# Summary of FEM with IMPACT

Summary by Christopher Sewell; Program by Jonas Forssell et. al.

Impact is a Finite Element Code which is based on an explicit time stepping algorithm. These kind of codes are used to simulate dynamic phenomena involving large deformations (such as car crashes). The explicit code is based on the simple formula of F=M*A where F represents a force, M is the mass of a body and A is the resulting acceleration of that body. All the code does is calculate the acceleration for the body and use a small step in time to translate this acceleration into a little displacement of the body. This displacement is then used to calculate a responding force since the body is elastic and can be stretched (thus creating a reaction force). This force is then used to calculate an acceleration and then the process is repeated again from the beginning. Impact is designed to be compatible with the GiD pre-and post processor (the program used to make a model and to look at the results after Impact has simulated the problem).

Three simple steps to do an FEM simulation:

1) Draw (or import) your model in a pre-processor program, such as GiD (in pre-processor mode). This defines the elements and their initial (undeformed) locations. Also specify constraints, such as boundary conditions (i.e., this node stays fixed in space), and material properties. Output a file in a specific format, such as Fembic.
2) Feed your FEM simulator, such as Impact, this file, and let it run on your computer farm for hours, days, weeks… It will write an output file containing the deformations of each node at specified time intervals, as well as stresses and strains for each element.
3) Load this file into a post-processor program, such as GiD (in post-processor mode). Play the animation.

GiD is apparently a fairly popular FEM pre- and post- processor; its web page is at:

http://www.gid-usa.com/index.html

Impact's web page is:

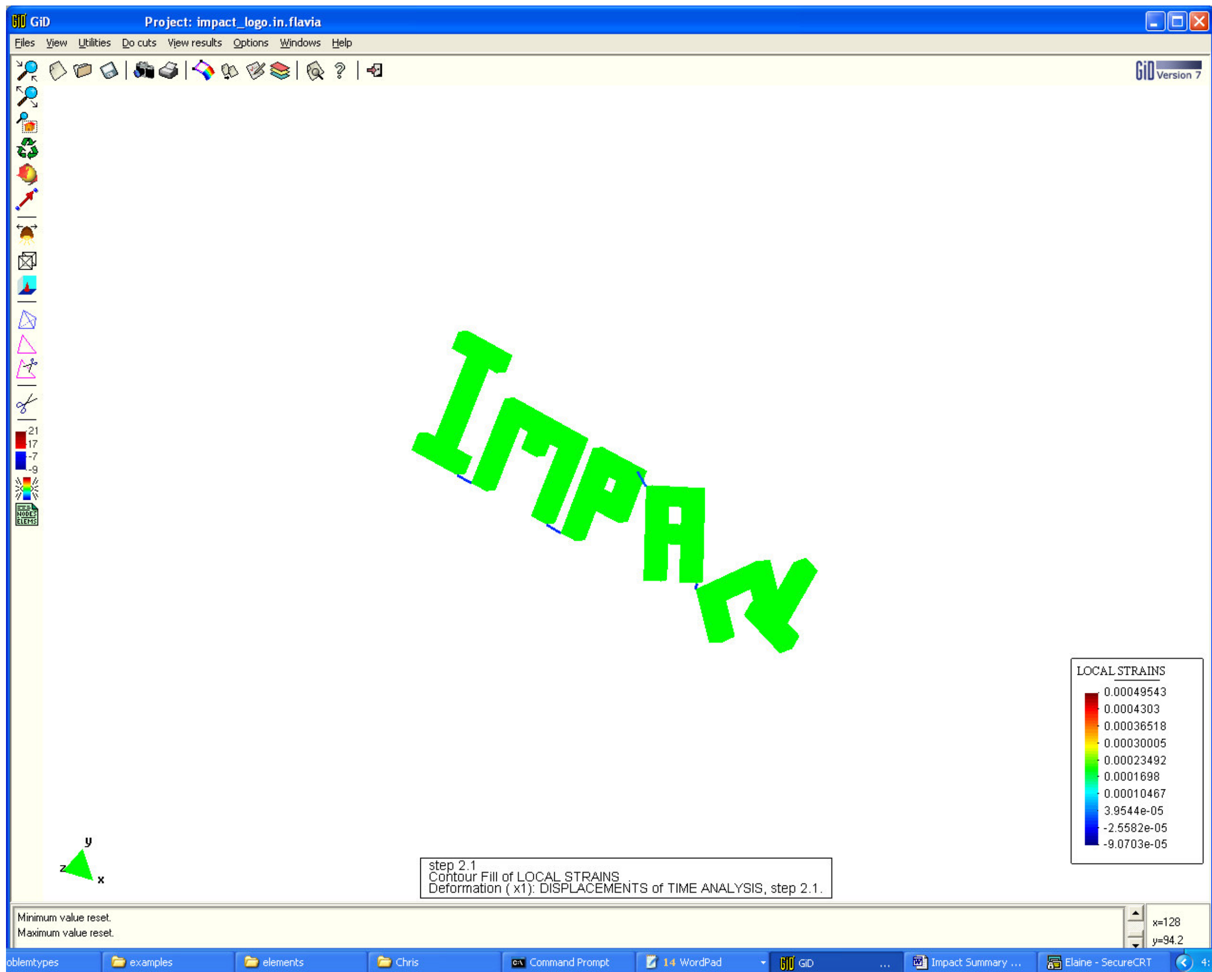
http://impact.sourceforge.net/index.html

There's even a paper on Impact, presented at the Second Annual Conference on Advances and Applications of GiD (see how big GiD is?):

http://impact.sourceforge.net/Impact_and_GiD_integration_paper.pdf

# Pre-processing

Here is a condensed sample input file in Fembic format. It can be written directly with an ASCII editor, or generated using an FEM "preprocessor" program, such as GiD, which is basically like a light version of 3D Studio Max with functionality to specify relevant material properties and output in Fembic format. The "hard problem" in preprocessing is generating a "good" mesh of finite elements if you need to improve accuracy, speed, and stability of the simulation. (How to optimize the discretation of the mesh is not the topic of Impact nor of this summary; we're just using simply created meshes and hoping they're good enough.) This sample input file contains a model of the letters spelling "IMPACT" (see picture of a frame from the final deformation simulation on the next page), with the letters made out of 8-integration point hexahedron elements (cubes with nodes at each corner) and connected to each other with 2-integration point rod elements (rods of a fixed diameter with nodes at both end points). The model is fixed in the upper left corner (boundary constraint) and pulled by a constant force in the lower right corner (load). In the simulation, the load pulls the 'T', which deforms the rest of the letters.




```
nodes
1 x = 0.0 y = 0.0 z = 0.0
2 x = 5.0 y = 0.0 z = 0.0
3 x = 10.0 y = 0.0 z = 0.0
. . .
20 x = 120.0 y = 0.0 z = 0.0 load = end_load
. . .
1131 x = 0.0 y = 35.0 z = -5.0 constraint = fixed
1132 x = 5.0 y = 35.0 z = -5.0
1133 x = 10.0 y = 35.0 z = -5.0
1134 x = 15.0 y = 35.0 z = -5.0

elements of type solid_iso_6
1 nodes = [1,2,1002,1001,21,22,1022,1021] nip = 8 material = e_steel
2 nodes = [2,3,1003,1002,22,23,1023,1022] nip = 8 material = e_steel
. . .
62 nodes = [100,101,1101,1100,124,125,1125,1124] nip = 8 material = e_steel
63 nodes = [101,102,1102,1101,125,126,1126,1125] nip = 8 material = e_steel

elements of type rod_2
64 nodes = [4,5] D = 5.0 material = e_steel
. . .
68 nodes = [122,123] D = 5.0 material = e_steel
```

```
constraints of type boundary_condition
fixed Vx = 0 Vy = 0 Vz = 0 Ax = 0 Ay = 0 Az = 0

materials of type elastoplastic
e_steel rho = 0.0000078 E = 210 nu = 0.3 yield_stress = 0.180 EP = 0.1

# There is also an elastic material available as follows:
# material of type elastic
# e_steel rho = 0.0000078 E = 210 nu = 0.3

loads
end_load Fx = 0.1 Fy = 0 Fz = 0.1 Mx = 0 My = 0 Mz = 0

controls
run from 0 to 20 step 0.0001
print every 0.1 step
```

# Processing

The open-source Impact code is really very good for learning about FEM. Simplicity and understandability are stated goals of Impact, although the full source is still many thousands of lines. However, I've cut out all the stuff dealing with parsing input files and writing output files, eliminated some of the optional and more complicated features, obvious variable declarations, get and set functions, etc., simplified some procedures, and added/deleted/modified comments, bringing it down to about 35 pages of concise, fairly understandable code. It's in Java, somewhat unusual for a computationally-intensive program, but they adhered well to object-orientation principles, so it's easy to learn what the progam does at any desired level of abstraction, learning or leaving as black boxes whatever components you are or are not interested in the details of.

