

CONCEPTS OF CAD-INTEGRATED FINITE ELEMENT ANALYSIS

Stefan M. Holzer

Informationsverarbeitung im Konstruktiven Ingenieurbau
Universität Stuttgart, Germany
e-mail: stefan.holzer@po.uni-stuttgart.de

Key words: geometric modeling, mesh generation, efficiency, hp-version

Abstract. *A CAD system contains only part of the information required for structural analysis, namely, geometrical data. In order to generate a finite element model, the CAD geometry needs to be simplified, e.g. by dimensional reduction. Furthermore, information on boundary conditions and loading needs to be supplied. These tasks cannot be performed solely on the basis of a geometrical model, but require engineering expertise. Therefore, direct automatic transition from a CAD model to a finite element model is not feasible.*

However, finite element analysis can be greatly aided by embedding in a CAD-like environment. Such an integrated FEM software needs to provide three distinct kinds of models: (a) a structural model, composed of the simplified geometry and boundary and loading conditions, as well as load case information; (b) a series of specifically tailored finite element meshes for each of the load cases; (c) a result evaluation model. Furthermore, the finite element analysis itself should provide means of convergence control and adaptivity. We discuss suitable data structures for the three kinds of models, strategies for the construction of these models, efficient algorithms, and practical applications.

1 Introduction

CAD is a tool for creating *geometrical models* of a real-world structure. A CAD system contains no information whatsoever which is directly useful for structural analysis. Therefore, the concept of "CAD-FEM integration" is misleading.

To arrive at a model suitable for structural analysis, a series of abstraction steps is necessary, each requiring additional engineering judgement and input from the user's side.

The first of these steps is breaking a complex structure into manageable members or sub-structures whose behavior can be analyzed individually, connected to the others only by load and support conditions. Even today, very little computer based assistance is available in this step, and little more can be expected, since the effect of this idealization can only be guided by experience.

The next step is geometric idealization, in particular, dimensional reduction. There is not one unique geometrical model for each structural member, but a hierarchic series of geometrical abstractions, ranging from one-dimensional to fully three-dimensional models. As an example, we mention idealizations of plates ranging from the "very thin" Kirchhoff plate to the "very thick" fully three-dimensional model, including intermediate abstraction levels such as the "moderately thick" Reissner-Mindlin plate. Geometric abstraction of structures with one particular dimension (e.g., plates or slabs with a distinct thickness dimension) can be aided by automation based on the concept of hierarchical modeling. It ought to be remarked that many geometrical idealizations which are already available in today's finite element codes, such as the Timoshenko beam, or the Reissner-Mindlin plate, provide excellent approximations of higher-dimensional geometrical idealizations, even without a full-featured hierarchical modelling capability. The Reissner-Mindlin plate theory can be viewed as a first link in a transition from purely two-dimensional to fully three-dimensional models.

Similarly, the material and loads need to be idealized, once again opening a range of possibilities, e.g., from linear elasticity up to plasticity or visco-plasticity.

Having solved all these idealization steps, one arrives at a mathematical model of the structure which is now amenable to numerical analysis. We do not have a finite element mesh, though. However, we do have a set of rules for mesh layout and numerical analysis, depending on geometry, boundary conditions, and loading. These rules can be applied to derive a finite element mesh from a "master model" describing the mathematical model.

In general, a structure is subject to various different load cases, each of which requires an individual finite element mesh layout. Therefore, one CAD geometry may correspond to a set of "master models" (structural models), and each structural model in turn may correspond to a set of completely different finite element models for each of the load cases.

However, in the final step of design, feed-back of the results of the numerical analysis to the geometrical design is required. This feed-back is unrelated to the individual finite element meshes, therefore, an additional framework is required for the result evaluation of the

numerical analysis. We call this unifying framework the "result model". The numerical results can be viewed as attributes to the topological entities of the result model, such as nodal attributes, or edge attributes.

2. Data structure for the structural model

In this contribution, we focus on two-dimensional models or hierarchic extensions of two-dimensional models. Figure 1 shows the graphical user interface of a finite element analysis code for Reissner-Mindlin plate problems.

Figure 1: Typical problem geometry in a plate bending problem from civil engineering

A typical problem to be solved by such a program is the analysis of a system of inter-connected plates, including multiply connected subdomains. Such a system of inter-connected plates in fact resembles the surface of a three-dimensional body in a representation that is mapped to the plane. There are no edges that are connected to more than two adjacent faces, and topological features like "dangling" edges or faces that meet only in a single vertex are illegal from the point of view of mathematical theory. For example, it is not legal to attach a one-dimensional element like a beam to a two-dimensional continuum like a Reissner-Mindlin plate in only one single point. We call such two-dimensional systems of inter-connected polygons "meshlike" plane structures in the following.

