HiQLab program documentation

David Bindel

August 9, 2007

Contents

Chapter 1

Core libraries

1.1 Mesh objects

The mesh data structure keeps track of the problem description and the state of the system. It does *not* keep track of the assembled stiffness matrix, nor the intermediate vectors used in Newton iterations, time-steps, etc; those are stored external to the mesh.

1.1.1 Mesh internal data structures

Node positions and connectivity

We use a variation on the standard `X` and `IX` arrays for representing the node positions and connectivities. We require that each node have `ndm` coordinates (so `X` is an `ndm`-by-`numnp` array), but we allow a variable number of nodes per element. In particular, we have in mind the possibility that macro-elements connected to many nodes may be stored in the same structure as ordinary elements connected to relatively few nodes. Therefore, we represent the element connectivity using two arrays: `NIX` and `IX`. `NIX` is an array of `numelt + 1` numbers such that element `i` is connected to nodes `IX[j]`, where `NIX[i] <= j < NIX[i+1]`.

In addition to the element-to-node connectivity structure represented by `IX` and `NIX`, there is also a node-to-element connectivity data structure represented by `IN` and `NIN`. This reverse map currently is not used for very much.

Element data

Each element is associated with an element object which is responsible for evaluating the local stiffness and residual, marking which degrees of freedom it uses, etc. These associations are stored in the `etypes` array. In addition, there is an array `etypes_owned` which is used to store what element objects the mesh has ownership of (i.e. which elements should be destroyed when the mesh object is destroyed). Not all elements in `etypes_owned` need be referenced from `etypes`, and vice-versa.

Variable assignments

There are three types of variables in HiQLab: *nodal* variables associated with a particular node; *branch* variables associated with a particular element; and *global* variables which are associated with global shape functions. Each of these variables is associated with a position in the various global arrays: all the nodal variables first, then all the branch variables, then all the globals. The index space is divvied up as follows:

- `i+j*maxndf` – node `j`, variable `i`
- `NB[j] <= k < NB[j+1]` – element `j` branch variables

- `NG[0] <= k < NG[1]` – global variables

The `ID` array maps this full set of variable indices into a reduced set of indices. Only active variables are assigned valid reduced indices; variables subject to displacement boundary conditions are assigned negative indices.

Boundary conditions

Each variable in the system can be assigned a displacement boundary condition (`BC[k] = 'u'`), a force boundary condition (`BC[k] = 'f'`), or no boundary condition (`BC[k] = ' '`). For those variables subject to boundary conditions, the corresponding nodal boundary values are stored in the `BV` and `BVi` arrays.

System state

The `U`, `V`, and `A` arrays contain the unreduced displacement, velocity, and acceleration variables. There are parallel arrays `Ui`, `Vi`, and `Ai` to contain the imaginary parts. The `F` and `Fi` variables contain the force residual vector.

The force variables associated with global shape functions must be computed in a separate pass after the ordinary (nodal and branch) forces. Right now, the code to deal with this is scattered somewhat haphazardly through the mesh implementation. A possibly better alternative would be to introduce an explicit representation of the global shape function matrix G, so that forming the full residual would involve an ordinary sparse matvec and forming the full tangent would involve an ordinary sparse matrix triple product.

History database

HiQLab provides space for history data, though at present none of the elements use it. The history database consists of three vectors. The index vector `NH` tracks which variables are allocated to which element; element `j` is assigned slots `NH[j] <= k < NH[j+1]`. The history data itself is stored in the vectors `HIST` and `HISTi`. The history vectors actually each have storage for two sets of variables, one for reading and one for writing. Which set of variables represents the saved state is represented by `which_hist` (which is 0 if the first half of the database is the saved state, 1 if the second half is the saved state).

Scaling data

HiQLab keeps two vectors of characteristic scales: `D1` represents the characteristic scales of the input vectors, and `D2` represents the characteristic scales of the output vectors. Therefore, if $\tilde{u}$, $\tilde{F}$, and $\tilde{K}$ are displacements, forces, and stiffnesses in some system of units, $u = D_1^{-1}\tilde{u}$, $F = D_2^{-1}\tilde{F}$, and $K = D_2^{-1}\tilde{K}D_1$ represent nondimensionalized versions of the displacements, forces, and stiffnesses. By keeping this scaling data in a single vector, we get the numerical advantages of nondimensionalization (in terms of sensitivity of results, ease of choosing convergence criteria, etc), but we can keep looking at numbers with units when numbers with units make sense.

Lua interpreter

For a typical mesh object, we will have an associated Lua interpreter which contains callback functions to set the boundary conditions and scaling vectors in an appropriate manner. The mesh may or may not also have ownership of the Lua interpreter. If the mesh does have ownership of the interpreter, then the mesh object destructor will close the interpreter. Thus mesh objects created from MATLAB or some other interface would typically have ownership of their Lua interpreter, while mesh objects created from the Lua interface might well use the Lua interpreter associated with the environment in which they were created.

1.1.2 Accessor functions

The variables in our system can be real or complex, and they can be associated with nodal variables, branch variables, or global variables. Rather than write many tiny accessor functions by hand, we write a few macros to produce these accessor functions.

```
#define defq_z_accessor1(name, array, index) \
   double&  name   (int i) { return array   [index]; } \
   double&  name##i(int i) { return array##i[index]; } \
   dcomplex name##z(int i) { return dcomplex(name(i), name##i(i));  }

#define defq_z_accessor2(name, array, index) \
   double&  name   (int i, int j) { return array   [index]; } \
   double&  name##i(int i, int j) { return array##i[index]; } \
   dcomplex name##z(int i, int j) { return dcomplex(name(i,j), name##i(i,j)); }

#define defq_z_accessor(name, array, index) \
   defq_z_accessor1(name, array, i) \
   defq_z_accessor2(name, array, index)

#define defq_meshz_accessor(u, U) \
   defq_z_accessor(u, U, j*maxndf+i) \
   defq_z_accessor(branch##u, U, NB[j]+i) \
   defq_z_accessor1(global##u, U, NG[0]+i)

#define defq_mesh_accessor(type, name, array) \
   type& name        (int i)        { return array[i];          } \
   type& name        (int i, int j) { return array[j*maxndf+i]; } \
   type& branch##name(int i, int j) { return array[NB[j]+i];    } \
   type& global##name(int i)        { return array[NG[0]+i];    }
```