The main function is in the class Impact, which gives a very, very high-level view of what the program does – it creates an instance of the class Smack (not sure why it's called this; a more appropriate name might be Simulator or something), initializes it, assembles mass matrices, sets initial conditions, and solves. Looking then into the Smack class, we see that the solve method steps from the initial time to the final time by increments of timestep. At each time step, it starts by looping through each element, calculating its strain and stress for each integration point (node) and calculating the nodal and external forces. It then updates all constraints, and loops through all nodes, calculating their new positions and checking for collisions. The details of calculating strains and stresses are in the subclasses of class Element. The simplest subclass is for rods with two integration points, Rod_2. These methods, in turn, depend on subclasses of class Material. The simplest subclass is for purely elastic materials, class Elastic. The details of calculating the new nodal positions (using a = f/m and the forces placed on the nodes by the elements) are in class Node. Positions may be affected by constraints, with implementations in subclasses of class Constraint. One common such subclass is class BoundaryConstraint. Elements are usually associated with an instance of class Contact_Line, which handles the details of collision detection, adding a repulsive force to nodes when elements come into contact. Simulation parameters parsed from the input file, such as timestep size, frame rate, etc. are stored in class ControlSet (not shown in this document).

The classes mentioned above provide an essentially complete FEM simulator, using simple elements (1D rods) and materials (elastic). At the end of this section are a couple of the more complicated subclasses: class Solid_Iso_6, a subclass of Element for hexahedrons with eight integration points (3D cubes with nodes at the corners), and class Elastoplastic, a subclass of Material for elastoplastic materials. (Both of these subclasses are used for the "IMPACT" letters example.)

```
public class Impact
{
    public static void main(java.lang.String[] args)
    {
        Smack smackinstance = new Smack();
        smackinstance.initialize(args[0]);
        smackinstance.assembleMassMatrix();
        smackinstance.setInitialConditions();
        smackinstance.solve();
    }
}


public class Smack
{
    public void initialize(String fname)
    {
        filename = new String(fname);
        Reader indatafile = new FembicReader(filename);

        number_of_elements = indatafile.numberOfElements();
        number_of_materials = indatafile.numberOfMaterials();
        number_of_nodes = indatafile.numberOfNodes();
        number_of_constraints = indatafile.numberOfConstraints();
        number_of_loads = indatafile.numberOfLoads();
        number_of_controls = indatafile.numberOfControls();

        elementlist = new Vector(number_of_elements);
        materiallist = new Vector(number_of_materials);
        nodelist = new Vector(number_of_nodes);
        constraintlist = new Vector(number_of_constraints);
        loadlist = new Vector(number_of_loads);
        controlset = new Controlset();

        for (i = 0; i < number_of_constraints; i++)
            constraintlist.addElement(
                    indatafile.getNextConstraint(nodelist));
        for (i = 0; i < number_of_loads; i++)
            loadlist.addElement(indatafile.getNextLoad());
        for (i = 0; i < number_of_nodes; i++)
            nodelist.addElement(
                    indatafile.getNextNode(constraintlist, loadlist));
        for (i = 0; i < number_of_materials; i++)
            Material m = indatafile.getNextMaterial();
            materiallist.addElement(m);
        for (i = 0; i < number_of_elements; i++)
            elementlist.addElement(
                    indatafile.getNextElement(materiallist, nodelist,
                                              loadlist));
        for (i = 0; i < number_of_controls; i++)
            indatafile.getControlSet(controlset);
        resultwriter = indatafile.getWriter(
                    nodelist, elementlist, controlset);
        resultwriter.initialize();
    }
```

```
public void assembleMassMatrix()
{
    for (i = 0; i < number_of_elements; i++)
    {
        temp_element = (Element) elementlist.elementAt(i);
        temp_element.assembleMassMatrix();
    }
    for (i = 0; i < number_of_nodes; i++)
    {
        temp_node = (Node) nodelist.elementAt(i);
        temp_node.determineMassMatrix();
    }
    for (i = 0; i < number_of_constraints; i++)
    {
        temp_constraint = (Constraint) constraintlist.elementAt(i);
        temp_constraint.determineMassMatrix(nodelist);
    }
}


public void setInitialConditions()
{
    for (i = 0; i < number_of_nodes; i++)
    {
        temp_node = (Node) nodelist.elementAt(i);
        temp_node.setInitialConditions();
    }
    for (j = 0; j < number_of_nodes; j++)
    {
        temp_node = (Node) nodelist.elementAt(j);
        if (j < (number_of_nodes - 1))
            temp_node.setRight_neighbour((Node)
                               nodelist.elementAt(j + 1));
        if (j > 0)
            temp_node.setLeft_neighbour((Node)
                               nodelist.elementAt(j - 1));
    }

    // Sort nodes:
    for (j = 0; j < number_of_nodes; j++)
    {
        for (i = 0; i < number_of_nodes; i++)
        {
            temp_node = (Node) nodelist.elementAt(i);
            temp_node.checkNeighbours();
        }
    }
    for (i = 0; i < number_of_elements; i++)
    {
        temp_element = (Element) elementlist.elementAt(i);
        temp_element.setInitialConditions();
    }
    for (i = 0; i < number_of_constraints; i++)
    {
        temp_constraint = (Constraint) constraintlist.elementAt(i);
```

```
        temp_constraint.setInitialConditions();
    }
    controlset.setInitialConditions();
    timestep = controlset.getTimestep();
    total_mass = 0;
    for (i = 0; i < number_of_nodes; i++)
    {
        temp_node = (Node) nodelist.elementAt(i);
        total_mass += temp_node.getMass();
    }
}


public void solve()
{
    int j;
    double ttemp = 1E10;
    boolean firsttime = true;
    int number_of_integration_points;

    time_info = new Date().getTime();

    for (time = controlset.getStarttime();
         time < controlset.getEndtime(); time += timestep)
    {
        for (i = 0; i < number_of_elements; i++)
        {
            temporary_element = (Element) elementlist.elementAt(i);
            temporary_element.updateLocalCoordinateSystem();
            number_of_integration_points =
                  temporary_element.getNumberOfIntegrationPoints();
            for (j = 0; j < number_of_integration_points; j++)
            {
                temporary_element.calculateStrain(timestep, j);
                temporary_element.calculateStress(j, timestep);
            }
            for (j = 0; j < number_of_integration_points; j++)
                temporary_element.calculateNodalForces(j);
            temporary_element.calculateExternalForces(time);
            temporary_element.calculateContactForces();
        }
        for (i = 0; i < number_of_constraints; i++)
        {
            temporary_constraint = (Constraint)
                             constraintlist.elementAt(i);
            temporary_constraint.update();
        }
        for (i = 0; i < number_of_nodes; i++)
        {
            temporary_node = (Node) nodelist.elementAt(i);
            temporary_node.calculateNewPosition(timestep, time);
            temporary_node.checkNeighbours();
        }

        if (controlset.timeToPrint(time))
            resultwriter.write(filename, time);
```

```
            for (i = 0; i < number_of_nodes; i++)
            {
                temporary_node = (Node) nodelist.elementAt(i);
                temporary_node.clearNodalForces();
            }
        }
    }
}
```

```
public abstract class Element
{
    public abstract void assembleMassMatrix();
    public abstract void calculateContactForces();
    public abstract void calculateExternalForces(double currtime);
    public abstract void calculateNodalForces(int j);
    public abstract void calculateStrain(double timestep, int i);
    public abstract void calculateStress(int i, double timestep);
    public abstract double checkTimestep(double current_timestep);
    public abstract int getNumberOfIntegrationPoints();
    public abstract void setInitialConditions();
    public abstract void updateLocalCoordinateSystem();

    // Set up local coordinate system: origin at first point, x-axis
    // from origin to second point, z-axis normal to plane of the three
    // points, y-axis orthogonal to x-axis and z-axis
    public void calculateLocalBaseVectors(
            double x1, double y1, double z1,
            double x2, double y2, double z2,
            double x3, double y3, double z3, Jama.Matrix bvs)
    {
        // Define the local x-axis.
        bvs.set(0, 0, x2 - x1);
        bvs.set(1, 0, y2 - y1);
        bvs.set(2, 0, z2 - z1);

        // Normalize
        bvs.set(0, 1, Math.sqrt((bvs.get(0, 0) * bvs.get(0, 0)) +
                                (bvs.get(1, 0) * bvs.get(1, 0)) +
                                (bvs.get(2, 0) * bvs.get(2, 0))));
        bvs.set(0, 0, bvs.get(0, 0) / bvs.get(0, 1));
        bvs.set(1, 0, bvs.get(1, 0) / bvs.get(0, 1));
        bvs.set(2, 0, bvs.get(2, 0) / bvs.get(0, 1));

        // Define temporary axis (here disguised as y-axis)
        bvs.set(0, 1, x3 - x1);
        bvs.set(1, 1, y3 - y1);
        bvs.set(2, 1, z3 - z1);

        // Calculate the z-axis as cross product of x and y axes
        bvs.set(
         0,2,(bvs.get(1,0)*bvs.get(2,1))-(bvs.get(2,0)*bvs.get(1,1)));
        bvs.set(
         1,2,(bvs.get(2,0)*bvs.get(0,1))-(bvs.get(0,0)*bvs.get(2,1)));
        bvs.set(
         2,2,(bvs.get(0,0)*bvs.get(1,1))-(bvs.get(1,0)*bvs.get(0,1)));
```

```
        // Normalize
        bvs.set(0, 1, Math.sqrt((bvs.get(0, 2) * bvs.get(0, 2)) +
                                (bvs.get(1, 2) * bvs.get(1, 2)) +
                                (bvs.get(2, 2) * bvs.get(2, 2))));
        bvs.set(0, 2, bvs.get(0, 2) / bvs.get(0, 1));
        bvs.set(1, 2, bvs.get(1, 2) / bvs.get(0, 1));
        bvs.set(2, 2, bvs.get(2, 2) / bvs.get(0, 1));

        // Calculate the y-axis as cross product of z and x axes
        bvs.set(
         0,1,(bvs.get(1,2)*bvs.get(2,0))-(bvs.get(2,2)*bvs.get(1,0)));
        bvs.set(
         1,1,(bvs.get(2,2)*bvs.get(0,0))-(bvs.get(0,2)*bvs.get(2,0)));
        bvs.set(
         2,1,(bvs.get(0,2)*bvs.get(1,0))-(bvs.get(1,2)*bvs.get(0, 0)));
    }
}
```

