

Object oriented design of a finite element code in Java *

G.P.Nikishkov[†]

Abstract: This paper presents the object oriented approach to programming the finite element method using the Java language. The developed finite element code consists of ten Java packages. Three main methods are related to generation of finite element models, solution of elastic and elastic-plastic boundary value problems, and visualization of models and solution results. Object-oriented model of the code is described. It is demonstrated that Java 1.5 new features are useful in development of the finite element code. Java 3D is used for visualization of models and results.

Keywords: Object oriented approach, Java, Java 3D, finite element method, elastic, elastic-plastic, visualization.

1 Introduction

The finite element method is used for computational modeling in its original form [Bathe (1996)] for several decades. Recently the method has been developed further in meshless form [Atluri and Shen (2002); Atluri(2004)].

Finite element codes were traditionally developed in Fortran and C languages, which support procedural programming. During last fifteen years, finite element development is gradually shifting towards an object oriented approach. Forte et al (1990) in one of the first publications on the object oriented approach to the finite element development, presented essential finite element classes such as elements, nodes, displacement and force boundary conditions, vectors and matrices. Several authors described detailed finite element architecture using the object oriented approach. Zimmermann et al. (1992) and Commend et al. (2001) proposed basics of object oriented class structures for elastic and

elastic-plastic structural problems. A flexible object oriented approach which isolates numerical modules from a structural model is presented by Archer et al. (1999). Macki devoted numerous papers and a book [Mackie (2001)] to various aspects of the finite element object oriented programming including creation of interactive codes with graphical user interface. Extensive bibliographical information on the object oriented approach in FEM and BEM is given by Mackerle (2004).

Mostly, object oriented finite element codes have been implemented in C++ programming language. It was shown that object oriented approach with C++ programming language could be used without sacrificing computational efficiency [Dubois-Pelerin and Zimmermann (1993)] in comparison to Fortran. A paper of Akin et al. (2002) advocates employing Fortran 90 for object oriented development of finite element codes since the authors consider Fortran execution faster than C++.

Java language introduced by Sun Microsystems possesses features, which make it attractive for using in computational modeling. Java is a simple language (simpler than C++). It has rich collection of libraries implementing various APIs (Application Programming Interfaces). With Java it is easy to create Graphical User Interfaces and to communicate with other computers over a network. Java has built-in garbage collector preventing memory leaks. Another advantage of Java is its portability. Java Virtual Machines (JVM) are developed for all major computer systems. JVM is embedded in most popular Web browsers. Java applets can be downloaded through the internet and executed within Web browser. Useful for object oriented design Java features are packages for organizing classes and prohibition of class multiple inheritance. This allows cleaner object-oriented design in comparison to C++. Despite its attractive features, Java is rarely used in finite element analysis. Just few publications can be found on object oriented Java finite element codes [Eyheramendy and Guibert (2004)]. Previously, Java had a reputation

* *Computer Modeling in Engng and Sciences*, 2006, 11, 81-90.

