```
Vector v;
for( int i = 0; i < 3; i++ )
{
   cout << v[i] << endl;
}
```

5.14.1 Class Reference

`Vector();`

Purpose: initializes all components to zero

`Vector( double x, double y, double z);`

Purpose: initializes with specified components

`Vector( const Vector& v );`

Purpose: initializes from existing vector

`double& operator[] ( const int& index );`

Purpose: returns reference to the specified component (0-based indexing: x, y, z)

```
const double& operator[] ( const int& index ) const;
```

Purpose: returns const reference to the specified component (0-based indexing: x, y, z)

```
Vector operator+( const Vector& v ) const;
```

Purpose: addition

```
Vector operator-( const Vector& v ) const;
```

Purpose: subtraction

```
Vector operator-( void ) const;
```

Purpose: negation

```
Vector operator*( const double& c ) const;
```

Purpose: right scalar multiplication

```
Vector operator/( const double& c ) const;
```

Purpose: scalar division

```
void operator+=( const Vector& v );
```

Purpose: addition / assignment

```
void operator-=( const Vector& v );
```

Purpose: subtraction / assignment

```
void operator*=( const double& c );
```

Purpose: scalar multiplication / assignment

```
void operator/=( const double& c );
```

Purpose: scalar division / assignment

```
double norm( void ) const;
```

Purpose: returns Euclidean length

```
double norm2( void ) const;
```

Purpose: returns Euclidean length squared

```
Vector unit( void ) const;
```

Purpose: returns unit vector

```
void normalize( void );
```

Purpose: divides by Euclidean length

```
Vector abs( void ) const;
```

Purpose: returns vector containing magnitude of each component

---

```
double x, y, z;
```

Purpose: components

5.15 Vertex

Vertex stores attributes associated with a mesh edge. The iterator he points to its "outgoing" halfedge. (See the documentation for a more in-depth discussion of the halfedge data structure.)

5.15.1 Class Reference

```
HalfEdgeIter he;
```

Purpose: points to the "outgoing" halfedge

```
Vector position;
```

Purpose: location of vertex in Euclidean 3-space

```
Vector normal( void ) const;
```

Purpose: returns the vertex normal

```
bool isIsolated( void ) const;
```

Purpose: returns true if the vertex is not contained in any face or edge; false otherwise

References

[1] Daniel Sieger and Mario Botsch. Design, implementation, and evaluation of the `Surface_mesh` data structure. In *Proceedings of the 20th International Meshing Roundtable*, 2011.