**public class Rod_2 extends Element**

```
{
    public Rod_2()
    {
        super();
        type = new String("ROD_2");
        inertia = new Jama.Matrix(3, 3);
        force = new Jama.Matrix(3, 1);
        stress = new Jama.Matrix(6, 1);
        strain = new Jama.Matrix(6, 1);
        dstrain = new Jama.Matrix(6, 1);
        local_coordinate_system = new Jama.Matrix(3, 3);
        processed = false;
    }


    public void assembleMassMatrix()
    {
        // Calculate element mass distribution
        initial_cross_section_area = (Math.PI*diameter*diameter)/4;
        cross_section_area = initial_cross_section_area;
        initial_length = java.lang.Math.sqrt(
                ((node1.getX_pos() - node2.getX_pos()) *
                 (node1.getX_pos() - node2.getX_pos())) +
                ((node1.getY_pos() - node2.getY_pos()) *
                 (node1.getY_pos() - node2.getY_pos())) +
                ((node1.getZ_pos() - node2.getZ_pos()) *
                 (node1.getZ_pos() - node2.getZ_pos())));
        mass = material.getDensity() * cross_section_area *
               initial_length;

        // Distribute the mass, half to each node.
        node1.addMass(mass / 2.0);
        node2.addMass(mass / 2.0);

        // Now, calculate the element local inertia
        // For a rod, there are two different inertias;
        // 1. Rotational inertia around the center longitudinal axis
```

```
    // 2. Rotational inertia around the middle point of the bar.
    // We will call them I1 and I2. Note that I3 is same as I2.
    this.setI1((mass * diameter * diameter) / 8);
    this.setI2((mass * initial_length * initial_length) / 12);
    this.setI3(this.getI2());

    // Now, set up the local coordinate system for the element.
    // The local x-axis will run from node 1 to node 2. The local
    // y-axis will be in the plane given by the local x-axis
    // and the point formed by the coordinates of node 1, but with
    // and offset of 1 mm in x and y direction.
    // The local z-axis will be orthogonal to the same plane.
    // In the case of a rod, the directions of the y-and z-axis are
    // really not that important since it is symmetric.
    this.updateLocalCoordinateSystem();

    // The local system has now been determined. It is unified in
    // length. The forces and inertias can now be expressed
    // relative to this  system and then transformed.
    // Transform the inertias and add half to each node.
    inertia = local_coordinate_system.times(inertia);
    node1.addInertia(inertia.times(0.5));
    node2.addInertia(inertia.times(0.5));

    // Set up mass matrix for collision element.
    internal_contact_element.assembleMassMatrix();
}


// The contact forces are calculated here. Let this element check
// it's segment(s) against any neighboring nodes.  Calculate
// distance from segment to node if distance < tolerance; then
// reactionforce = f(tolerance), and add this reactionforce to the
// element's nodes.  Continue this check until all the nodes within
// the contact tolerance have been checked.
public void calculateContactForces()
{
    internal_contact_element.calculateContactForces();
}


public void calculateExternalForces(double currtime)
{
    // Calculate gravity and load forces; convert, add to node.
    internal_contact_element.calculateExternalForces(currtime);
}


public void calculateNodalForces(int integration_point)
{
    Jama.Matrix global_force;
    force.set(0, 0, stress.get(0, 0) * cross_section_area);
    force.set(1, 0, 0);
    force.set(2, 0, 0);

    // Transform to global coordinates
    global_force = local_coordinate_system.times(force).copy();
```

```
    // Add this force contribution to the node
    node1.addInternalForce(global_force.times(1));
    node2.addInternalForce(global_force.times(-1));
}


public void calculateStrain(double timestep, int integration_point)
{
    xpos1 = node1.getX_pos();
    ypos1 = node1.getY_pos();
    zpos1 = node1.getZ_pos();

    xpos2 = node2.getX_pos();
    ypos2 = node2.getY_pos();
    zpos2 = node2.getZ_pos();

    new_length = java.lang.Math.sqrt(
            ((xpos2 - xpos1) * (xpos2 - xpos1)) +
            ((ypos2 - ypos1) * (ypos2 - ypos1)) +
            ((zpos2 - zpos1) * (zpos2 - zpos1)));
    dstrain.set(0, 0,
        Math.log(1 + ((new_length-initial_length)/initial_length))-
                 strain.get(0, 0));

    // Calculate new cross section area (incompressible material)
    cross_section_area = (initial_cross_section_area *
                          initial_length) / new_length;
}


public void calculateStress(int integration_point, double timestep)
{
    // Normally, this would be a matrix (but not for a rod)
    material.calculateStressOneDimensional(
                     strain, dstrain, stress, timestep);
}


public void setInitialConditions()
{
    material = (Material) material.clone();
    material.setInitialConditions();
    internal_contact_element.setInitialConditions();
}


public void updateLocalCoordinateSystem()
{
    super.calculateLocalBaseVectors(
        0, 0, 0,
        node2.getX_pos() - node1.getX_pos(),
        node2.getY_pos() - node1.getY_pos(),
        node2.getZ_pos() - node1.getZ_pos(),
        node2.getZ_pos() - node1.getZ_pos(),
        node1.getX_pos() - node2.getX_pos(),
        node2.getY_pos() - node1.getY_pos(),
```

```
            local_coordinate_system);
        internal_contact_element.updateLocalCoordinateSystem();
    }
}


public abstract class Material implements Cloneable
{
    public abstract void calculateStressOneDimensional(
        Jama.Matrix strain, Jama.Matrix dstrain, Jama.Matrix stress,
        double timestep);
    public abstract void calculateStressThreeDimensional(
        Jama.Matrix strain, Jama.Matrix dstrain, Jama.Matrix stress,
        double timestep);
    public abstract void calculateStressTwoDimensionalPlaneStress(
        Jama.Matrix strain, Jama.Matrix dstrain, Jama.Matrix stress,
        double timestep);
    public abstract void setInitialConditions();
    public abstract double wavespeedOneDimensional(double p,double p2);
    public abstract double wavespeedThreeDimensional(double p,
                                                     double p2);
    public abstract double wavespeedTwoDimensional(double p,double p2);
}


public class Elastic extends Material implements Cloneable
{
    public Elastic()
    {
        type = new String("ELASTIC");
        new_stress = new Jama.Matrix(6, 1);
        stiffness_matrix_3d = new Jama.Matrix(6, 6);
        stiffness_matrix_plane_stress = new Jama.Matrix(6, 6);
    }


    // This method calculates a stress based on a given strain, strain
    // increase and stress for an element.  This is the default input
    // for a material law.  For an elastic material, just multiply
    // with Young's Modulus.  All Matrices are 6x1: strain =
    // [epsilonx, epsilony, epsilonz, gammaxy, gammayz, gammazx]
    // transposed;  dstrain = same format as strain but latest
    // increment; stress = [sigmax, sigmay, sigmaz, tauxy, tauyz,
    // tauzx] transposed
    public void calculateStressOneDimensional(Jama.Matrix strain,
            Jama.Matrix dstrain, Jama.Matrix stress, double timestep)
    {
        strain.plusEquals(dstrain);
        stress.set(0, 0, strain.get(0, 0) * youngs_modulus);
    }


    public void calculateStressThreeDimensional(Jama.Matrix strain,
            Jama.Matrix dstrain, Jama.Matrix stress, double timestep)
    {
        strain.plusEquals(dstrain);
        stress.setMatrix(0,5,0,0,stiffness_matrix_3d.times(strain));
```

```java
    }


    public void calculateStressTwoDimensionalPlaneStress(
        Jama.Matrix strain, Jama.Matrix dstrain, Jama.Matrix stress,
        double timestep)
    {
        stress.setMatrix(0,5,0,0,
                       stiffness_matrix_plane_stress.times(strain));

        // To predict thinning of element, update the strain increment
        // in thickness direction as an isotropic
        dstrain.set(2, 0, -(nu / (1 - nu)) * (dstrain.get(0, 0) +
                    dstrain.get(1, 0)));

        // Update the strain
        strain.plusEquals(dstrain);
    }


    public void setInitialConditions()
    {
        double c = youngs_modulus / ((1.0 + nu) * (1.0 - (2.0 * nu)));
        double c2 = youngs_modulus / (1.0 - (nu * nu));
        double G = youngs_modulus / (2.0 * (1.0 + nu));

        stiffness_matrix_3d.set(0, 0, (1.0 - nu) * c);
        stiffness_matrix_3d.set(0, 1, nu * c);
        stiffness_matrix_3d.set(0, 2, nu * c);
        stiffness_matrix_3d.set(1, 0, nu * c);
        stiffness_matrix_3d.set(1, 1, (1.0 - nu) * c);
        stiffness_matrix_3d.set(1, 2, nu * c);
        stiffness_matrix_3d.set(2, 0, nu * c);
        stiffness_matrix_3d.set(2, 1, nu * c);
        stiffness_matrix_3d.set(2, 2, (1.0 - nu) * c);