1.1.3 Mesh declarations

```
class Mesh {
 public:

    Mesh(int ndm, int maxnen = 0, int maxndf = 0);
    virtual ~Mesh();

    /** Get Lua state used by (not necessarily owned by) system */
    lua_State* get_lua()          { return L; }
    void set_lua(lua_State* argL) { L = argL; }

    /** Get out a named scale from the Lua state */
    double get_scale(const char* name);

    /** Set scaling vector D1 and D2 */
    void clear_scaling_vector();
```

```
void set_nodal_u_scale(int i, double s);
void set_nodal_f_scale(int i, double s);
void set_scaling_vector();

/** Get scaling vector D1 or D2 */
void get_scaling_vector(double* su, double* sf, int is_reduced = 1);

/** Add node(s), element(s), or global(s) to the mesh directly.
 *  In each case, return the identifier of the first item created.
 */
int add_node(double* x, int n=1);
int add_element(int* e, Element* etype, int nen, int n=1);
int add_global(int n);

/** Construct a Cartesian block. */
void add_block(double* x1, double* x2, int* m,
               Element* etype, int order=1);
void add_block(double x1, double x2, int m,
               Element* etype, int order=1);
void add_block(double x1, double y1,
               double x2, double y2,
               int nx, int ny,
               Element* etype, int order=1);
void add_block(double x1, double y1, double z1,
               double x2, double y2, double z2,
               int nx, int ny, int nz,
               Element* etype, int order=1);

/** Tie mesh nodes in a range.  Only affects the element connectivity
 *  array; tied nodes are not removed from the mesh data structure.
 */
void tie(double tol, int start=-1, int end=-1);

/** Remove any nodes which are not referenced by the IX array.
 *  Should be called immediately after tie().
 */
void remove_unused_nodes();

/** Get a clean mesh which has no redundant nodes, i.e. mesh
 *  which has no nodes with dof labeled NULL(-1)
 */
Mesh* get_clean_mesh();

/** Allocate space for global arrays and assign initial index map. */
void initialize();

/** Reassign the index map. */
int assign_ids();

/** Assemble tangent matrix structure. */
void assemble_struct(QStructAssembler* K);
```

```
/** Assemble global mass and stiffness matrices. */
void assemble_dR(QAssembler* K, double cx=1, double cv=0, double ca=0);

/** Assemble global reaction vector (goes into f) */
void assemble_R();

/** Assemble time-averaged energy flux at node points */
void mean_power(double* E);

/** Clear current boundary condition */
void clear_bc();

/** Use Lua to set boundary conditions */
void set_bc         (const char* funcname);
void set_globals_bc (const char* funcname);
void set_elements_bc(const char* funcname);

/** Apply boundary conditions based on saved Lua functions */
void apply_bc();

/** Build alternate vectors (for drive or sense) using Lua */
void get_vector(const char* fname, double* v, int is_reduced = 1);

/** Evaluate a Lua function at every nodal point */
void get_lua_fields(const char* funcname, int n, double* fout);

/** Evaluate global shape functions via Lua funcs */
double shapeg(int i, int j);
void   shapeg(double* v, double c, int j,
              int is_reduced, int vstride = 1);

/** Copy i*omega*u and -omega^2*u into a and v */
void make_harmonic(dcomplex omega);
void make_harmonic(double omega_r, double omega_i = 0) {
    make_harmonic(dcomplex(omega_r, omega_i));
}

int numnp()         { return X.size()  / ndm;    }
int numelt()        { return etypes.size();      }
int numglobals()    { return NG[1]-NG[0];        }
int get_numid()     { return numid;              }
int get_ndm()       { return ndm;                }
int get_ndf()       { return maxndf;             }
int get_nen()       { return maxnen;             }
int nbranch_id()    { return NB[numelt()]-NB[0]; }

int       get_nen    (int i) { return NIX[i+1]-NIX[i]; }
int       get_nne    (int i) { return NIN[i+1]-NIN[i]; }
int       nbranch_id (int i) { return NB[i+1] -NB[i];  }
int       nhist_id   (int i) { return NH[i+1] -NH[i];  }
```

```
Element*& etype      (int i) { return etypes[i];       }
int&      nbranch    (int i) { return NB[i];           }
int&      nhist      (int i) { return NH[i];           }

int&      ix(int i, int j) { return IX[NIX[j] + i]; }
int&      in(int i, int j) { return IN[NIN[j] + i]; }
double&   x (int i, int j) { return X[j*ndm + i];   }

/** Get the global array indices for different variables */
int inode  (int i, int j) { return j*maxndf+i; }
int ibranch(int i, int j) { return NB[j]+i;    }
int iglobal(int i)        { return NG[0]+i;    }

/** Get and set the ID and BC arrays */
defq_mesh_accessor( int,  id, ID )
defq_mesh_accessor( char, bc, BC )

/** Get and set the boundary values, u, v, a, and f arrays */
defq_meshz_accessor( bv,  BV  )
defq_meshz_accessor( u ,  U   )
defq_meshz_accessor( v ,  V   )
defq_meshz_accessor( a ,  A   )
defq_meshz_accessor( f ,  F   )

void set_u (double* u = NULL, double* v = NULL, double* a = NULL);
void set_ui(double* u = NULL, double* v = NULL, double* a = NULL);
void set_uz(double* u = NULL, double* ui = NULL,
            double* v = NULL, double* vi = NULL,
            double* a = NULL, double* ai = NULL);
void set_uz(dcomplex* u = NULL, dcomplex* v = NULL, dcomplex* a = NULL);

void set_f (double* f = NULL);
void set_fi(double* f = NULL);
void set_fz(double* fr = NULL, double* fi = NULL);
void set_fz(dcomplex* f = NULL);

void get_u (double* u = NULL, double* v = NULL, double* a = NULL);
void get_ui(double* u = NULL, double* v = NULL, double* a = NULL);
void get_uz(double* u = NULL, double* ui = NULL,
            double* v = NULL, double* vi = NULL,
            double* a = NULL, double* ai = NULL);
void get_uz(dcomplex* u = NULL, dcomplex* v = NULL, dcomplex* a = NULL);

void get_f (double* f = NULL);
void get_fi(double* f = NULL);
void get_fz(double* fr = NULL, double* fi = NULL);
void get_fz(dcomplex* f = NULL);

/** Access element history entries */
double get_hist (int i, int j) { return HIST [hist_index(i,j,0)]; }
double get_histi(int i, int j) { return HISTi[hist_index(i,j,0)]; }
```

```
    double put_hist (int i, int j, double x) { HIST [hist_index(i,j,1)] = x; }
    double put_histi(int i, int j, double x) { HISTi[hist_index(i,j,1)] = x; }
    void swap_hist() {
        which_hist = (which_hist+1)%2;
    }

    /** Get and set scaling parameters for primary and secondary variables */
    defq_mesh_accessor( double, d1,  D1  )
    defq_mesh_accessor( double, d2,  D2  )

    Element* own(Element* e);
    void     own(lua_State* L);

    void lua_getfield(const char* name);
    void lua_setfield(const char* name);
    int  lua_method(const char* name, int nargin, int nargout);


};
```

1.1.4 Mesh construction, destruction, and memory management

Most of the data in the mesh object lives in C++ containers, which are automatically managed. An exception to this is the element types and the Lua state. We require that the user explicitly hand over ownership of these types for two reasons. First, it may not always make sense for the mesh object to own these objects – for example, when using the Lua front-end, the Lua interpreter will typically outlive the mesh! Second, the mesh object isn't responsible for constructing these objects, so unless there is some explicit registration mechanism, the mesh will not necessarily know that the objects exist.