Three-dimensional bodies whose surfaces fulfil restrictions similar to those listed above are so-called *two-manifolds*. Data structures for *boundary representation (B-rep) models* of *two-manifolds* include the *winged edge* and *half winged edge* data structures (see [1]).

Therefore, it seems natural to *adopt B-rep* data structures for three-dimensional, two-manifold *bodies* in order to model two-dimensional "meshlike" plane structures. In some way, the *B-rep* representation of the three-dimensional manifold can be "abused" as a representation of a two-dimensional "meshlike" structure. Similarly, data structures for three-dimensional *non-manifold* structures can be utilized for topological modeling of three-dimensional "meshlike" structures.

In our case, the structural model is based on an implementation of the winged-edge data structure. Topologically, the model differs from a *B-rep* model of a three-dimensional body only in that the "surface" of the "body" is not closed if the surface describes a "meshlike" plane structure. However, we remedy this fault by adding an imaginary "outer face" to the model, which formally achieves the closure of the imaginary two-manifold.

The winged edge data structure is based on the topological entities *vertex*, *edge*, *face*, *shell*, and *region*. The winged edge data structure uses only fixed-size topological relations. However, the original winged-edge data structure has been extended by the *loop* entity in order to permit enclosures and openings inside the faces. A *loop* is a closed ring of edges. Each *face* entity contains a list of *loops*, the first of which is implicitly assumed to denote the

outer boundary of the face, whereas the remaining loops are meant to describe inner openings of multiply connected domains. The loop list is the only dynamically-sized component of the extended winged-edge data structure.

An object-oriented implementation of such an extended winged-edge data structure reads as follows:

```
class CVertex : public CWEObject
{
protected:
    double x, y, z;
    CEdge* m_pE;
public:
    CVertex(CModel* pM, double x, double y, double z=0);
    void AddEdge(CEdge* pE);
    BOOL RemoveEdge(CEdge* pE);
    double GetX(){return x;}
    double GetY(){return y;}
    double GetZ(){return z;}
// retrieval functions:
    int CountE();
    CEdge* GetE(Iterator& pos);
    void GetFaces(CList<CFace>* pList);
    void GetEdges(CList<CEdge>* pList);
// manipulation functions:
    ...
};
```

The *vertex* class contains only one reference to an edge. The topological relation *vertex-edge* can be computed exploiting the links of the *edge* entity. The winged edge data structure hinges essentially on the *edge* entities:

```
class CEdge : public CWEObject
{
protected:
    // vertex pointers
    CVertex* v1;
    CVertex* v2;
    // clockwise edge pointers:
    CEdge* epcw;
    CEdge* encw;
    // counterclockwise edge pointers:
    CEdge* epccw;
    CEdge* enccw;
    // loop pointers
    CLoop* lcw;
    CLoop* lccw;
    // geometric shape:
    ...
public:
```

```

    CEdge(CModel* pM, CVertex* p1, CVertex* p2, CLoop* pL);
    virtual ~CEdge();
// retrieval functions:
    CVertex* GetV1(const CLoop* l) const;
    CVertex* GetV2(const CLoop* l) const;
    CVertex* GetV1() const {return v1;}
    CVertex* GetV2() const {return v2;}
    CVertex* GetOtherV(CVertex* pV);
    CLoop* GetLccw(){return lccw;}
    CFace* GetLcw(){return lcw;}
    ...
};

```

The edge entity contains eight references to other topological entities, including a subset of the *edge-edge* adjacency relation, the *edge-vertex* relation (start and end vertex), and two references to adjacent *loops*, or *faces*. In the case of an edge that pertains to the outer boundary of a two-dimensional array of faces, one of the *loop* pointers points to a *loop* that belongs to an imaginary "outside" face. This "outer" *loop* permits to follow all edges of the outer boundary of the structure in clock-wise direction. Each hole is also characterized by a *loop* that circles the hole in clockwise direction. The *loop* data structure is defined as follows:

```

class CLoop : public CWEObject
{
protected:
    CEdge* m_pE;    // starting edge of loop
    CLoop* m_pLn;  // next loop in intrusive list
    CLoop* m_pLp;  // previous loop in intrusive list
    CFace* m_pF;   // points to face
public:
    CLoop(CModel* pM, CEdge* pE, CFace* pF, CLoop* prev=0,
          CLoop* next=0);
    ~CLoop();
// retrieval functions:
    CEdge* GetE(){return m_pE;}
    CFace* GetF(){return m_pF;}
    CVertex* GetV(Iterator& pos) const;
    CEdge* GetE(Iterator& pos) const;
    void GetEdges(CList<CEdge>* pList);
    int CountE();
    ...
};

```

The *loop* is an artificial topological entity, the main purpose of which is to permit the description of multiply connected faces. Each edge belongs to exactly two loops, in exactly the same fashion as one edge being used by two faces in the original winged edge data structure.