[†] University of Aizu, Aizu-Wakamatsu 965-8580, Japan

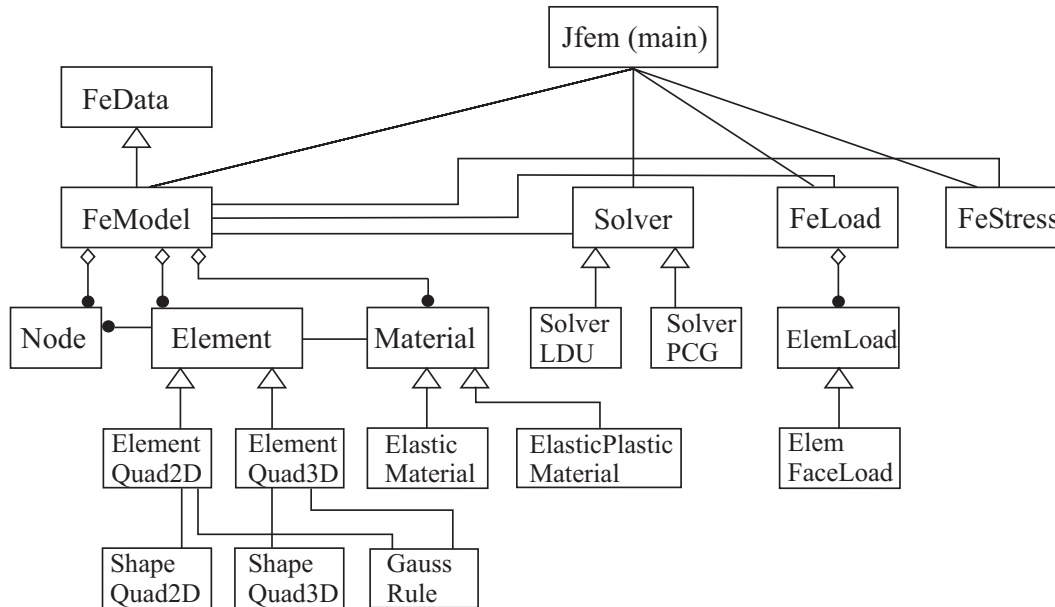


Figure 2: Class diagram of the finite element processor.

Main class *Jfem* contains main method of the finite element processor. Typical finite element solution flow is composed of data input, assembly of the global equation system, solution of the equation system, computing stresses and results output. Constructor *Jfem* creates four objects responsible for performing computational procedure:

FeModel - finite element model, data input;

Solver - assembly of the global stiffness matrix from element contributions, solution of the global equation system;

FeLoad - load case, assembly of the global load vector;

FeStress - stress increment, results output.

3.2 Finite element model

Main purposes of class *FeModel* are to read information on the finite element model, to store this information and to provide it to other classes. Class *FeModel* extends class *FeData* (Fig. 2). Class *FeData* contains scalars, arrays and objects used for description of the finite element model:

Scalars: number of elements, number of nodes, number of degrees of freedom per node etc.;

Arrays: node coordinates, displacement boundary conditions;

Objects: elements, materials.

Object-oriented approach allows to create reusable, extensible, and reliable code components. However, the extensive use of the object-oriented paradigm everywhere might not be always ideal for computational efficiency of the code. Object creation and destruction are expensive operations. The use of large amount of small objects can lead to considerable time and space overhead. Thus, we tried to find a compromise between using objects and providing computational efficiency. A possible way to increase computing performance is using primitive types in place of objects. That is why information on the finite element model is mostly stored as scalar and arrays.

New features of Java 1.5 include typesafe *Enums* and new class *Scanner* useful for data input. *Enums* is a flexible object-oriented enumerated type facility, which allows one to create enumerated types with arbitrary methods and fields. The *java.util.Scanner* class can be used to convert text into primitives or Strings. It also offers a way to conduct regular expression based searches on streams, file data and strings. A simple example below shows unformatted unordered input of two scalar variables *nEl* and *nNod* using Java 1.5:

```

enum vars {
    nel, nnod
}
int nel, nnod;
Scanner sc = new Scanner(new File(s));
sc.useDelimiter("\\s*=\\s*|\\s+");
while (sc.hasNext()) {
    String name = vars.valueOf(sc.next());
    switch (name) {
        case nel: nel = sc.nextInt();
                break;
        case nnod: nnod = sc.nextInt();
    }
}

```

Input file may contain expressions of the type

```
nnod = 200 nel = 100
```

in free format and in any order. Names and values of the variables are separated by equal sign and any number of blanks.

In the above code fragment, names of the variables are declared in `enum vars`. Scanner object `sc` is created for the file with the name `String s`. Method `sc.useDelimiter` sets equal sign and blank as delimiters for the input text. Method `sc.hasNext` allows to check if this scanner has another token in its input. Methods `sc.next` and `sc.nextInt` read string and integer from the input.

3.3 Element

Class *Element* represents a base abstract class for all finite element types. Description of *Element* is given in Table 1.

Each element object contains element name, material number, array of nodal connectivities and arrays of accumulated stresses and stress increment. Constructor *Element* creates element object using its name, number of nodes and number of integration points where stresses are stored. In the code, element objects are created using constructor *NewElement* with element name. *NewElement* class contains all element types in *enum* structure. The Java 1.5 *enum* declaration defines a full-fledged class (dubbed an *enum* type). It allows one to add arbitrary methods and fields to an *enum* type and to implement arbitrary interfaces. Constructor *NewElement* creates element objects of different types employing just element names.