It is a checked error to try to unassign or reassign ownership of the Lua state after the first assignment.

```
Mesh::Mesh(int ndm, int maxnen, int maxndf) :
    ndm(ndm), maxnen(maxnen), maxndf(maxndf), numid(0), which_hist(0),
    lua_owned(NULL), L(NULL)
{
    NG[0] = 0;
    NG[1] = 0;
    NIX.push_back(0);
}


Mesh::~Mesh()
{
    for (unsigned i = 0; i < etypes_owned.size(); ++i)
        if (etypes_owned[i])
            delete etypes_owned[i];
    if (lua_owned)
        lua_close(lua_owned);
}
```

```
Element* Mesh::own(Element* e)
{
    etypes_owned.push_back(e);
    return e;
}


void Mesh::own(lua_State* L)
{
    assert(lua_owned == NULL);
    assert(L != NULL);
    lua_owned = L;
}
```

1.1.5 Mesh scale management

We keep two distinct sets of scaling information associated with a mesh.

First, there is the abstract scaling information associated with different variable types (temperatures, displacements, etc). The user provides access to these scales by associating a `get_scale` method with the Lua mesh object. This method maps a string argument to a numeric scaling value. If the method isn't there, or doesn't return anything, we return a default scale value of 1.

Second, there are scales associated with each primary and dual variable in the system (including element and global variables). These scales are stored in the `d1` and `d2` arrays, respectively. The default scales are all ones, but the defaults would typically be overridden by the `set_scaling_vector` method. This method in turn calls `set_nodal_u_scale` and `set_nodal_f_scale` to set all the scales associated with a particular slot in the nodal variable array (since a given slot is associated with a particular field – see the element interface description).

```
double Mesh::get_scale(const char* name)
{
    CHECK_LUA 1;
    lua_pushstring(L, name);
    double retval = 1;
    if (lua_method("get_scale", 1, 1) == 0) {
        retval = lua_tonumber(L,-1);
        lua_pop(L,1);
    }
    return retval;
}


void Mesh::clear_scaling_vector()
{
    int N = ID.size();
    for (int i = 0; i < N; ++i) {
        d1(i) = 1;
        d2(i) = 1;
```

```
    }
}


void Mesh::set_nodal_u_scale(int i, double s)
{
    for (int j = 0; j < numnp(); ++j)
        d1(i,j) = s;
}


void Mesh::set_nodal_f_scale(int i, double s)
{
    for (int j = 0; j < numnp(); ++j)
        d2(i,j) = s;
}


void Mesh::set_scaling_vector()
{
    CHECK_LUA;
    lua_method("set_scaling_vector", 0, 0);
}


void Mesh::get_scaling_vector(double* su, double* sf, int is_reduced)
{
    int N = (is_reduced ? ID.size() : numnp()*maxndf );
    for (unsigned i = 0; i < (unsigned) N; ++i) {
        int k = (is_reduced ? id(i) : i);
        if (k >= 0) {
            if (su) su[k] = D1[i];
            if (sf) sf[k] = D2[i];
        }
    }
}
```

1.1.6 Basic mesh generation

```
int Mesh::add_node(double* x, int n)
{
    int id = X.size() / ndm;
    for (int i = 0; i < n*ndm; ++i)
        X.push_back(x[i]);
    return id;
}
```

```
int Mesh::add_global(int n)
{
    int id = NG[1];
    NG[1] += n;
    return id;
}


int Mesh::add_element(int* e, Element* etype, int nen, int n)
{
    int id = etypes.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < nen; ++j)
            IX.push_back(e[i*nen+j]);
        end_add_element(etype);
    }
    return id;
}


void Mesh::end_add_element(Element* etype)
{
    NIX.push_back(IX.size());
    etypes.push_back(etype);
    NB.push_back(0);
    NH.push_back(0);
    if (etype)
        etype->initialize(this, etypes.size()-1);
}
```

1.1.7 Block mesh generation

The block mesh generators produce rectilinear coordinate-aligned block meshes, which can subsequently be turned into more interesting shape by mapping and tying operations. For each dimension, we specify the coordinate range ($[x_1, x_2]$) and the number of *nodes* (not elements) in that direction. It is a checked error to try to specify a number of nodes that doesn't yield an integer number of elements.

```
void Mesh::add_block(double* x1, double* x2, int* m,
                     Element* etype, int order)
{
    if (ndm == 1)
        add_block(x1[0], x1[0], m[0], etype, order);

    else if (ndm == 2)
        add_block(x1[0], x1[1],
                  x2[0], x2[1],
                  m[0],  m[1], etype, order);
```

```
    else if (ndm == 3)
        add_block(x1[0], x1[1], x1[2],
                  x2[0], x2[1], x2[2],
                  m[0],  m[1],  m[2], etype, order);
}


void Mesh::add_block(double x1, double x2, int nx,
                     Element* etype, int order)
{
    assert( nx > 0 && (nx-1)%order == 0 );

    int start_node = numnp();
    for (int ix = 0; ix < nx; ++ix) {
        double x[1];
        x[0] = ((nx-1-ix)*x1 + ix*x2) / (nx-1);
        add_node(x);
    }

    for (int ix = 0; ix < nx-order; ix += order) {
        for (int jx = 0; jx < order+1; ++jx)
            IX.push_back( (ix+jx) + start_node );
        end_add_element(etype);
    }
}


void Mesh::add_block(double x1, double y1,
                     double x2, double y2,
                     int nx, int ny,
                     Element* etype, int order)
{
    assert( nx > 0 && (nx-1)%order == 0 );
    assert( ny > 0 && (ny-1)%order == 0 );

    int start_node = numnp();
    for (int ix = 0; ix < nx; ++ix) {
        for (int iy = 0; iy < ny; ++iy) {
            double x[2];
            x[0] = ((nx-1-ix)*x1 + ix*x2) / (nx-1);
            x[1] = ((ny-1-iy)*y1 + iy*y2) / (ny-1);
            add_node(x);
        }
    }

    for (int ix = 0; ix < nx-order; ix += order)
        for (int iy = 0; iy < ny-order; iy += order) {
            for (int jx = 0; jx < order+1; ++jx)
                for (int jy = 0; jy < order+1; ++jy)
                    IX.push_back( (iy+jy) + ny*(ix+jx) + start_node );
            end_add_element(etype);
```

```
        }
}


void Mesh::add_block(double x1, double y1, double z1,
                     double x2, double y2, double z2,
                     int nx, int ny, int nz,
                     Element* etype, int order)
{
    assert( nx > 0 && (nx-1)%order == 0 );
    assert( ny > 0 && (ny-1)%order == 0 );
    assert( nz > 0 && (nz-1)%order == 0 );

    int start_node = numnp();

    for (int ix = 0; ix < nx; ++ix) {
        for (int iy = 0; iy < ny; ++iy) {
            for (int iz = 0; iz < nz; ++iz) {
                double x[3];
                x[0] = ((nx-1-ix)*x1 + ix*x2) / (nx-1);
                x[1] = ((ny-1-iy)*y1 + iy*y2) / (ny-1);
                x[2] = ((nz-1-iz)*z1 + iz*z2) / (nz-1);
                add_node(x);
            }
        }
    }

    for (int ix = 0; ix < nx-order; ix += order)
        for (int iy = 0; iy < ny-order; iy += order)
            for (int iz = 0; iz < nz-order; iz += order) {
                for (int jx = 0; jx < order+1; ++jx)
                    for (int jy = 0; jy < order+1; ++jy)
                        for (int jz = 0; jz < order+1; ++jz)
                            IX.push_back( (iz+jz) + nz*((iy+jy) + ny*(ix+jx)) +
                                          start_node );
                end_add_element(etype);
            }
}
```