The *face-loop* relation is embodied by an intrusive linked list embedded in the *loop* data structure. Therefore, the *face* data structure reduces to:

```
class CFace : public CWEObject
{
protected:
    CLoop* m_pL;
public:
    CFace(CModel* pM);
    virtual ~CFace();
// retrieval functions:
    CVertex* GetV(Iterator& pos) const;
    CLoop* GetL(Iterator& pos) const;
    int CountV();
    ...
};
```

In our implementation, all winged edge entities belong to classes derived from a base class `CWEObject` ("winged edge object"). This base class provides persistency functions as well as a hybrid *iterator* class used in all retrieval functions.

The elements of the structural model belong to classes derived from `CVertex`, `CEdge`, `CLoop`, and `CFace`. Inheritance is used to supply the structural components with suitable *attributes*.

The *attributes* to be included in the structural model include support conditions and edge loads (associated with the *edge* entities), material, dead load, and cross-section data (associated with *face* entities), and geometrical data (associated with *vertex* and *edge* entities, respectively). In addition, support and load conditions are associated with *time*. In our case, we consider only time-independent problems, so that the *time* association can be replaced by a *load case* class.

In addition to the *retrieval functions* shown in the code listings above, we have implemented a set of EULER operators for building up the model and modifying an existing model. Higher-level manipulation functions such as splitting of edges, splitting of faces, and so on, have also been provided. In combination with a graphical user interface that offers many CAD-like functions, it is very easy and fast to build up a consistent topological model of the two-dimensional idealization of the structure. Operations like insertion of vertices into existing edges will be carried out automatically when the user constructs a new polygon next to an existing one, so that inconsistent "unconnected", but geometrically adjacent edges cannot be input.

3. Data structure for the finite element model

We have found that the structural model of a two-dimensional structure resembles the surface of a two-manifold body that has been mapped to the plane. This is even more striking for a plane finite element mesh. A plane finite element mesh could be modelled by the original winged edge data structure without modification. In our case, the extended winged edge data structure being available, we have utilized this data structure for the finite element mesh as well.

In other words, our implementation differs drastically from the classical "element list" and "node list" data structure generally used for finite element meshes. Theoretically, we could accommodate finite elements with arbitrarily many vertices, with holes, and with arbitrarily curved edges. While our approach might appear like over-kill on first view, it is important to remark that these features of our implementation help greatly in the following fields:

Figure 2: Typical hierarchical mesh refinement.

Mesh adaption. Mesh refinement invariably produces "elements with more than four vertices" in a first step. Consider, for example, a uniform refinement of a finite element mesh as shown in figure 2. In the first step, edges of elements that have been marked for refinement (in our case, all edges) will be bisected. In the next step, the resulting "eight-noded" elements will be split each into four new, smaller quadrilaterals. While the end product consists of quadrilaterals exclusively, the intermediate model cannot be represented by the classical FE mesh data structure.

p-version analysis and combined mesh refinement and p-adaption. Our program is laid out to permit an extension of the polynomial degrees (cf. [2]), as well as mesh refinement, in order to ensure fast convergence of local and global quantities. To this end, hierarchical higher-order trial functions have been implemented, which can be grouped into three classes as displayed in figure 3. *Vertex modes* are the classical vertex-based bi-linear trial functions. In addition, we use *edge modes*, which are identically zero on the boundary of the element, save one edge. Finally, *interior* or *bubble* modes are non-zero only inside the element, but not on its boundary. In order to ensure the C^0 continuity of the finite element approximation, the coefficients associated with these trial functions can be handled quite naturally as *attributes* to the pertaining topological entities *vertex*, *edge*, and *face*.

Error estimation. The assembly procedure associated with *p*-type trial functions, as well as residual-based error estimation, rely heavily on evaluations of topological adjacency relations. These are much easier computable for a finite element data structure based on the winged edge data structure than they are for the classical kind of finite element data structure.