Element
+ name: String + matNo: int - connectivities: int[] - sStress: double[][] - dStress: double[][] - stiffMat: static double[][] - elemVec: static double[] - elemCoords: static double[][] - elemTemps: static double[]
+ Element(String name, int nCon, int nGauss) + stiffnessMatrix(): double[][] + thermalVector(): double[] + equivFaceLoad(ElemFaceLoad surLd): int + getElemFaces(): int[][] + getStrainsAtIntPoint(int intPoint): double[] + getTemperatureAtIntPoint(int intPoint): double + extrapolateToNodes(double[] vAtIntPoints): double[] + setElemConnectivities(int[] indel) + setElemMaterial(int mat) + setElemXy() + setElemXyT() + getElemMaterial(): int + getElemXy(): double[][] + setElemT(): double[] + assembleElemVector(double[] glVector) + disAssembleElemVector(double[] glVector) + getElemConnectivities(): int[]

Table 1: Description of class *Element*.

Classes *ElementQuad2D* and *ElementQuad3D* inherit some methods from class *Element* and implement actual methods for the two-dimensional quadrilateral element with 8 nodes and for the three-dimensional hexahedral element with 20 nodes. Both elements have quadratic interpolation of geometry and field variables. Classes *ShapeQuad2D* and *ShapeQuad3D* contain methods for quadratic shape functions. Methods of a particular element class compute stiffness matrix, thermal vector, equivalent nodal load, strains at integration point and perform other necessary element operations (see Table 1).

3.4 Material

Class *Material* is a base abstract class for material constitutive equations. Table 2 provides methods of *Material* class.

Classes *ElasticMaterial* and *ElasticPlasticMaterial* extend class *Material*. Method *getConstitutiveMatrix* provides elasticity matrix or elastic-plastic constitutive matrix. Method *strainToStress* computes stress incre-

Material
+ stressState: String
- elastModulus: double
- poissonRatio: double
- thermExpansion: double
- properties: double[]
+ Material(String stressState)
+ getConstitutiveMatrix(): double[][]
+ strainToStress(double[] ep, double t, double[] sig)
+ setElastModulus(double e)
+ setPoissonRatio(double nu)
+ setThermExpansion(double alpha)
+ setProperties(double[]p)
+ getElastModulus(): double
+ getPoissonRatio(): double
+ getThermExpansion(): double
+ getProperties(): double[]

Table 2: Description of class *Material*.

ment using given strain increment.

3.5 Equation solver

Base abstract class *Solver* includes methods for assembly and solution of the global equation system. Its major methods are:

assembleGlobStifMat() - assembly of the global stiffness matrix;

assembleElemStifMat() - assembly of element contribution to the global stiffness matrix;

solve(double x[]) - solution of the global equation system with the right-hand side *x*.

Class *Solver* does not contain any information about storage schemes for the global stiffness matrix and about solution methods. Particular storage and solution methods are implemented in subclasses, which extend *Solver*. Subclass *SolverLDU* implements LDU decomposition method for the solution of the global equation system stored in the symmetric profile format. Subclass *SolverPCG* uses preconditioned conjugate gradient (PCG) method for the solution of equation system with a storage in sparse row format.

Currently the Java compiler does not have enough means for powerful code optimization. Because of this an attention should be devoted to a code tuning. It is necessary to identify code segments, which consume major computing time and to tune them manually. Tuning of the equation systems solution procedure is discussed

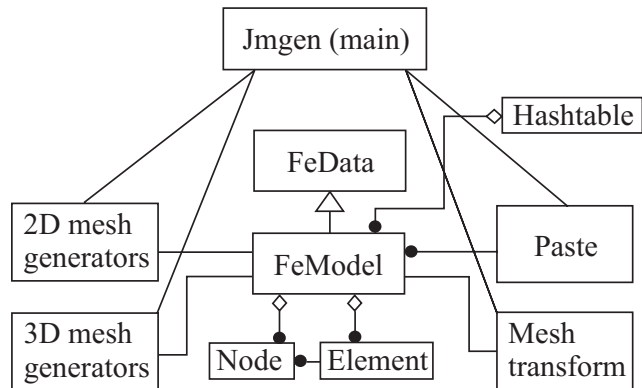


Figure 3: Class diagram of the finite element preprocessor.