1.1.8 Tied meshes

We tie meshes together by approximately sorting the coordinates, looking first at x, then y, then z. Nodes that are equivalent under this approximate sort (and therefore appear together in the sorted node list) are tied. For each equivalence class, we pick one representative node, and map all references to other nodes in the class into references to that representative.

The assumption in this algorithm is that all nodes that are supposed to be tied together are within a given tolerance of each other (and all nodes that are not supposed to be tied are farther than that tolerance). It's possible to build a pathological mesh where this assumption fails; for example, if we have nodes at $(0, 0)$, $(0, \epsilon)$, $(0, -\epsilon)$ where ϵ is the tolerance, then the relation we've defined is not transitive, and we don't have an equivalence relation. In this case, things that should probably be tied together might not be. I regard

this as an unchecked blunder on the part of the analyst.

```
class CoordSorter {
public:
    CoordSorter(int ndm, Mesh& mesh, double tol) :
        ndm(ndm), mesh(mesh), tol(tol) {}

    int operator()(int i, int j)
    {
        for (int k = 0; k < ndm; ++k) {
            if (mesh.x(k,i) < mesh.x(k,j)-tol)   return  1;
            if (mesh.x(k,i) > mesh.x(k,j)+tol)   return  0;
        }
        return 0;
    }

    int compare(int i, int j)
    {
        for (int k = 0; k < ndm; ++k) {
            if (mesh.x(k,i) < mesh.x(k,j)-tol)   return  -1;
            if (mesh.x(k,i) > mesh.x(k,j)+tol)   return   1;
        }
        return 0;
    }

private:
    int ndm;
    Mesh& mesh;
    double tol;
};


void Mesh::tie(double tol, int start, int end)
{
    int np = numnp();
    if (start < 0) start = 0;
    if (end   < 0) end   = np;
    int nt = end-start;

    // Set scratch = start:end-1, then sort the indices according
    // to the order of the corresponding nodes.
    //
    CoordSorter sorter(ndm, *this, tol);
    vector<int> scratch(nt);
    for (int i = 0; i < nt; ++i)
        scratch[i] = start+i;
    sort(scratch.begin(), scratch.end(), sorter);

    // map[i] := smallest index belonging to equivalence class of p[i]
    //
```

```
    vector<int> map(np);
    for (int i = 0; i < np; ++i)
        map[i] = i;

    int inext;
    for (int i = 0; i < nt; i = inext) {
        int class_id = scratch[i];
        for (inext = i+1; inext < nt; ++inext) {
             if (sorter.compare(scratch[i], scratch[inext]) != 0)
                 break;
             if (scratch[inext] < scratch[i])
                class_id = scratch[inext];
        }
        for (int j = i; j < inext; ++j)
            map[scratch[j]] = class_id;
    }

    // Apply map to element connectivity array
    //
    for (int i = 0; i < (int) IX.size(); ++i)
        if (IX[i] >= 0)
            IX[i] = map[IX[i]];
}
```

1.1.9 Removing redundant nodes

`get_clean_mesh` creates a new mesh in which any nodes that have no assigned variables are removed. The resulting mesh does not own any storage; it is not initialized; and it does not have access to the Lua interpreter.

`remove_unused_nodes` removes all unused nodes from the mesh, which I think is the functionality we actually wanted. This routine should be called before initialization (or the mesh should be reinitialized immediately afterward).

```
Mesh* Mesh::get_clean_mesh()
{
    // -- Clear boundary conditions
    clear_bc();
    assign_ids();

    // -- Construct new mesh
    Mesh* mesh_n = new Mesh(ndm,maxnen,maxndf);

    // -- Add new nodes
    int numnp_n = 0;
    vector<double> x_n(X.size());
    vector<int>    map_n(numnp());
    for (int i = 0; i < numnp(); ++i) {
        int ifix = 0;
```

```
        for (int j = 0; j < maxndf; ++j)
            ifix += ID[maxndf*i+j];
        if (ifix > -maxndf) {
            map_n[i] = numnp_n;
            for (int k = 0; k < ndm; ++k)
                x_n[numnp_n*ndm+k] = X[i*ndm+k];
            numnp_n++;
        } else
            map_n[i] = -1;
    }
    mesh_n->add_node(&(x_n[0]), numnp_n);

    // -- Add element connectivity
    for (int i = 0; i < numelt(); ++i) {
        int nen = get_nen(i);
        int e_n[nen];
        for (int j = 0; j < nen; ++j)
            e_n[j] = map_n[ix(j,i)];
        mesh_n->add_element(e_n, etypes[i], nen, 1);
    }

    // -- Add globals
    mesh_n->add_global(numglobals());

    // -- Reinitialize current mesh
    apply_bc();
    set_scaling_vector();

    return mesh_n;
}


void Mesh::remove_unused_nodes()
{
    vector<int> map_used(numnp());
    clear(map_used);

    // Determine which nodes are used by IX
    for (int i = 0; i < IX.size(); ++i)
        map_used[IX[i]] = 1;

    // Compress out unused nodes, building an index map as we go
    int num_used = 0;
    for (int i = 0; i < numnp(); ++i) {
        if (map_used[i]) {
            for (int j = 0; j < ndm; ++j)
                x(num_used,i) = x(j,i);
            map_used[i] = num_used++;
        }
    }
    X.resize(ndm*num_used);
```

```
    // Remap the IX array
    for (int i = 0; i < IX.size(); ++i)
        IX[i] = map_used[IX[i]];
}
```