        stiffness_matrix_3d.set(3, 3, G / 2.0);
        stiffness_matrix_3d.set(4, 4, G / 2.0);
        stiffness_matrix_3d.set(5, 5, G / 2.0);

        stiffness_matrix_plane_stress.set(0, 0, c2);
        stiffness_matrix_plane_stress.set(0, 1, nu * c2);
        stiffness_matrix_plane_stress.set(1, 0, nu * c2);
        stiffness_matrix_plane_stress.set(1, 1, c2);
        stiffness_matrix_plane_stress.set(3, 3, ((1 - nu) * c2) / 2);
        stiffness_matrix_plane_stress.set(4, 4,
                  ((5.0 / 6.0) * (1 - nu) * c2) / 2);
        stiffness_matrix_plane_stress.set(5, 5,
                   ((5.0 / 6.0) * (1 - nu) * c2) / 2);
    }
}


public class Node
{
    // The matrices are organized as follows:
    // Force: [Fx Fy Fz Mx My Mz]
    // Position: [x y z x_rot y_rot z_rot]
```

```
// Velocity: [x_vel y_vel z_vel x_rot_vel y_rot_vel z_rot_vel]
// Acceleration: [x_acc y_acc z_acc x_rot_acc y_rot_acc z_rot_acc]
// Displacement: [x_dpl y_dpl z_dpl x_rot_dpl y_rot_dpl z_rot_dpl]
// Mass: mass
// Inertia: [Ixx Ixy Ixz]
//          [Iyx Iyy Iyz]
//          [Izx Izy Izz]
// Load is a reference to any external boundary condition applied
// to the node.  If no boundary condition exists, the load matrix
// looks like the force matrix, but is filled with zeroes.
public Node()
{
    vel = new Jama.Matrix(6, 1);
    acc = new Jama.Matrix(6, 1);
    dpl = new Jama.Matrix(6, 1);
    dpl_old = new Jama.Matrix(6, 1);
    pos = new Jama.Matrix(6, 1);
    pos_orig = new Jama.Matrix(6, 1);
    force = new Jama.Matrix(6, 1);
    internal_force = new Jama.Matrix(6, 1);
    external_force = new Jama.Matrix(6, 1);
    contact_force = new Jama.Matrix(6, 1);
    internal_force_old = new Jama.Matrix(6, 1);
    external_force_old = new Jama.Matrix(6, 1);
    contact_force_old = new Jama.Matrix(6, 1);
    force_positive = new Jama.Matrix(6, 1);
    lastload = new Jama.Matrix(6, 1);
    inertia = new Jama.Matrix(3, 3);
}


// This method updates the node position. It uses the central
// difference scheme, without damping. Note that the methods
// applyXXXXXCondition() applies the boundary condition to the
// calculations by just changing the calculated value to the one
// set in the boundary condition.
public void calculateNewPosition(double timestep, double currtime)
{
    // Store old values
    for (i = 0; i < 6; i++)
    {
        dpl_old.set(i, 0, dpl.get(i, 0));
        external_force_old.set(i, 0, external_force.get(i, 0));
        internal_force_old.set(i, 0, internal_force.get(i, 0));
        contact_force_old.set(i, 0, contact_force.get(i, 0));
    }

    // Add external forces
    if (load != null)
    {
        lastload = load.getLoad(currtime);
        force.plusEquals(lastload);
    }

    // Calculate acceleration: a = f/m
    // Linear part
    acc.setMatrix(0,2,0,0,force.getMatrix(0,2,0,0).times(1/mass));
```

```
    // Rotational accelerations
    acc.setMatrix(3,5,0,0,
           inv_inertia.times(force.getMatrix(3,5,0,0).
           minus(vel.getMatrix(3,5,0,0).
           vectorProduct(inertia.times(vel.getMatrix(3,5,0,0))))));

    // Add gravity etc.
    if (load != null)
        acc.plusEquals(load.getAcc(currtime));

    // Apply constraints
    if (constraint != null)
        constraint.applyAccelerationConditions(this, currtime);

    // Compute velocities
    vel.plusEquals(acc.times(timestep));

    // Compute displacements
    dpl.plusEquals(vel.times(timestep));

    // Apply velocity constraints
    if (constraint != null)
        constraint.applyVelocityConditions(this, currtime);

    // Calculate the energies (reverse sign for internal)
    for (i=0; i<6; i++)
    {
        external_energy += 0.5*timestep*(vel.get(i,0) *
                           external_force_old.get(i,0);
        internal_energy -= 0.5*timestep*(vel.get(i,0) *
                           internal_force_old.get(i,0);
        contact_energy += 0.5*timestep*(vel.get(i,0) *
                           contact_force_old.get(i,0);
    }
}


// This method is needed for the contact algorithm. Each node has a
// handle to the closest neighboring node to the right and to the
// left. After the position of the node has been updated, the node
// should check its neighbor to see if it has moved past it in
// space. If that is the case, the handles must be shifted and the
// node must find a new neighbor.
public void checkNeighbours()
{
    boolean finished = false;
    while (!finished)
    {
        // Assume we will finish this time
        finished = true;

        // Check if the left node is not to the left side anymore.
        if (left_neighbour != null)
        {
            if (left_neighbour.getX_pos() > this.getX_pos())
            {
```

```
                temp_neighbour=left_neighbour.getLeft_neighbour();
                left_neighbour.setRight_neighbour(right_neighbour);
                if (right_neighbour != null)
                    right_neighbour.setLeft_neighbour(left_neighbour);
                left_neighbour.setLeft_neighbour(this);
                right_neighbour = left_neighbour;
                left_neighbour = temp_neighbour;
                if (left_neighbour != null)
                    left_neighbour.setRight_neighbour(this);

                // Repeat loop to check again
                finished = false;
            } else
            {

            // Check if the right node is still to the right.
            if (right_neighbour != null)
            {
                if (right_neighbour.getX_pos() < this.getX_pos())
                {
                    temp_neighbour =
                            right_neighbour.getRight_neighbour();
                    right_neighbour.setLeft_neighbour(left_neighbour);
                    if (left_neighbour != null)
                        left_neighbour.setRight_neighbour(
                                                 right_neighbour);
                    right_neighbour.setRight_neighbour(this);
                    left_neighbour = right_neighbour;
                    right_neighbour = temp_neighbour;
                    if (right_neighbour != null)
                        right_neighbour.setLeft_neighbour(this);

                    // Repeat loop to check again.
                    finished = false;
                }
            }
        }
    }
}


public void clearNodalForces()
{
    for (i = 0; i < 6; i++)
    {
        internal_force.set(i, 0, 0);
        external_force.set(i, 0, 0);
        contact_force.set(i, 0, 0);
        force_positive.set(i, 0, 0);
        force.set(i, 0, 0);
    }
}


public void determineMassMatrix()
{
    inv_inertia = inertia.inverse();
```

```
}


public void setInitialConditions()
{
    pos_orig.set(3,0,0);pos_orig.set(4,0,0);pos_orig.set(5,0,0);

    dpl.set(0, 0, 0);dpl.set(1, 0, 0);dpl.set(2, 0, 0);
    dpl.set(3, 0, 0);dpl.set(4, 0, 0);dpl.set(5, 0, 0);

    // The velocities should be set according to any related constraint
    if (constraint == null)
    {
        this.setX_vel(0);this.setY_vel(0);this.setZ_vel(0);
        this.setX_rot_vel(0);this.setY_rot_vel(0);this.setY_rot_vel(0);
        this.setX_acc(0);this.setY_acc(0);this.setZ_acc(0);
        this.setX_rot_acc(0);this.setY_rot_acc(0);this.setY_rot_acc(0);

    } else
    {
        constraint.applyAccelerationConditions(this, 0);
        constraint.applyVelocityConditions(this, 0);
        constraint.registerNode(this);
    }

    // Old displacements are equal to start conditions
    dpl_old = dpl.copy();

    // Set initial energies
    internal_energy = 0;
    external_energy = 0;
    contact_energy = 0;
}


public Contact_Line getContact_Line(int nr)
{
    return linelist[nr];
}


public void addContact_Line(Contact_Line c_element)
{
    if (linelist == null)
    {
        linelist = new Contact_Line[1];
        linelist[0] = c_element;
    } else
    {
        Contact_Line[] tmp = linelist;
        linelist = new Contact_Line[linelist.length + 1];
        i = 0;

        // Copy all the data in the previous array (tmp)
        while (i < (linelist.length - 1)) {
            linelist[i] = tmp[i];
            i++;
```

```
            }

            // And add the last one to the end of the new array
            linelist[linelist.length - 1] = c_element;
        }
    }

    // Check if there is a contact_line element connected to the other node
    public boolean hasContact_LineElementConnectedTo(Node endnode)
    {
        // Is there any element connected to this node?
        if (linelist == null)
            return false;

        // Check all elements
        int i = 0;
        while (i < linelist.length)
        {
            if (linelist[i].getNode(0).equals(endnode))
                return true;
            if (linelist[i].getNode(1).equals(endnode))
                return true;
            i++;
        }
        return false;
    }
}
```