by Nikishkov et al (20030. During LDU decomposition of the global stiffness matrix, triple loop in which one matrix column is used to modify another column, takes most computing time. Tuning can be done by unrolling two outer loops. In the tuned LDU decomposition a block of matrix columns modifies another block of matrix columns thus economizing data loads from memory. Tuning of the PCG solution is performed by unrolling one inner loop in sparse matrix-vector product.

While tuning requires some additional efforts, its use may considerably enhance performance of the Java code making it comparable to performance of the analogous C code.

4 Mesh generation

4.1 Block-decomposition method and code structure

Mesh generation in the Jfea code is based on the block-decomposition method [Schneiders (2000)]. In the block-decomposition method, the user divides a solution domain into multiple blocks in such a way that each block is suitable for the local meshing process. Mesh generation within blocks performed by various mesh generators. Some blocks can be meshed using both two-dimensional and three-dimensional approaches. First, two-dimensional mesh is created. Then this mesh is swept in space to produce three-dimensional mesh block. A pair of generated mesh blocks can be pasted together in order to create new mesh block.

Class diagram of the finite element preprocessor is

shown in Fig. 3. Class *Jmgen* contains main method and activates all other classes necessary for mesh creation. Each class performs some action on one or more finite element models. *FeModel* objects are stored in a hashtable *blocks*. For example, any mesh generator can create a mesh block as *FeModel* and can put it in *blocks* hashtable under the name specified by the user.

Since all modules are independent then main class calls them by name of its class as shown in the following code fragment.

```
Scanner sc = new Scanner(new File(s));
while (sc.hasNext()) {
    String c = sc.next().toLowerCase();
    if (name.equals("#")) {
        sc.nextLine();
        continue;
    }
    Class w = Class.forName("gener."+c);
    w.newInstance();
}
```

Here the scanner *sc* reads data from an input file. If a read token appears to be *"#"* then this line is considered a comment and the scanner reads token from the next line. If string *c* is not a comment character it is supposed to be a class name. New object *c* from package **gener** is created using methods *forName* and *newInstance*.

4.2 Mesh generators

Various mesh generators can be included in the code without any difficulty. A mesh generator creates *FeModel* object. Generated nodes and elements are placed in the *FeModel* object using its appropriate methods. Then the model object is stored in hashtable *blocks*.

Usually one mesh block has simple shape and topology. Because of this relatively simple mesh generators based on mapping from local to global coordinates can be used. Two such mesh generators are implemented in the postprocessor code. Class *genquad2d* contains a mesh generator for a two-dimensional quadratic quadrilateral area. The area is specified by eight nodal points. Regular mesh is generated in local coordinates. Then nodal coordinates are transformed into the global cartesian system using quadratic shape functions. Similar three-dimensional class *genquad3d* generates a mesh inside a curves hexahedral area, which in general is determined by twenty nodal points.

Another useful method for three-dimensional mesh generation is sweeping a two-dimensional mesh in space. Mesh generator *sweep* extracts a two-dimensional mesh with a given name from hashtable and moves it along specified three-dimensional trajectory. The two-dimensional mesh is copied at predetermined positions. These two-dimensional sections serve as a skeleton for creation of a three-dimensional mesh. Application of transformations to two-dimensional sections allows to produce complicated three-dimensional meshes.

4.3 Connecting blocks and other operations

Mesh blocks are connected together by pasting surfaces with coincident nodes. Class *paste* provides a method for producing new mesh from given two mesh fragments. Surface nodes of two mesh fragments are compared to each other. If a distance between a pair of nodes is less than a specified tolerance then two nodes are merged. This means the node is deleted from the second mesh and its number is registered in a list for subsequent modification of the connectivity information.