1.1.10 Initialization

Mesh initialization occurs after the geometry of the mesh has been established and before any analysis takes place. This is where we decide how big all the major arrays will be, and set up the index structures that are used for all of our assembly operations.

```
void Mesh::initialize()
{
    build_reverse_map();              // Build node-to-element map
    initialize_sizes();               // Figure out maxnen and maxndf
    int M  = initialize_indexing();   // Assign branch and global dofs
    int MH = initialize_history();    // Assign history storage
    initialize_arrays(M, MH);         // Allocate storage arrays

    // -- Assign IDs, BCs, and scales
    assign_ids();
    apply_bc();
    set_scaling_vector();
}


void Mesh::initialize_sizes()
{
    for (int i = 0; i < etypes.size(); ++i) {
        int nen = get_nen(i);
        if (maxnen < nen)
            maxnen = nen;
    }
    for (int i = 0; i < etypes.size(); ++i) {
        if (etypes[i]) {
            int elt_maxndf = etypes[i]->max_id_slot() + 1;
            if (maxndf < elt_maxndf)
                maxndf = elt_maxndf;
        }
    }
}


void Mesh::build_reverse_map()
{
    int N = numnp();
    int M = etypes.size();
```

```
    clear(NIN,N+1);

    // -- NIN[i] := elements for node i
    for (unsigned i = 0; i < IX.size(); ++i)
        if (IX[i] >= 0)
            NIN[IX[i]]++;

    // -- NIN[i] := elements for nodes 0 .. i
    for (int i = 1; i < N+1; ++i)
        NIN[i] += NIN[i-1];

    // -- Build reverse map via bucket sort
    clear(IN, NIN[N]);
    for (int i = 0; i < M; ++i) {
        for (int j = NIX[i]; j < NIX[i+1]; ++j) {
            int node = IX[j];
            int slot = --NIN[node];
            IN[slot] = i;
        }
    }
}


int Mesh::initialize_indexing()
{
    // -- Assign index ranges for branch dofs
    int M = numnp()*maxndf;
    for (int i = 0; i < etypes.size(); ++i) {
        int start_M = M;
        M += NB[i];
        NB[i] = start_M;
    }
    NB.push_back(M);

    // -- Assign index ranges for global dofs
    NG[0] += M;
    NG[1] += M;
    M = NG[1];

    return M;
}


int Mesh::initialize_history()
{
    int MH = 0;
    for (int i = 0; i < etypes.size(); ++i) {
        int start_MH = MH;
        MH += NH[i];
        NH[i] = start_MH;
    }
```

```
    NH.push_back(MH);
    return MH;
}


void Mesh::initialize_arrays(int M, int MH)
{
    clear(U, Ui, M);
    clear(V, Vi, M);
    clear(A, Ai, M);
    clear(F, Fi, M);
    clear(BV,BVi,M);
    clear(BC,M);
    clear(ID,M);
    clear(D1,M);
    clear(D2,M);
    clear(HIST, HISTi, 2*MH);
}
```

1.1.11 Assigning reduced indices

The `assign_ids` method sets up the map from the full index set to the reduced index set. Only the active variables (variables that someone cares about that are not subject to a displacement BC) are assigned valid reduced indices. This method is called at initialization, but it can also be called at subsequent phases in the analysis if the user wishes to change the boundary conditions.

```
int Mesh::assign_ids()
{
    int N = etypes.size();   // Number of elements
    int M = ID.size();       // Number of potential dofs

    // -- Mark active dofs
    clear(ID);
    for (int i = 0; i < N; ++i) {
        if (etypes[i])
            etypes[i]->assign_ids(this, i);
    }
    for (int i = NG[0]; i < NG[1]; ++i)
        ID[i] = 1;

    // -- Assign IDs to nodal dofs
    numid = 0;
    for (int i = 0; i < M; ++i) {
        if (ID[i] != 0 && BC[i] != 'u')
            ID[i] = numid++;
        else
            ID[i] = -1;
    }
```

```
    return numid;
}
```

1.1.12 Assembly loops

The `assemble_dR` routine assembles a linear combination of mass, damping, and stiffness matrices. The `assemble_struct` routine is used to accumulate the nonzero structure of the matrix for pre-allocation.

The `assemble_R` routine assembles the residual. It's a little quirky because it not only assembles the residual contributions from the elements, but it also computes the components of the extended residual that are associated with global shape functions. That is, we compute something like

$$R_g = G^T R_0$$

where R_g is the residual associated with the global shapes described by the columns of G, and R_0 is the unreduced residual without the global shape functions.

The `mean_power` loop assembles a lumped L^2 projection of the mean power as computed at each Gauss point. It's a sort of quirky routine – why just the mean power, and not also other Gauss point quantities like the stress? We should probably rethink this routine t some point.

```
void Mesh::assemble_struct(QStructAssembler* K)
{
    for (unsigned i = 0; i < etypes.size(); ++i)
        if (etypes[i])
            etypes[i]->assemble_struct(this, i, K);
}


void Mesh::assemble_dR(QAssembler* K, double cx, double cv, double ca)
{
    for (unsigned i = 0; i < etypes.size(); ++i)
        if (etypes[i])
            etypes[i]->assemble_dR(this, i, K, cx, cv, ca);
}


void Mesh::assemble_R()
{
    set_fz((dcomplex*) NULL);

    for (unsigned i = 0; i < etypes.size(); ++i)
        if (etypes[i])
            etypes[i]->assemble_R(this, i);

    if (numglobals() > 0) {
        for (int j = 0; j < numglobals(); ++j) {
            vector<double> scratch;
            scratch.resize(ID.size());
```

```
            shapeg(&(scratch[0]), 1, j, 0);
            double fgjr = 0;
            double fgji = 0;
            for (unsigned k = 0; k < ID.size(); ++k) {
                fgjr += F [k]*scratch[k];
                fgji += Fi[k]*scratch[k];
            }
            globalf (j) += fgjr;
            globalfi(j) += fgji;
        }
    }
}


void Mesh::mean_power(double* E)
{
    FieldEvalPower mean_power_func;
    vector<double> Mdiag(numnp());
    memset(&(Mdiag[0]), 0, numnp()     * sizeof(double));
    memset(E,           0, numnp()*ndm * sizeof(double));
    for (unsigned i = 0; i < etypes.size(); ++i)
        if (etypes[i])
            etypes[i]->project_L2(this, i, ndm, &(Mdiag[0]),
                                  E, mean_power_func);
    for (int i = 0; i < numnp(); ++i)
        if (Mdiag[i])
            for (int j = 0; j < ndm; ++j)
                E[i*ndm+j] /= Mdiag[i];
}
```