**public abstract class Constraint**

```
{
    public abstract void setInitialConditions();
    public abstract void applyAccelerationConditions(Node nod,
                                                     double currtime);
    public abstract void applyVelocityConditions(Node nod,
                                                     double currtime);
    public abstract void registerNode(Node nod);
    public abstract void update();
    public abstract void determineMassMatrix(java.util.Vector nodelist);


    public Jama.Matrix calculateLocalBaseVectors(
        double x1, double y1, double z1, double x2, double y2, double z2,
        double x3, double y3, double z3)
    {
        Jama.Matrix base_vector_system = new Jama.Matrix(3, 3);
        Jama.Matrix local_x_axis = new Jama.Matrix(3, 1);
        Jama.Matrix local_y_axis = new Jama.Matrix(3, 1);
        Jama.Matrix local_z_axis = new Jama.Matrix(3, 1);

        // Define the local x-axis.
        local_x_axis.set(0, 0, x2 - x1);
        local_x_axis.set(1, 0, y2 - y1);
        local_x_axis.set(2, 0, z2 - z1);

        // Define temporary axis (here disguised as y-axis)
        local_y_axis.set(0, 0, x3 - x1);
```

```
        local_y_axis.set(1, 0, y3 - y1);
        local_y_axis.set(2, 0, z3 - z1);

        // Calculate the z-axis
        local_z_axis = local_x_axis.vectorProduct(local_y_axis);

        // Calculate the y-axis
        local_y_axis = local_x_axis.vectorProduct(local_z_axis);

        // Normalize and set up the base vector system matrix.
        local_x_axis.timesEquals(1.0 / local_x_axis.length());
        local_y_axis.timesEquals(1.0 / local_y_axis.length());
        local_z_axis.timesEquals(1.0 / local_z_axis.length());
        base_vector_system.setMatrix(0, 2, 0, 0, local_x_axis);
        base_vector_system.setMatrix(0, 2, 1, 1, local_y_axis);
        base_vector_system.setMatrix(0, 2, 2, 2, local_z_axis);
        return base_vector_system;
    }
}
```

```
public class BoundaryCondition extends Constraint
{
    public void applyAccelerationConditions(Node nod, double currtime)
    {
        if (axis_is_set)
        {
            if (x_vel_is_on(currtime))
            {
                 x = 0;
            }
            else
            {
                if (x_acc_is_on(currtime))
                    x = x_acc.value(currtime);
                else
                    x = (nod.getX_acc() * axis.get(0, 0)) +
                        (nod.getY_acc() * axis.get(1, 0)) +
                        (nod.getZ_acc() * axis.get(2, 0));
            }
            if (y_vel_is_on(currtime))
            {
               y = 0;
            } else
            {
                if (y_acc_is_on(currtime))
                    y = y_acc.value(currtime);
                else
                    y = (nod.getX_acc() * axis.get(0, 1)) +
                        (nod.getY_acc() * axis.get(1, 1)) +
                        (nod.getZ_acc() * axis.get(2, 1));
            }
            if (z_vel_is_on(currtime))
            {
              z = 0;
            } else
            {
```

```
        if (z_acc_is_on(currtime))
            z = z_acc.value(currtime);
        else
            z = (nod.getX_acc() * axis.get(0, 2)) +
                (nod.getY_acc() * axis.get(1, 2)) +
                (nod.getZ_acc() * axis.get(2, 2));
    }
    nod.setX_acc((x * axis.get(0, 0)) + (y * axis.get(0, 1)) +
                 (z * axis.get(0, 2)));
    nod.setY_acc((x * axis.get(1, 0)) + (y * axis.get(1, 1)) +
                 (z * axis.get(1, 2)));
    nod.setZ_acc((x * axis.get(2, 0)) + (y * axis.get(2, 1)) +
                 (z * axis.get(2, 2)));

    if (x_rot_vel_is_on(currtime))
    {
       x = 0;
    } else
    {
        if (x_rot_acc_is_on(currtime))
            x = x_rot_acc.value(currtime);
        else
            x = (nod.getX_rot_acc() * axis.get(0, 0)) +
                (nod.getY_rot_acc() * axis.get(1, 0)) +
                (nod.getZ_rot_acc() * axis.get(2, 0));
    }
    if (y_rot_vel_is_on(currtime))
    {
         y = 0;
    } else
    {
        if (y_rot_acc_is_on(currtime))
            y = y_rot_acc.value(currtime);
        else
            y = (nod.getX_rot_acc() * axis.get(0, 1)) +
                (nod.getY_rot_acc() * axis.get(1, 1)) +
                (nod.getZ_rot_acc() * axis.get(2, 1));
    }

    if (z_rot_vel_is_on(currtime))
    {
         z = 0;
    } else
    {
        if (z_rot_acc_is_on(currtime))
            z = z_rot_acc.value(currtime);
        else
            z = (nod.getX_rot_acc() * axis.get(0, 2)) +
                (nod.getY_rot_acc() * axis.get(1, 2)) +
                (nod.getZ_rot_acc() * axis.get(2, 2));
    }
    nod.setX_rot_acc(
          (x * axis.get(0, 0)) + (y * axis.get(0, 1)) +
          (z * axis.get(0, 2)));
    nod.setY_rot_acc(
          (x * axis.get(1, 0)) + (y * axis.get(1, 1)) +
          (z * axis.get(1, 2)));
```

```
        nod.setZ_rot_acc(
             (x * axis.get(2, 0)) + (y * axis.get(2, 1)) +
             (z * axis.get(2, 2)));
    } else
    {
        if (x_vel_is_on(currtime))
            nod.setX_acc(0);
        else if (x_acc_is_on(currtime))
            nod.setX_acc(x_acc.value(currtime));
        if (y_vel_is_on(currtime))
            nod.setY_acc(0);
        else if (y_acc_is_on(currtime))
            nod.setY_acc(y_acc.value(currtime));
        if (z_vel_is_on(currtime))
            nod.setZ_acc(0);
        else if (z_acc_is_on(currtime))
            nod.setZ_acc(z_acc.value(currtime));
        if (x_rot_vel_is_on(currtime))
            nod.setX_rot_acc(0);
        else if (x_rot_acc_is_on(currtime))
            nod.setX_rot_acc(x_rot_acc.value(currtime));
        if (y_rot_vel_is_on(currtime))
            nod.setY_rot_acc(0);
        else if (y_rot_acc_is_on(currtime))
            nod.setY_rot_acc(y_rot_acc.value(currtime));
        if (z_rot_vel_is_on(currtime))
            nod.setZ_rot_acc(0);
        else if (z_rot_acc_is_on(currtime))
            nod.setZ_rot_acc(z_rot_acc.value(currtime));
    }
}


public void applyVelocityConditions(Node nod, double currtime)
{
    if (axis_is_set)
    {
        if (x_vel_is_on(currtime))
            x = x_vel.value(currtime);
        else
            x = (nod.getX_vel() * axis.get(0, 0)) +
                (nod.getY_vel() * axis.get(1, 0)) +
                (nod.getZ_vel() * axis.get(2, 0));
        if (y_vel_is_on(currtime))
            y = y_vel.value(currtime);
        else
            y = (nod.getX_vel() * axis.get(0, 1)) +
                (nod.getY_vel() * axis.get(1, 1)) +
                (nod.getZ_vel() * axis.get(2, 1));
        if (z_vel_is_on(currtime))
            z = z_vel.value(currtime);
        else
            z = (nod.getX_vel() * axis.get(0, 2)) +
                (nod.getY_vel() * axis.get(1, 2)) +
                (nod.getZ_vel() * axis.get(2, 2));
        nod.setX_vel((x * axis.get(0, 0)) + (y * axis.get(0, 1)) +
                     (z * axis.get(0, 2)));
```

```
        nod.setY_vel((x * axis.get(1, 0)) + (y * axis.get(1, 1)) +
                     (z * axis.get(1, 2)));
        nod.setZ_vel((x * axis.get(2, 0)) + (y * axis.get(2, 1)) +
                     (z * axis.get(2, 2)));

        if (x_rot_vel_is_on(currtime))
            x = x_rot_vel.value(currtime);
        else
            x = (nod.getX_rot_vel() * axis.get(0, 0)) +
                (nod.getY_rot_vel() * axis.get(1, 0)) +
                (nod.getZ_rot_vel() * axis.get(2, 0));
        if (y_rot_vel_is_on(currtime))
            y = y_rot_vel.value(currtime);
        else
            y = (nod.getX_rot_vel() * axis.get(0, 1)) +
                (nod.getY_rot_vel() * axis.get(1, 1)) +
                (nod.getZ_rot_vel() * axis.get(2, 1));
        if (z_rot_vel_is_on(currtime))
            z = z_rot_vel.value(currtime);
        else
            z = (nod.getX_rot_vel() * axis.get(0, 2)) +
                (nod.getY_rot_vel() * axis.get(1, 2)) +
                (nod.getZ_rot_vel() * axis.get(2, 2));
        nod.setX_rot_vel((x * axis.get(0, 0)) + (y * axis.get(0, 1)) +
                         (z * axis.get(0, 2)));
        nod.setY_rot_vel((x * axis.get(1, 0)) + (y * axis.get(1, 1)) +
                         (z * axis.get(1, 2)));
        nod.setZ_rot_vel((x * axis.get(2, 0)) + (y * axis.get(2, 1)) +
                         (z * axis.get(2, 2)));
    } else
    {
        if (x_vel_is_on(currtime))
            nod.setX_vel(x_vel.value(currtime));
        if (y_vel_is_on(currtime))
            nod.setY_vel(y_vel.value(currtime));
        if (z_vel_is_on(currtime))
            nod.setZ_vel(z_vel.value(currtime));
        if (x_rot_vel_is_on(currtime))
            nod.setX_rot_vel(x_rot_vel.value(currtime));
        if (y_rot_vel_is_on(currtime))
            nod.setY_rot_vel(y_rot_vel.value(currtime));
        if (z_rot_vel_is_on(currtime))
            nod.setZ_rot_vel(z_rot_vel.value(currtime));
    }
}


public void setInitialConditions()
{
}


public void update()
{
    if (axis_is_set && update_is_set)
        axis = this.calculateLocalBaseVectors(
             node[0].getX_pos(),node[0].getY_pos(),node[0].getZ_pos(),
```

```
                node[1].getX_pos(),node[1].getY_pos(),node[1].getZ_pos(),
                node[2].getX_pos(),node[2].getY_pos(),node[2].getZ_pos());
    }


    public void determineMassMatrix(java.util.Vector nodelist)
    {
        if (axis_is_set)
        {
            for (j = 0; j < 3; j++)
                node[j] = super.findNode(
                            super.getNodeNumber(j + 1, nodes), nodelist);

            axis = this.calculateLocalBaseVectors(
                 node[0].getX_pos(),node[0].getY_pos(),node[0].getZ_pos(),
                 node[1].getX_pos(),node[1].getY_pos(),node[1].getZ_pos(),
                 node[2].getX_pos(),node[2].getY_pos(),node[2].getZ_pos());
        }
    }
}
```