Other operations on mesh blocks include:

- transform* - translate, scale and rotate node locations of a mesh block;
- copy* - create a copy of a mesh block under new name;
- delete* - remove a mesh block from the hashtable;
- readmesh* - read a mesh block from a file;
- writemesh* - write a mesh block to a file.

5 Visualization

5.1 Using Java 3D API

In general, the development of visualization software is a complicated task. One possibility to make this task simpler is to employ well-established visualization tools like the Visualization Toolkit VTK [Schroeder et al (1998)]. Such tools have a lot of opportunities for visualization. Nevertheless, particular features necessary for finite element visualization, especially in the case of higher-order elements, may be missing.

Fortunately, Java provides an object-oriented graphics library for three-dimensional visualization, which is called the Java 3D [Sowizral et al (2000)]. The developer specifies geometry of visual objects, their appearance and behavior and light sources as Java 3D objects.

Java 3D objects are placed into a scene graph data structure. The scene graph is a tree structure that specifies the content of a virtual universe, and how it is to be rendered. After compiling, the scene is rendered automatically with "quasi"-photographic quality. The latter means that the effects of light source shading are shown, however, object shades and reflections are ignored.

The Java 3D employs linear triangular and quadrilateral filled polygons for surface representation. Visualization of finite element models consisting of simplest elements is almost straightforward. However, for higher order elements the transformation of element surfaces into triangular polygons should be done carefully taking into account both geometry features and result field gradients.

5.2 Visualization algorithm

The input data for the visualization is a finite element model produced by the preprocessor *Jmgen* (no results) or by the processor *Jfem* (contains solution results). Primary results (displacements) are obtained at nodes of the finite element model. Secondary finite element results (like stresses), which are expressed through derivatives of the primary results, usually have the best precision at some points inside elements. For models composed of 20-node finite elements stresses have the most precise values at $2 \times 2 \times 2$ reduced Gaussian integration points. If two-dimensional 8-node elements are used, the best values of stresses are at 2×2 Gauss integration points.

The visualization algorithm consists of the following main steps [Nikishkov (2003)].

- Obtain continuous field of finite element results by extrapolation from reduced integration points inside elements to element nodes with subsequent averaging.
- Create the surface of the finite element model or create model section where results will be displayed.
- Subdivide curved element faces into flat triangles on the basis of surface curvature and gradient of results.
- Create contour pictures by specifying coordinates of one-dimensional color pattern at triangle vertices.

In order to obtain continuous stress fields, stresses at reduced integration points are extrapolated to finite element nodes and are averaged with the use of contributions from adjacent finite elements. After this, nodal

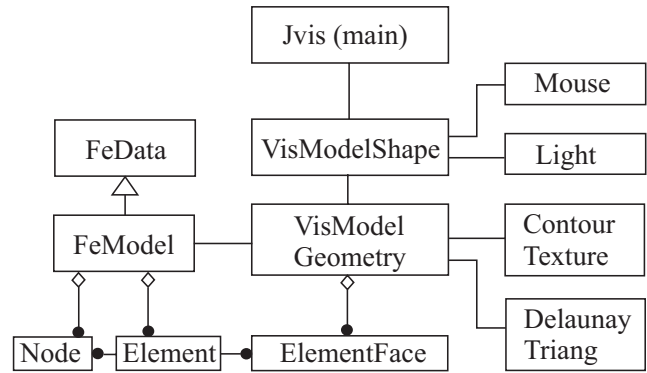


Figure 4: Class diagram of the finite element postprocessor.

stresses can be interpolated inside elements using element shape functions.

Creation of the model surface is based on the fact that outer element faces are present in the model only once while inner faces belong to two finite elements. The surface of the finite element model is created from element faces, which are mentioned in element connectivities one time.

Subdivision of quadratic element faces depends on two factors: curvature of the surface and range of result function over the surface. The subdivision into triangular elements is performed using the Delaunay triangulation procedure. Numbers of subdivisions on face edges are determined by its curvature and by results ranges between nodes.

Java 3D provides three-dimensional rendering of polygons with a possibility of texture interpolation. Texture interpolation technique is employed to create color contours inside triangles produced after subdivision of curved element surfaces. A one-dimensional texture containing desired number of color bands is generated. Values of functions at triangular vertices are transformed to texture coordinates using specified scale. These texture coordinates are supplied to the Java 3D rendering engine, which generates a three-dimensional image.