1.1.13 Boundary condition manipulations

The boundary conditions are specified through the Lua fields `bcfunc`, `shapegbc`, and `elementsbc`. This is a little bit of a relic, though, as all the actual work of setting up the boundary conditions is now done through the `form_bcs` function. Since nothing in the C++ code cares directly about anything but `form_bcs`, why do we still have C++ setters for these other functions?

```
void Mesh::set_bc(const char* funcname)
{
    lua_getglobal(L, funcname);
    lua_setfield("bcfunc");
}


void Mesh::set_globals_bc(const char* funcname)
{
    lua_getglobal(L, funcname);
    lua_setfield("shapegbc");
```

```
}


void Mesh::set_elements_bc(const char* funcname)
{
    lua_getglobal(L, funcname);
    lua_setfield("elementsbc");
}


void Mesh::apply_bc()
{
    apply_lua_bc();

    // Reassign IDs
    assign_ids();

    // Copy boundary data into place
    set_bc_u(U, BV );
    set_bc_u(Ui,BVi);
    set_bc_f(F, BV );
    set_bc_f(Fi,BVi);

    // Update node disps that are linked to globals
    for (int j = 0; j < numglobals(); ++j) {
        shapeg(&(U [0]), U [NG[0]+j], j, 0);
        shapeg(&(Ui[0]), Ui[NG[0]+j], j, 0);
    }
}


void Mesh::apply_lua_bc()
{
    CHECK_LUA;
    lua_method("form_bcs", 0, 0);
}


void Mesh::clear_bc()
{
    clear(U,  Ui);
    clear(V,  Vi);
    clear(A,  Ai);
    clear(F,  Fi);
    clear(BV, BVi);
    clear(BC);
    clear(ID);
}
```

1.1.14 Setting up time-harmonic analysis

For a time-harmonic motion with frequency ω, we have $\hat{v} = i\omega\hat{u}$ and $\hat{a} = i\omega\hat{v}$. The `make_harmonic` motion sets u and a based on these relations.

```
void Mesh::make_harmonic(dcomplex omega)
{
    dcomplex iomega = dcomplex(0,1)*omega;
    for (unsigned i = 0; i < U.size(); ++i) {
        dcomplex uu(U[i], Ui[i]);
        dcomplex vv = uu*iomega;
        dcomplex aa = vv*iomega;

        V [i] = real(vv);
        Vi[i] = imag(vv);
        A [i] = real(aa);
        Ai[i] = imag(aa);
    }
}
```

1.1.15 Lua accessors

We use Lua functions to define the drive and sense vectors used in transfer function evaluation, auxiliary fields that we might want to plot, and the global shape functions.

```
void Mesh::get_vector(const char* fname, double* v, int is_reduced)
{
    CHECK_LUA;
    QVecAssembler va(v, NULL, this, is_reduced);
    lua_getglobal(L, fname);
    tolua_pushusertype(L, this, "Mesh");
    tolua_pushusertype(L, &va, "QVecAssembler");
    lua_pcall2(L, 2, 0);
}
```

```
void Mesh::get_lua_fields(const char* funcname, int n, double* fout)
{
    lua_pushstring(L, funcname);
    lua_gettable(L, LUA_GLOBALSINDEX);
    if (!lua_isfunction(L,-1)) {
        printf("Error in get_lua_fields: [%s]\n", funcname);
        lua_pop(L,1);
        return;
    }

    int func = lua_gettop(L);
```

```
    int N    = numnp();
    memset(fout, 0, n*N * sizeof(double));

    for (int j = 0; j < N; ++j) {
        int t = lua_gettop(L)+1;
        lua_pushvalue(L, func);
        for (int i = 0; i < ndm; ++i)
            lua_pushnumber(L, x(i,j));
        if (lua_pcall2(L, ndm, n) == 0) {
            for (int i = 0; i < n && i+t < (int) lua_gettop(L)+1; ++i)
                fout[j*n+i] = lua_tonumber(L, i+t);
        }
        lua_pop(L, n);
    }
    lua_pop(L,1);
}


double Mesh::shapeg(int i, int j)
{
    CHECK_LUA 0;
    lua_pushnumber(L, i);
    lua_pushnumber(L, j);
    double retval = 0;
    if (lua_method("get_shapeg", 2, 1) == 0) {
        retval = lua_tonumber(L,-1);
        lua_pop(L,1);
    }
    return retval;
}


void Mesh::shapeg(double* v, double c, int j, int is_reduced, int vstride)
{
    if (!L || c == 0)
        return;

    QVecAssembler va(v, NULL, this, is_reduced, vstride);
    tolua_pushusertype(L, &va, "QVecAssembler");
    lua_pushnumber(L, c);
    lua_pushnumber(L, j);
    lua_method("get_shapeg_vec", 3, 0);
}
```

1.1.16 Setting and getting `U`, `V`, `A`, `F`

There are a lot of variants of setting and getting the system state vectors, simply because they're all potentially complex-valued. These functions all copy to/from reduced vectors, with part of the data merged in from the boundary value arrays.

1.1.17 Lua support methods

These routines support access to the Lua side of the mesh object. The `lua_getfield`, `lua_setfield`, and `lua_method` methods let us access fields and methods associated with the Lua object that represents the mesh from tolua.

```
void Mesh::lua_getfield(const char* name)
{
    tolua_pushusertype(L, this, "Mesh");
    lua_pushstring(L, name);
    lua_gettable(L, -2);
    lua_remove(L, -2);
}


void Mesh::lua_setfield(const char* name)
{
    tolua_pushusertype(L, this, "Mesh");
    lua_insert(L,-2);
    lua_pushstring(L, name);
    lua_insert(L,-2);
    lua_settable(L,-3);
    lua_pop(L,1);
}


int Mesh::lua_method(const char* name, int nargin, int nargout)
{
    lua_getfield(name);
    if (!lua_isfunction(L,-1)) {
        lua_pop(L,nargin+1);
        return -1;
    }
    lua_insert(L,-1-nargin);
    tolua_pushusertype(L, this, "Mesh");
    lua_insert(L,-1-nargin);
    return lua_pcall2(L, nargin+1, nargout);
}
```

1.2 Element interface

The `Element` class in HiQLab really refers to an element type. Individual elements in the mesh are not represented by separate objects; rather, we use a "flyweight" pattern in which characteristics of generic element types are stored inside an element object, and characteristics of particular instances are stored externally (in the mesh object). This external information is passed to the element methods as a pointer to the mesh and an element identifier. In order to do element-by-element initialization and assembly tasks, then, we loop over each element in the mesh and make calls of the form `mesh->etypes[eltid]->method(mesh, eltid, ...)`.

1.2.1 Variable slots

Assembling element contributions into a global system requires that all elements share a common idea of what nodal variable number is assigned to what type of variable. The variable slot mechanism provides a way to reconcile the global assignment of nodal variable numbers with a local assignment specific to each element type.

For example, suppose we wanted to perform a simulation that combined mechanical, electrical, and thermal calculations. We might decide at the global level that variables 0-2 at each node correspond to displacements, variable 3 corresponds to electrostatic potential, and variable 4 corresponds to temperature. But a thermomechanical element would not know about electrostatic potential; from the perspective of that element, perhaps variables 0-2 should correspond to displacement and variable 3 should correspond to temperature. In this case, the slot array for the coupling element would look like {`0, 1, 2, 4`}.