**public class Contact_Line extends Element**

```
{
    public Contact_Line()
    {
        node = new Node[2];
        force = new Jama.Matrix(3, 1);
        P = new Matrix(3, 3);
        a_b_distance = new Matrix(3, 1);
        v1 = new Matrix(3, 1);
        v2 = new Matrix(3, 1);
        v3 = new Matrix(3, 1);
        trash = new Matrix(3, 1);
        factor = 10;
    }


    public void assembleMassMatrix()
    {
        this.calculateLocalVariables();
    }


    public void calculateContactForces()
    {
        this.calculateLocalVariables();

        // Find out the end nodes with smallest and largest x-coordinate
        if (node[1].getX_pos() < node[0].getX_pos())
        {
            smallest = node[1];
            largest = node[0];
        } else
        {
            smallest = node[0];
            largest = node[1];
```

```
}
current = smallest;
while ((current.getLeft_neighbour() != null) &&
       (current.getLeft_neighbour().getX_pos() >
       (smallest.getX_pos()-radius)))
    current = current.getLeft_neighbour();
smallest = current;
current = largest;
while (
    (current.getRight_neighbour() != null) &&
    (current.getRight_neighbour().getX_pos() <
    (largest.getX_pos() +  radius)))
    current = current.getRight_neighbour();
largest = current;
current = smallest;

// Determine max and min coordinates of the element
y_min = Math.min(node[0].getY_pos(), node[1].getY_pos()) - radius;
y_max = Math.max(node[0].getY_pos(), node[1].getY_pos()) + radius;
z_min = Math.min(node[0].getZ_pos(), node[1].getZ_pos()) - radius;
z_max = Math.max(node[0].getZ_pos(), node[1].getZ_pos()) + radius;

// Check nodes from smallest to largest
finished = false;

while (! finished)
{
    i = 0;

    // Check nodal contact
    if (this.isInContact(current))
    {
        current.addContactForce(v3.times(factor * (1 -
               (a_b_distance.get(2, 0) / radius))));
        n1.addContactForce(v3.times(-factor * (1 -
               (a_b_distance.get(2, 0) / radius)) *
               (1 - a_b_distance.get(0, 0))));
        n2.addContactForce(v3.times(-factor * (1 -
               (a_b_distance.get(2, 0) / radius)) *
                a_b_distance.get(0, 0)));
    }

    // Check line contact with all lines connected to this node.
    Contact_Line c_element = current.getContact_Line(i);

    while (c_element != null)
    {
         n3 = c_element.getNode(0);
         n4 = c_element.getNode(1);

         // Check for contact
         if (this.isInContact(c_element))
         {
              // Calculate the force and add to the nodes.
              // The force is directed along the v3 axis.
              n1.addContactForce(v3.times(
                    -1*(1 - a_b_distance.get(0, 0)) * factor * (1 -
```

```
                        (a_b_distance.get(2, 0) / radius))));
                n2.addContactForce(v3.times(
                      -1 * a_b_distance.get(0, 0) * factor * (1 -
                      (a_b_distance.get(2, 0) / radius))));
                n3.addContactForce(v3.times(
                      (1 - a_b_distance.get(1, 0)) * factor * (1 -
                      (a_b_distance.get(2, 0) / radius))));
                n4.addContactForce(v3.times(
                      a_b_distance.get(1, 0) * factor * (1 -
                      (a_b_distance.get(2, 0) / radius))));



            }

            // Now, check next element attached to the node
            i++;
            c_element = current.getContact_Line(i);
        }

        // Now, check next node.
        if (current == largest)
            finished = true;
        else
            current = current.getRight_neighbour();
    }
}


private boolean isInContact(Node c_node)
{
    // Skip own node
    if (c_node.equals(n1) || c_node.equals(n2))
        return false;

    // Skip nodes outside the element "box"
    if ((c_node.getY_pos() < y_min) || (c_node.getY_pos() > y_max))
        return false;
    if ((c_node.getZ_pos() < z_min) || (c_node.getZ_pos() > z_max))
        return false;

    // Compute distance
    v2 = c_node.getPos().minus(n1.getPos());

    // v3 = v1.vectorProduct(v2);
    v3.set(0, 0, (v1.get(1, 0) * v2.get(2, 0)) -
        (v1.get(2, 0) * v2.get(1, 0)));
    v3.set(1, 0, (v1.get(2, 0) * v2.get(0, 0)) -
        (v1.get(0, 0) * v2.get(2, 0)));
    v3.set(2, 0, (v1.get(0, 0) * v2.get(1, 0)) -
        (v1.get(1, 0) * v2.get(0, 0)));

    a_b_distance.set(2, 0, v3.length() / l1);

    // Skip if outside distance
    if (a_b_distance.get(2, 0) > radius)
        return false;
```

```
    // v3 = v3.vectorProduct(v1);
    trash.set(0, 0, (v3.get(1, 0) * v1.get(2, 0)) -
        (v3.get(2, 0) * v1.get(1, 0)));
    trash.set(1, 0, (v3.get(2, 0) * v1.get(0, 0)) -
        (v3.get(0, 0) * v1.get(2, 0)));
    trash.set(2, 0, (v3.get(0, 0) * v1.get(1, 0)) -
        (v3.get(1, 0) * v1.get(0, 0)));
    v3.set(0, 0, trash.get(0, 0));
    v3.set(1, 0, trash.get(1, 0));
    v3.set(2, 0, trash.get(2, 0));

    if (v3.length() != 0)
        v3.timesEquals(1 / v3.length());

    v2 = v2.minus(v3.times(a_b_distance.get(2, 0)));

    // And determine a
    double a = v2.length() / l1;

    // Figure out if vector is going the other way
    v2 = v1.plus(v2);

    if (v2.length() < l1)
        a = -a;

    // Check if contact node is outside the endpoints.
    // If this is the case, make a direct radial vector
    // and recalculate the distance to eliminate nodes that
    // "slip in" from the end side.
    if (a < 0)
    {
        v3 = c_node.getPos().minus(n1.getPos());
        a_b_distance.set(2, 0, v3.length());
        if (a_b_distance.get(2, 0) > radius)
            return false;
        v3.timesEquals(1 / a_b_distance.get(2, 0));