5.3 Visualization code structure

Class structure of the visualization code is shown in Fig. 4. The code serves as a model visualizer and as a postprocessor of the finite element results.

Main class *Jvis* creates object *VisModelShape*. This object contains all necessary information for Java 3D

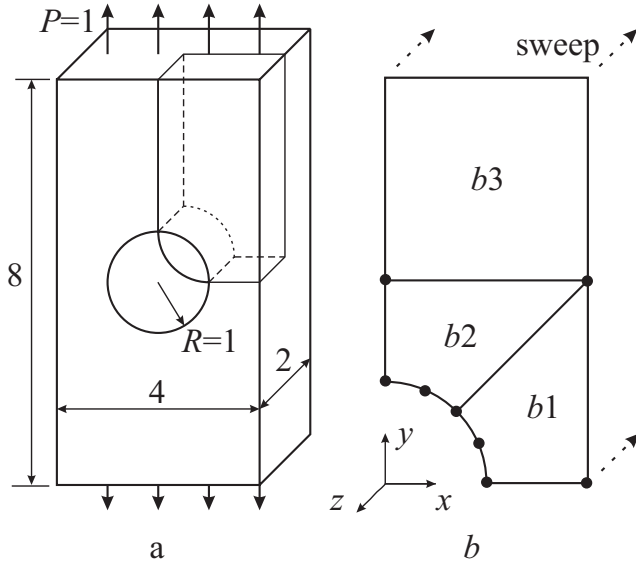


Figure 5: Example problem: a) tensile plate with a hole; b) mesh generation using block decomposition and sweeping.

scene graph including visual model shape, mouse behavior (class *Mouse*) and lights (class *Light*). Class *VisModelGeometry* creates geometric arrays for the visual model using data of the finite element model *FeModel*. Class *DelaunayTriang* helps to triangulate element faces *EminentFace* using Delaunay triangulation algorithm. Class *ContourTexture* create textures necessary for visualization of finite element results as color contours.

6 Example

Let us demonstrate the use of the finite element Java system *Jfea* on a simple example shown in Fig. 5.a: tension of a thick rectangular plate with a central hole.

6.1 Mesh generation

Mesh is generated for one eights of the specimen (see Fig. 5.a). A schematic of mesh generation is depicted in Fig. 5.b. A two-dimensional area is decomposed into blocks *b1*, *b2* and *b3* of simple shapes. Two-dimensional meshes inside blocks are created by local mesh generators. Block are pasted together in a mesh. Subsequent sweeping produces a three-dimensional mesh. Two-dimensional meshes are composed of 8-node quadrilateral elements. The 20-node hexahedral element is used

in the three-dimensional mesh. Input data for preprocessor *Jmgen* is given in Table 3. The first line in the Table is a comment. A rule for comments is as follows: after token #, the rest of the line is ignored. Uppercase and lowercase characters can be mixed freely since all characters are transformed to lowercase during data interpretation.

```
# Data file for Jmgen
GenQuad8 b1
  nh = 4  nv = 4
  xyp = 1 0 0 0 2 0 0 0
        2 2 0 0 .7071 .7071 .9239 .3827
  res = .15 .15 0.85 0
end
GenQuad8 b2
  nh = 4  nv = 3
  xyp = .7071 .7071 0 0 2 2 0 0
        0 2 0 0 0 1 .3827 .9239
  res = .15 0 0.85 0
end
Paste b1 b2 b12
  eps = 0.01
end
Rectangle b3
  nx = 3  ny = 3
  xs = 0 0.6667 1.3333 2
  ys = 2 2.6667 3.3333 4
end
Paste b12 b3 b123
  eps = 0.1
end
Sweep b123 m3d
  nlayers = 4
  zlay = 0 .25 .5 .75 1
end
WriteMesh m3d example.mesh
```

Table 3: Data file for mesh generation.