1.2.2 Element declarations

```
class Element {
public:
    Element(int nslots);
    virtual ~Element();

    /** Initialize element eltid of the mesh.  Allocate element vars. */
    virtual void initialize(Mesh* mesh, int eltid);

    /** Mark element IDs as active */
    virtual void assign_ids(Mesh* mesh, int eltid);

    /** Assemble structure */
    virtual void assemble_struct(Mesh* mesh, int eltid, QStructAssembler* K);

    /** Assemble element matrices */
    virtual void assemble_dR(Mesh* mesh, int eltid, QAssembler* K,
                             double cx=1, double cv=0, double ca=0);

    /** Assemble element residual */
    virtual void assemble_R(Mesh* mesh, int eltid);

    /** Compute the (lumped) L2 projection of a field defined
     *  at Gauss nodes.  Output is Mdiag += sum(integral N*N') and
     *  xfields += integral N*xfunc.
     */
    virtual void project_L2(Mesh* mesh, int eltid, int nfields,
                            double* Mdiag, double* xfields,
                            FieldEval& xfunc);

    /** Compute the stresses for an element at parent coordinates X.
     *  Returns an incremented stress pointer.  Use two calls to
     *  compute the stress associated with a complex displacement.
     */
    virtual double* stress(Mesh* mesh, int eltid, double* X, double* stress);

    /** Compute the time-averaged energy flux for an element at parent
```

```
     *  coordinates X.  Returns an incremented pointer.
     */
    virtual double* mean_power(Mesh* mesh, int eltid, double* X, double* EX);

    /** Read and write the id_slot map
     */
    int  num_id_slots()        { return id_slots.size();  }
    int  id_slot(int i) const  { return id_slots[i]; }
    int& id_slot(int i)        { return id_slots[i]; }
    void id_slot(int i, int x) { id_slots[i] = x;   }
    int  max_id_slot();

protected:
    std::vector<int> id_slots;    // id_slots[i] = slot for ith local var

    /** Default initializer -- just marks the mesh ID array. */
    void assign_ids_default(Mesh* mesh, int elt);

    void set_local_arrays(Mesh* mesh, int eltid,
                          int* id, double* nodex, int ndm);
    void set_local_id(Mesh* mesh, int eltid, int* id);
    static void set_local_x(Mesh* mesh, int eltid, double* nx, int ndm);

    void set_local_u (Mesh* mesh, int eltid, double*   nu);
    void set_local_ui(Mesh* mesh, int eltid, double*   nu);
    void set_local_u (Mesh* mesh, int eltid, dcomplex* nu);
    void set_local_v (Mesh* mesh, int eltid, double*   nv);
    void set_local_vi(Mesh* mesh, int eltid, double*   nv);
    void set_local_v (Mesh* mesh, int eltid, dcomplex* nv);
    void set_local_a (Mesh* mesh, int eltid, double*   na);
    void set_local_ai(Mesh* mesh, int eltid, double*   na);
    void set_local_a (Mesh* mesh, int eltid, dcomplex* na);

    void add_local_f (Mesh* mesh, int eltid, double*   nodef1);
    void add_local_f (Mesh* mesh, int eltid, dcomplex* nodef1);
};
```

1.2.3 Element construction and destruction

The only internal data structure common to all element types is the variable slot map. By default, we initialize this map to an appropriately-sized identity. We defer to a higher-level routine (typically in the Lua code) the coordination of how slots should be assigned in a multiphysics problem

Because this is a base class, we need a virtual destructor. Said destructor doesn't need to do anything, though.

```
#define ME Element


ME::ME(int nslots) :
    id_slots(nslots)
```

```
{
    for (int i = 0; i < nslots; ++i)
        id_slots[i] = i;
}


ME::~ME()
{
}
```

1.2.4 Element initialization

The element initialization phase is where the element tells HiQLab how many branch variables and history variables it will need. It does this by setting `mesh->nbranch(eltid)` and `mesh->nhist(eltid)`. An element without branch or history variables can use the default (empty) initialization routine.

```
void ME::initialize(Mesh* mesh, int eltid)
{
}
```

1.2.5 Allocating variables

The `assign_ids` method marks the nodal and branch variables that it will use. By default, we assume that this includes all the nodal variables listed in the slots array at each node attached to the element, plus any branch variables that were allocated. Some element types might not use all variables at all nodes, though; for example, a mixed pressure-displacement formulation could have pressure only associated with an internal node.

```
void ME::assign_ids(Mesh* mesh, int eltid)
{
    assign_ids_default(mesh, eltid);
}


void ME::assign_ids_default(Mesh* mesh, int eltid)
{
    int nen = mesh->get_nen(eltid);
    for (int j = 0; j < nen; ++j) {
        int nodeid = mesh->ix(j,eltid);
        for (int i = 0; i < num_id_slots(); ++i)
            mesh->id(id_slots[i], nodeid) = 1;
    }
    int nbranch = mesh->nbranch_id(eltid);
    for (int j = 0; j < nbranch; ++j)
        mesh->branchid(j, eltid) = 1;
}
```

1.2.6 Assembling the residual and tangent

The tangent stiffness looks like

$$K = c_u \frac{\partial R}{\partial u} + c_v \frac{\partial R}{\partial v} + c_a \frac{\partial R}{\partial a}$$

where u, v, and a are the displacement, velocity, and acceleration vectors, respectively. The `assemble_dR` method adds this element's contribution to K. The `assemble_struct` method just assembles the nonzero structure of the tangent stiffness.

The residual vector $R(u, v, a)$ is assembled into the F vector in the mesh object. `assemble_R` adds this element's contribution.

```
void ME::assemble_struct(Mesh* mesh, int eltid, QStructAssembler* K)
{
    int nen = mesh->get_nen(eltid);
    int nslots = num_id_slots();
    int nbranch = mesh->nbranch_id(eltid);
    std::vector<int> id(nen*nslots+nbranch);
    set_local_id(mesh, eltid, &(id[0]));
    K->add(&(id[0]), nen*nslots+nbranch);
}


void ME::assemble_dR(Mesh* mesh, int eltid, QAssembler* K,
                     double cx, double cv, double ca)
{
}


void ME::assemble_R(Mesh* mesh, int eltid)
{
}
```

1.2.7 Lumped L^2 projections

Computing a lumped L^2 projection of a quantity defined at the Gauss points is a two-step procedure. First, call the element `project_L2` method on each element in the mesh. The output of this calculation should be

$$\begin{aligned} \texttt{Mdiag(i)} &= \int_\Omega N_i \, d\Omega \\ \texttt{xfields(i,j)} &= \int_\Omega N_i(x) v_j(x) \, d\Omega \end{aligned}$$

where $v_j(x)$ is the jth scalar component of the fields to be evaluated, and $N_i(x)$ is the shape associated with a unit displcement of variable i.