        a = 0;
    } else if (a > 1)
    {
        v3 = c_node.getPos().minus(n2.getPos());
        a_b_distance.set(2, 0, v3.length());
        if (a_b_distance.get(2, 0) > radius)
            return false;
        v3.timesEquals(1 / a_b_distance.get(2, 0));
        a = 1;
    }
    a_b_distance.set(0, 0, a);
    return true;
}


private boolean isInContact(Contact_Line cl)
{
    // Skip own element
    if (cl.equals(this))
        return false;
```

```
// Skip nodes outside the element "box"
if ((n3.getY_pos() < y_min) && (n4.getY_pos() < y_min))
    return false;
if ((n3.getY_pos() > y_max) && (n4.getY_pos() > y_max))
    return false;
if ((n3.getZ_pos() < z_min) && (n4.getZ_pos() < z_min))
    return false;
if ((n3.getZ_pos() > z_max) && (n4.getZ_pos() > z_max))
    return false;

// Skip c_elements directly connected to this element
if (n3.equals(n1) || n3.equals(n2))
    return false;
if (n4.equals(n1) || n4.equals(n2))
    return false;

// The element is crossing this element.
// Calculate a basic vector
v2 = n4.getPos().minus(n3.getPos());

// Calculate the crossing direction
// v3 = v1.vectorProduct(v2,v3);
v3.set(0, 0, (v1.get(1, 0) * v2.get(2, 0)) -
    (v1.get(2, 0) * v2.get(1, 0)));
v3.set(1, 0, (v1.get(2, 0) * v2.get(0, 0)) -
    (v1.get(0, 0) * v2.get(2, 0)));
v3.set(2, 0, (v1.get(0, 0) * v2.get(1, 0)) -
    (v1.get(1, 0) * v2.get(0, 0)));
double area = v3.length();

// Check if parallel
if (area < 1E-15) return false;

// The lines are not parallel. Proceed as usual
// Normalize the v3 vector in length
v3.timesEquals(1 / area);

// A crossing point exists.
// Fill in on the P matrix
P.setMatrix(0, 2, 1, 1, v2.times(-1));
P.setMatrix(0, 2, 2, 2, v3);

// Calculate the values
a_b_distance = P.inverse().times(n3.getPos().minus(n1.getPos()));

// Check if the crossing point is within the lines
if ((a_b_distance.get(0, 0) < 0) || (a_b_distance.get(0, 0) > 1))
    return false;
if ((a_b_distance.get(1, 0) < 0) || (a_b_distance.get(1, 0) > 1))
    return false;

// Check if the distance is too great
if (Math.abs(a_b_distance.get(2, 0)) > radius)
    return false;

// No, this is definitely in contact
```

```
        if (a_b_distance.get(2, 0) < 0)
        {
            a_b_distance.set(2, 0, Math.abs(a_b_distance.get(2, 0)));
            v3.timesEquals(-1);
        }
        return true;
    }


    // In this element, the transformation to local coordinate system is
    // made automatically in the matrix algebra of calculating the element
    // strains (they are  derived in global directions).
    public void updateLocalCoordinateSystem()
    {
    }

    private void calculateLocalVariables()
    {
        // Update the p matrix
        n1 = node[0];
        n2 = node[1];

        // Own vector
        v1 = n2.getPos().minus(n1.getPos());

        // vector length
        l1 = v1.length();

        // The P matrix
        P.setMatrix(0, 2, 0, 0, v1);
    }
}
```

```
public class Solid_Iso_6 extends Element
{
    public Solid_Iso_6()
    {
        material = new Material[8];
        xsi = new double[8];
        etha = new double[8];
        phi = new double[8];
        node = new Node[8];
        W = new double[8];
        H = new Jama.Matrix(6, 9);
        M = new Jama.Matrix(8, 3);
        d = new Jama.Matrix(24, 1);
        f = new Jama.Matrix(24, 1);
        B = new Jama.Matrix[8];
        for (i = 0; i < 8; i++)
            B[i] = new Jama.Matrix(6, 24);
        D = new Jama.Matrix(3, 8);
        J = new Jama.Matrix[8];
        for (i = 0; i < 8; i++)
            J[i] = new Jama.Matrix(3, 3);
        J_inv = new Jama.Matrix(3, 3);
        P = new Jama.Matrix(9, 9);
```

```
    N = new Jama.Matrix(9, 24);
    strain = new Jama.Matrix[8];
    for (i = 0; i < 8; i++)
        strain[i] = new Jama.Matrix(6, 1);
    dstrain = new Jama.Matrix[8];
    for (i = 0; i < 8; i++)
        dstrain[i] = new Jama.Matrix(6, 1);
    stress = new Jama.Matrix[8];
    for (i = 0; i < 8; i++)
        stress[i] = new Jama.Matrix(6, 1);
    number_of_integration_points = 8;
}


// This method calculates the mass matrix of the element. The matrix is
// lumped, i.e. the mass is concentrated to the element nodes. There is
// a consistent way of determining the mass distribution by using the
// HRZ Lumping Scheme: 1. Compute the diagonal coefficients of the
// consistent  mass matrix 2. Compute the total mass of the element m
// 3. Compute a number s by adding the diagonal coefficients Mii
// associated with translational d.o.f (but not rotational d.o.f, if
// any) that are mutually parallel and in the same direction 4. Scale
// all the diagonal coefficients by multiplying them by the ratio m/s,
// thus preserving the total mass of the element. The mass matrix m =
// integral (densityN_transposeNdV) This integral can be translated
// into a gauss point summation.
public void assembleMassMatrix()
{
    // Initialize mass
    mass = new Jama.Matrix(24, 24);
    total_mass = 0;
    s = 0;

    // This element has 8 nodes and with 3 direction each. This gives a
    // 24x24 matrix.  It will be different for each integration point.

    // Define M (the node coordinate matrix)
    for (j = 0; j < 8; j++)
    {
        M.set(j, 0, node[j].getX_pos());
        M.set(j, 1, node[j].getY_pos());
        M.set(j, 2, node[j].getZ_pos());
    }

    // Now, start by computing N, via D
    // Compute D. Use only one integration point. Then xsi=phi=etha=0
    calculateD(0, 0, 0);

    // Create the N matrix by expanding the D matrix
    calculateN();

    // Now, do the matrix calculation to derive the jacobian.
    J[0] = D.times(M);

    // Sum up the mass contribution