Mesh blocks *b1* and *b2* are produced by *GenQuad8* generator. Key points shown in Fig. 5.b by dark circles are specified in array *xyp* to define curved quadrilaterals. Relative sizes of smallest elements on edges are determined by values in array *res*. Free format data input provide a possibility to use default values of parameters. For example, it is not necessary to specify array *res* if all elements should have equal size along edges of a quadrilateral block. In the absence of array *res* default zero values are adopted that means equal element size along edges.

Mesh blocks *b1* and *b2* are connected together by *Paste* module. The resulting mesh is stored under the

name *b12*. Parameter *eps* determines coordinate tolerance for joining nodes from two mesh blocks.

Upper block *b3* is meshed by module *Rectangle*, which creates a mesh inside a rectangular block using specified locations of corner nodes at block edges. Another pasting of mesh blocks *b12* and *b3* produces a final two-dimensional mesh *b123*.

A three-dimensional mesh *m3d* is created by sweeping the two-dimensional mesh *b123* along the negative direction of *z* axis.

Resulting three-dimensional mesh is written to file *example.mesh* using module *WriteMesh*. The mesh is used by the processor code *Jfem* during problem solution. The mesh can be visualized by the visualization code *Jvis*.

```
# Data file for Jfem
stressState = threeD
includeFile example.mesh
solver = LDU
# Material properties
materNo = 1
elastModulus = 1000
poissonRatio = 0.3
# Displacement boundary conditions
boxConstrDispl = x 0.0
-0.01 0.99 -1.01 0.01 4.01 0.01
boxConstrDispl = y 0.0
0.99 -0.01 -1.01 2.01 0.01 0.01
boxConstrDispl = z 0.0
-0.01 -0.01 -1.01 2.01 4.01 -0.99
# Load
loadStep = 1
boxSurForce = n 1.0
-0.01 3.99 -1.01 2.01 4.01 0.01
```

Table 4: Data file for finite element solution.

6.2 Problem solution

Input data for the finite element processor is presented in Table 4. Small amount of data is necessary since the mesh prepared by the preprocessor is included using instruction `includeFile`. A principle of adopting default values of many parameters is also contribute to the reduction of input data. For example, the instruction

```
solver = LDU
```

specifies that LDU method is employed for the solution of the finite element equation system. This instruction can be omitted since LDU solver is the default one.

Next, material properties are specified. For material number `materNo` we determine just mechanical constants that are necessary for the selected type of analysis.

Different options exist for the specification of boundary conditions. In the example, both displacement boundary conditions and force boundary conditions are generated on surfaces, which are identified by a bounding box with given diagonal ends. Instruction `boxConstrDispl` implies that the following data is given: constraint direction, constraint value, three coordinates of the first diagonal end and three coordinates of the second diagonal end. Specification of a normal distributed force on a surface is performed in an analogous way.

Computing time for the example problem (148 20-node elements and 907 nodes) is 1 s on a PC computer with Intel Pentium 4 2.8MHz processor.

6.3 Visualization of results

The created finite element model and results of problem solution are visualized using postprocessor *Jvis* as shown in Fig. 6. Using mouse the user can rotate, zoom and pan the finite element model. The user interface is intentionally created simple. Parameters related to what and how to be visualized are given on the command line or in data file. Possible ways of visualization include: model, deformed model, contour plots of various quantities for deformed and undeformed models.

7 Conclusion

In this paper, the architecture of a finite element system written in Java has been presented. The system is designed according to object oriented principles. Is is organized into ten Java packages. Four packages are shared between all three codes, other six packages are specific for applications. The system has three main methods corresponding to three applications: preprocessor for mesh generation, processor for stress analysis and postprocessor for visualization.

We find that new features of Java 1.5 like data scanner and enumerated type are very helpful in development of finite element applications. Java 3D API provides means for easy visualization of finite element models and results. Acceptable computational efficiency of the Java code can be achieved with solver code tuning.