In our implementation, each topological entity of the finite element mesh keeps a reference to an associated "master" entity in the structural model, if any exists. Therefore, it is straightforward to take care of proper adjustment of the geometrical shape of the finite element mesh in case of mesh refinement, or to incorporate high-level shape descriptions for finite elements, using the blending function method in the *p*-version of the finite element analysis. It is advisable not to keep actual *pointers* to the master entities, but rather "smart" pointers in order to be able to adapt the relations between a finite element mesh and its master model at any time, avoiding "dangling" pointers.

In our approach, we keep an arbitrary number of finite element meshes. At each time-step or in each load-case, an individual mesh may be used.

4. Mesh generation

We have discussed the data structure for the finite element model. This data structure will be employed to work with an existing finite element mesh. However, it is by no means required to use this data structure during the initial *mesh generation process*. In contrast, the automatic mesh generator that has been applied in our code uses the advancing front strategy, and therefore it is straightforward to supply the mesh generator with a simple linked-list-based data structure. Only after completion of the mesh generation process, the winged-edge representation of the entire mesh will be built up. This is reasonable not only for speed considerations, but also because intermediate stages of the mesh generation process might not be represented by the winged edge data structure. For example, there is no way to guarantee that the mesh generator would not generate – in some stage of the mesh generation process – two patches of elements that have only one point in common. However, such a situation cannot be represented by the winged edge data structure.

We note that, in contrast to advancing front algorithms, other mesh-generation techniques such as recursive bi-section lend themselves very well for yet another application of the winged-edge data structure, since, by construction of the meshing algorithm, situations that are outside the scope of the winged-edge data structure cannot arise when such an algorithm is employed, whereas bisection operations can be carried out extremely efficiently on the base of a winged edge representation.

With a view to the data structure chosen for the *result model* (see below), we note that advancing front mesh generation tends to "fill" the area with vertices and elements in a fairly well-distributed way.

Finally, a note on *coarse mesh generation for p- and hp-type finite element approximation* might seem appropriate here. Common mesh generators tend to fill an arbitrary complex structure with a large number of small elements. However, in our approach, we intend to start from a finite element mesh that is initially as coarse as possible.

Very little literature is available on *coarse mesh generation*. In our approach, we start generally from a background *mesh density function*, as has been suggested in various contributions for *adaptive remeshing* strategies. However, the question remains how to obtain a suitably coarse background mesh density function for the *initial step* of the computation. In order to obtain a background density function, we proceed as follows:

Firstly, we create a uniform, dense mesh of triangles, using any mesh generator available. We use this mesh for solving an *auxiliary problem*, which is defined as follows: The problem is governed by a LAPLACE equation with inhomogeneous DIRICHLET boundary conditions. These boundary conditions are given by the desired finite element edge lengths on the boundary. C^0 continuity of the boundary conditions is ensured by nodal averaging. After solving this problem by a simple finite element approximation, using linear ansatz functions on the dense initial mesh, we obtain the required edge lengths at all vertices of the dense background mesh. In the second step, we make use of this density distribution in order to generate the "coarsest possible" finite element mesh as a starting point of our adaptive algorithms. Details of the algorithm have been described in [3].

While this approach might seem excessively costly (we create a coarse mesh at the expense of first creating a dense mesh), it does not really mean any overhead on our total program, because the dense mesh of triangles will serve additional purposes, besides providing a base for computing the density function. In fact, the background mesh is at the same time the base of our *result model*. Furthermore, there is only one unique background mesh, which will be used as a basis for generating arbitrarily many finite element mesh for arbitrarily many load cases.

5. Data structure for the result model

As noted already above in section 4, we keep an independent, stand-alone *result model* in addition to the structural and finite element models. Because our software system is based on individual finite element meshes for each load case, a unique reference frame is required for loadcase superposition and other post-processing operations. Also, for time-dependent problems, a reference frame for mesh transition mapping is required.

To this end, we keep the result model. The result model is also the base of graphical representations of the results. As opposed to finite element models, the result model will be created once and never touched again afterwards. The result model ought to consist of geometrical simplices that lend themselves readily for purposes like graphical representation, interpolation, and local query.

Therefore, we have decided to use a (uniform) mesh of triangles for the result model. It has already been pointed out in section 4 that this mesh is also used as a mesh for determining and storing the mesh density function during coarse (or non-uniform) mesh generation.

The data structure chosen for the result model resembles classical finite element data structures closely. There is a list of triangles, each triangle referring to its three vertices. Furthermore, there is a collection of vertices, and all data associated with the result model are treated as attributes to the vertices of the mesh. Edges do not show up in the result mesh data structure. There are no attributes associated with the triangles themselves.