At the end of the initial loop, we scale `xfields(i,j)` by `Mdiag(i)` in order to get the approximate L^2 projection of the fields onto the nodal shape functions.

```
void ME::project_L2(Mesh* mesh, int eltid, int nfields,
                    double* Mdiag, double* xfields,
                    FieldEval& func)
{
}
```

1.2.8 Gauss point quantities

Right now, there are two types of quantities that we can compute in an element (typically at the Gauss points). The first is the stress components; the second is the mean energy flux in a time-harmonic simulation.

The latter does seem fairly specific, and it's not coded for all elements. This piece of the infrastructure should probably be handled more cleanly.

```
double* ME::stress(Mesh* mesh, int eltid, double* X, double* stress)
{
    return stress;
}
```

```
double* ME::mean_power(Mesh* mesh, int eltid, double* X, double* EX)
{
    return EX;
}
```

1.2.9 Counting the number of ID slots in use

The `max_id_slot` routine returns the last (global) variable index listed in the slot array. Taking the maximum of the `max_id_slot` return values over all elements tells us the maximum number of variables we need to allocate per node. I cannot think of any good reason that a user should call this routine – it only seems useful inside the mesh `initialize` method.

```
int ME::max_id_slot()
{
    int max_slot = 0;
    for (int i = 0; i < num_id_slots(); ++i)
        if (max_slot < id_slots[i])
            max_slot = id_slots[i];
    return max_slot;
}
```

1.2.10 Transferring data from global arrays

For element calculations, we typically will want slices of the global `ID`, `X`, `U`, `V`, and `A` arrays. That's what the `set_local_*` routines do. Of these, the only one that is less than obvious is the `set_local_id` function, which returns in its output array the list of all nodal variables for the element, followed by the list of all branch variables.

```
void ME::set_local_arrays(Mesh* mesh, int eltid,
                          int* id, double* nodex, int ndm)
{
    set_local_x(mesh,  eltid, nodex, ndm);
    set_local_id(mesh, eltid, id);
}


void ME::set_local_id(Mesh* mesh, int eltid, int* id1)
{
    int nen = mesh->get_nen(eltid);
    QMatrix<int> id(id1, num_id_slots(), nen);
    for (int j = 0; j < nen; ++j) {
        int nodeid = mesh->ix(j,eltid);
        for (int i = 0; i < num_id_slots(); ++i)
            id(i,j) = mesh->inode(id_slots[i], nodeid);
    }
    int nbranch = mesh->nbranch_id(eltid);
    for (int i = 0; i < nbranch; ++i)
        id1[nen*num_id_slots()+i] = mesh->ibranch(i,eltid);
}


void ME::set_local_x(Mesh* mesh, int eltid, double* nodex1, int ndm)
{
    int nen = mesh->get_nen(eltid);
    QMatrix<double> nodex(nodex1, ndm,nen);
    for (int j = 0; j < nen; ++j) {
        int nodeid = mesh->ix(j,eltid);
        for (int i = 0; i < ndm; ++i)
            nodex(i,j) = mesh->v(i,nodeid);
    }
}


#define getter1(name, t, v) \
void ME::name(Mesh* mesh, int eltid, t* nodex1)    \
{                                                  \
    int nen = mesh->get_nen(eltid);                \
    QMatrix<t> nodex(nodex1, num_id_slots(), nen);     \
    for (int j = 0; j < nen; ++j) {                \
        int nodeid = mesh->ix(j,eltid);            \
        for (int i = 0; i < num_id_slots(); ++i)       \
```

```
            nodex(i,j) = mesh->v(id_slots[i],nodeid);   \
    }                                                   \
    int nbranch = mesh->nbranch_id(eltid);              \
    for (int i = 0; i < nbranch; ++i)                   \
        nodex1[nen*num_id_slots()+i] = mesh->v(mesh->ibranch(i,eltid));  \
}

getter1(set_local_u,  double,   u)
getter1(set_local_ui, double,   ui)
getter1(set_local_u,  dcomplex, uz)
getter1(set_local_v,  double,   v)
getter1(set_local_vi, double,   vi)
getter1(set_local_v,  dcomplex, vz)
getter1(set_local_a,  double,   a)
getter1(set_local_ai, double,   ai)
getter1(set_local_a,  dcomplex, az)
```

1.2.11 Assembling data from global arrays

The output of some of the assembly loops goes to external storage, but the `assemble_R` functions assemble their results into the mesh `F` array. The `add_local_f` functions take the local element residual contribution and add it into this array in the default way.

```
void ME::add_local_f(Mesh* mesh, int eltid, double* nodef1)
{
    int nen = mesh->get_nen(eltid);
    QMatrix<double> nodef(nodef1, num_id_slots(),nen);
    for (int j = 0; j < nen; ++j) {
        int nodeid = mesh->ix(j,eltid);
        for (int i = 0; i < num_id_slots(); ++i)
            mesh->f(id_slots[i],nodeid) += nodef(i,j);
    }
    int nbranch = mesh->nbranch_id(eltid);
    for (int i = 0; i < nbranch; ++i)
        mesh->f(mesh->ibranch(i,eltid)) += nodef1[nen*num_id_slots()+i];
}


void ME::add_local_f(Mesh* mesh, int eltid, dcomplex* nodef1)
{
    int nen = mesh->get_nen(eltid);
    QMatrix<dcomplex> nodef(nodef1, num_id_slots(),nen);
    for (int j = 0; j < nen; ++j) {
        int nodeid = mesh->ix(j,eltid);
        for (int i = 0; i < num_id_slots(); ++i) {
            mesh->f (id_slots[i],nodeid) += real(nodef(i,j));
            mesh->fi(id_slots[i],nodeid) += imag(nodef(i,j));
        }
```

```
    }
    int nbranch = mesh->nbranch_id(eltid);
    for (int i = 0; i < nbranch; ++i) {
        mesh->f (mesh->ibranch(i,eltid)) += real(nodef1[nen*num_id_slots()+i]);
        mesh->fi(mesh->ibranch(i,eltid)) += imag(nodef1[nen*num_id_slots()+i]);
    }
}
```

1.3 Evaluating fields

There are a variety of functions that we might want to evaluate inside elements: stress, heat flux, power flux, potential gradient, etc. We might also want some interesting combination of these variables. The `FieldEval` interface provides a uniform way of calling these functions, so we can (for example) write a generic loop to evaluate a function $f(X)$ at all the Gauss points in the mesh and then compute an L^2 projection.

At present, the only implementation of this interface is `FieldEvalPower`, which makes a call back to the element `mean_power` method.

```
class FieldEval {
 public:
    virtual ~FieldEval();
    virtual void operator()(Mesh* mesh, int eltid, double* X, double* fX) = 0;
};

class FieldEvalPower : public FieldEval {
 public:
    ~FieldEvalPower();
    void operator()(Mesh* mesh, int eltid, double* X, double* fX);
};
```