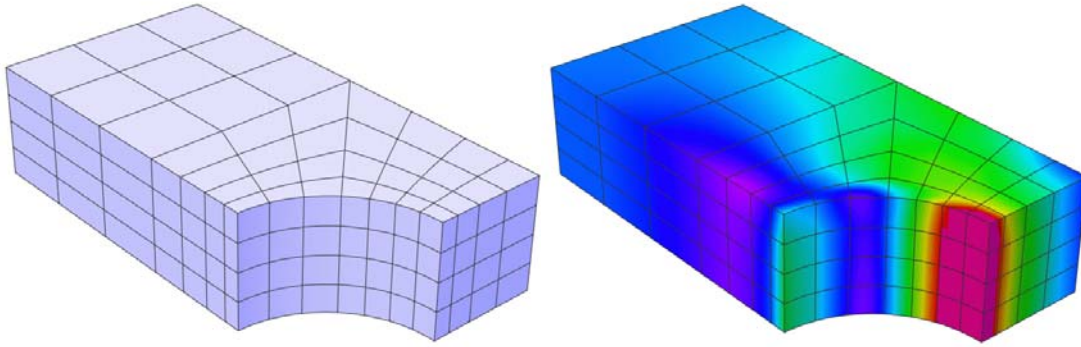


Figure 6: Visualization of the generated finite element mesh and graphical presentation of results (equivalent stress) as contours.

The general conclusion is that object oriented approach with programming in Java allows to develop well-organized finite element applications with acceptable computational efficiency. While here the object oriented design has been applied to the traditional finite element algorithm, it can be also used for codes based on meshless methods.

References

- Akin, J.E.; Singh, M.** (2002): Object-oriented Fortran 90 P-adaptive finite element method. *Advances in Engineering Software*, 33, 461-468.
- Archer, G.C.; Fennes, G.; Thewalt, C.** (1999): A new object-oriented finite element analysis program architecture. *Computers and Structures*, 70, 63-75.
- Atluri, S.N.; Shen, S.** (2002): The Meshless Local Petrov-Galerkin (MLPG) Method: A Simple and less-costly alternative to the finite element and boundary element methods. *CMES*, 3, 11-52.
- Atluri, S.N.** (2004): *The Meshless Method (MLPG) for Domain and Bie Discretizations*. Tech Science Press, Los Angeles.
- Bathe, K.-J.** (1996): *Finite Element Procedures*. Prentice-Hall, Englewood Cliffs, NJ.
- Commend, S.; Zimmermann, T.** (2001): Object-oriented nonlinear finite element programming: a primer. *Advances in Engineering Software*, 32, 611-628.
- Dubois-Pelerin, Y.; Zimmermann, T.** (1993): Object-oriented finite element programming. III. An efficient implementation in C++. *Computer Meth. Appl. Mech. Eng.* 108, 165-183.
- Eyheramendy, D.; Guibert D.** (2004): A Java approach for F.E. computational mechanics. *ECCOMAS 2004 (P.Neittaanmaki et al eds.)*, 13 p.
- Forde, B.W.R.; Foschi, R.O.; Steimer, S.F.** (1990): Object-oriented finite element analysis. *Computers and Structures*, 34, 355-374.
- Mackie, R.I.** (2001): *Object oriented methods and finite element analysis*. Saxe-Coburg Publications, Stirling, Scotland.
- Mackerle, J.** (2004): Object-oriented programming in FEM and BEM: a bibliography (1990-2003). *Advances in Engineering Software*, 35, 325336.
- Nikishkov, G.P.** (2003): Generating contours on FEM/BEM higher-order surfaces using Java 3D textures. *Advances in Engineering Software*, 34, 469-476.
- Nikishkov, G.P.; Nikishkov, Yu.G.; Savchenko, V.V.** (2003): Comparison of C and Java performance in finite element computations. *Computers and Structures*, 81, 2401-2408.
- Schneiders, R.** (2000): Quadrilateral and hexahedral element meshes. In: *Handbook of Grid Generations (J.F.Thompson et al Eds, CRC Press, 21.1-26.*
- Schroeder, W.; Martin, K.; Lorensen, B.** (1998): *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, Englewood Cliffs, NJ, USA.
- Sowizral, H.; Rushforth, K.; Deering, M.** (2000): *The Java 3D API Specification*. Addison-Wesley, Reading, MA.
- Zimmermann, T.; Dubois-Pelerin, Y.; Bomme, P.** (1992): Object-oriented finite element programming. I. Governing principles. *Computer Meth. Appl. Mech. Eng.* 98, 291303.