In order to be able to perform geometric queries very efficiently, the vertices of the result mesh are kept in a point quadtree (see [4]) rather than in a list. A point quadtree is a data structure that describes a recursive quadrisection of the plane where, in our case, the common point of the four quadrants is identical to a vertex of the result mesh. The *data* associated with each node of the point quadtree read as simply as follows:

```
class QuadTree
{
private:
    QuadRoot* m_pRoot;
    QuadTree* Son[4];
    CRVertex* m_pInfo; // pointer to result model vertex
public:
    ...
};
```

For convenience, we include a pointer back to the root in all nodes of the tree. It is much easier to build a quadtree than to modify it. However, modification is not required in the setting discussed here. It is worth noting that most mesh generators which might be utilized for generating the result mesh produce the vertices of the mesh in a sequence which is quite well suited for inserting into a quadtree; generally, no further actions for balancing the tree are required, because the vertices created are evenly distributed about all parts of the structure, particularly when using an advancing front algorithm. The following table (table 1) shows the height of the quadtrees resulting from the structure shown in figure 1, using uniform result meshes:

Number of vertices	height of point quadtree	minimum tree height
454	15	2
1712	23	5
6750	49	7
27301	109	8
107678	143	9

Table 1: Height of point quadtree resulting from straightforward point insertion based on reverse ordering from advancing front algorithm.

These experimental data have been obtained by inserting the vertices into the point quadtree in the *reverse order* of their creation. Naturally, this simple insertion strategy does not produce a balanced or nearly balanced tree, as indicated by the "minimum tree height" in the table; however, the table indicates that the height of the tree is growing slower than proportional to the square root of the number of vertices inserted. Therefore, the point quadtree is by far more efficient than a linear linked list representation. By the way, insertion of the points in the opposite sequence results in approximately twice the height. Figure 3 shows the result mesh at $N=6750$.

Figure 3: Typical result model.

Based on the point quadtree data structure, the local mesh density can be determined very quickly during the mesh generation process. In order to find out the desired mesh density at a point $(x;y)$, we simply select all vertices of the background mesh which are inside a square, the size of which corresponds to two times the average element size of the mesh triangles. The local density is then computed as the average of the densities stored at all vertices inside the searching square.

Similarly, the finite element post-processing works as follows: Starting from a bounding rectangle, all vertices of the result model that are "close to" a finite element of the finite element model are selected, exploiting the point quadtree. For each of these vertices, the inverse mapping to the standard reference frame for the finite element is performed by a Newton-Raphson scheme (in general, the mapping is non-linear due to arbitrarily curved edges), excluding vertices which are actually inside the bounding box, but outside the element. The computation of stresses is then performed for all the points inside the element under consideration. This is done in a loop over all elements.

Results are loosely associated with the vertices by a map included in each load case or time step. There are no direct lists of result attributes associated with the result mesh vertices, since such lists would make access to results by load case and insertion or replacement of results a tedious and slow process.

6. Automatic analysis

In the model implementation discussed here, both the uniform h -extension and the uniform p -extension of the finite element method have been implemented. Both are based on a hierarchical refinement concept, such that, during the extension of the ansatz spaces, the lower-order ansatz spaces are completely embedded in the higher-order ansatz spaces. Therefore, it is possible to monitor *global convergence* in a suitable norm (here, the energy norm) by RICHARDSON extrapolation on the base of three subsequent finite element approximations (cf. [2]). Furthermore, the user may define monitoring points in which the convergence of local quantities like stresses may be recorded.

In the p -version approach, we use meshes that have been tailored specifically to meet the needs of the problems, i.e., we introduce point refinements *a priori* in all potentially singular points, as well as anisotropic edge refinements close to boundaries where we expect edge effects (boundary layers; there is a boundary layer in the shear forces in Reissner-Mindlin theory). Figure 4 shows a typical hp -version mesh used for this purpose.

Figure 4: Typical hp -version mesh.

Figure 5: Typical global convergence. Uniform p extension on a hp -type mesh and uniform h extension.

Figure 6: Typical result of monitoring *local* results. Convergence

References

- [1] B. G. Baumgart: *Geometric modeling for computer vision*. Research report, Stanford University, (1974).
- [2] B. A. Szabó and I. Babuska, *Finite Element Analysis*, John Wiley & Sons, New York, (1991).

- [3] S. M. Holzer: *Mesh generation for hp type finite element analysis of Reissner-Mindlin Plates*, preprint no. 99-02, Informationsverarbeitung im Konstruktiven Ingenieurbau, Universität Stuttgart, (1999).
- [4] H. Samet: *The design and analysis of spatial data structures*. Addison-Wesley, Reading, (1989).