    //(The weight function for one gauss point is 2^3 = 8)
    mass = N.transpose().times(N).times(material[0].getDensity())
```

```
            .times(J[0].det()).times(8.0);

    // Keep only the diagonal elements
    for (j = 0; j < 24; j++)
        for (k = 0; k < 24; k++)
            mass.set(j, k, (j == k) ? mass.get(j, k) : 0);

    // Now, we have the 24x24 mass matrix. Move on to step 2, compute
    // total mass of the element.  The total mass is equal to the
    // volume integrand of density. Gauss translation means
    // sum[density*jacobian*weightfunction] over all integration
    // points.

    // Compute D
    calculateD(0, 0, 0);

    // Now, do the matrix calculation to derive the Jacobian.
    J[0] = D.times(M);

    // Sum up the mass contribution (Weight function 2^3 = 8)
    total_mass = material[0].getDensity() * J[0].det() * 8;

    // Third stage is to compute a scaling factor s using u-direction
    // for all nodes
    for (i = 0; i < 8; i++)
        s += mass.get(3 * i, 0);

    // Fourth stage is to scale all diagonal coefficients using
    // total_mass/s as a scaling factor
    for (i = 0; i < 24; i++)
        mass.set(i, i, (mass.get(i, i) * total_mass) / s);

    // We now have a lumped mass matrix stored in the mass matrix. It
    // must be distributed to the nodes.  Use only the u-component.
    // Assume the others are the same. (They should be)
    for (i = 0; i < 8; i++)
        node[i].addMass(mass.get(3 * i, 3 * i) / 8.0);
}


private void calculateD(double xsi, double phi, double etha)
{
    // Calculate the D-matrix for integration point i
    D.set(0, 0, -(1.0 / 8) * (1 - etha) * (1 + phi));
    D.set(0, 1, -(1.0 / 8) * (1 - etha) * (1 - phi));
    D.set(0, 2, -(1.0 / 8) * (1 + etha) * (1 - phi));
    D.set(0, 3, -(1.0 / 8) * (1 + etha) * (1 + phi));
    D.set(0, 4, (1.0 / 8) * (1 - etha) * (1 + phi));
    D.set(0, 5, (1.0 / 8) * (1 - etha) * (1 - phi));
    D.set(0, 6, (1.0 / 8) * (1 + etha) * (1 - phi));
    D.set(0, 7, (1.0 / 8) * (1 + etha) * (1 + phi));
    D.set(1, 0, -(1.0 / 8) * (1 - xsi) * (1 + phi));
    D.set(1, 1, -(1.0 / 8) * (1 - xsi) * (1 - phi));
    D.set(1, 2, (1.0 / 8) * (1 - xsi) * (1 - phi));
    D.set(1, 3, (1.0 / 8) * (1 - xsi) * (1 + phi));
    D.set(1, 4, -(1.0 / 8) * (1 + xsi) * (1 + phi));
    D.set(1, 5, -(1.0 / 8) * (1 + xsi) * (1 - phi));
```

```
        D.set(1, 6, (1.0 / 8) * (1 + xsi) * (1 - phi));
        D.set(1, 7, (1.0 / 8) * (1 + xsi) * (1 + phi));
        D.set(2, 0, (1.0 / 8) * (1 - xsi) * (1 - etha));
        D.set(2, 1, -(1.0 / 8) * (1 - xsi) * (1 - etha));
        D.set(2, 2, -(1.0 / 8) * (1 - xsi) * (1 + etha));
        D.set(2, 3, (1.0 / 8) * (1 - xsi) * (1 + etha));
        D.set(2, 4, (1.0 / 8) * (1 + xsi) * (1 - etha));
        D.set(2, 5, -(1.0 / 8) * (1 + xsi) * (1 - etha));
        D.set(2, 6, -(1.0 / 8) * (1 + xsi) * (1 + etha));
        D.set(2, 7, (1.0 / 8) * (1 + xsi) * (1 + etha));
    }


    private void calculateN()
    {
        // Calculate N-matrix for integration point i by expanding D-matrix
        N.setMatrix(0, 2, 0, 0, D.getMatrix(0, 2, 0, 0));
        N.setMatrix(3, 5, 1, 1, D.getMatrix(0, 2, 0, 0));
        N.setMatrix(6, 8, 2, 2, D.getMatrix(0, 2, 0, 0));
        N.setMatrix(0, 2, 3, 3, D.getMatrix(0, 2, 1, 1));
        N.setMatrix(3, 5, 4, 4, D.getMatrix(0, 2, 1, 1));
        N.setMatrix(6, 8, 5, 5, D.getMatrix(0, 2, 1, 1));
        N.setMatrix(0, 2, 6, 6, D.getMatrix(0, 2, 2, 2));
        N.setMatrix(3, 5, 7, 7, D.getMatrix(0, 2, 2, 2));
        N.setMatrix(6, 8, 8, 8, D.getMatrix(0, 2, 2, 2));
        N.setMatrix(0, 2, 9, 9, D.getMatrix(0, 2, 3, 3));
        N.setMatrix(3, 5, 10, 10, D.getMatrix(0, 2, 3, 3));
        N.setMatrix(6, 8, 11, 11, D.getMatrix(0, 2, 3, 3));
        N.setMatrix(0, 2, 12, 12, D.getMatrix(0, 2, 4, 4));
        N.setMatrix(3, 5, 13, 13, D.getMatrix(0, 2, 4, 4));
        N.setMatrix(6, 8, 14, 14, D.getMatrix(0, 2, 4, 4));
        N.setMatrix(0, 2, 15, 15, D.getMatrix(0, 2, 5, 5));
        N.setMatrix(3, 5, 16, 16, D.getMatrix(0, 2, 5, 5));
        N.setMatrix(6, 8, 17, 17, D.getMatrix(0, 2, 5, 5));
        N.setMatrix(0, 2, 18, 18, D.getMatrix(0, 2, 6, 6));
        N.setMatrix(3, 5, 19, 19, D.getMatrix(0, 2, 6, 6));
        N.setMatrix(6, 8, 20, 20, D.getMatrix(0, 2, 6, 6));
        N.setMatrix(0, 2, 21, 21, D.getMatrix(0, 2, 7, 7));
        N.setMatrix(3, 5, 22, 22, D.getMatrix(0, 2, 7, 7));
        N.setMatrix(6, 8, 23, 23, D.getMatrix(0, 2, 7, 7));
    }


    // This method calculates the nodal forces resulting from the stresses
    // in the element. The forces are all added up in a force matrix f with
    // the format: f = [ fx1 fy1 fz1 fx2 fy2 ..... fz8 ]
    public void calculateNodalForces(int i)
    {


        // The nodal force array is the sum of all integration point
        // contributions times the gauss weight factor. In this particular
        // element, where the order of gauss evaluation is 2, the weight
        // factor is 1 and the multiplication is therefore not done.
        // Otherwise, the expression would have been:
        // f = B.transpose().times(stress[i]).times(J.det()).
        // times(weightfactor);
```

```
    // Start by calculating the force vector contribution
    f = B[i].transpose().times(stress[i]).times(J[i].det()).times(
            W[i] * W[i] * W[i]);

    // Now, all the node contributions are represented in this matrix.
    // It needs to be split up. Each node contribution must then be
    // added to each node
    for (n = 0; n < 8; n++)
    {
        // Split up; extract a 3x1 matrix from the large matrix.
        global_force = [ fxN fyN fzN ]
        global_force = f.getMatrix(3 * n, (3 * n) + 2, 0, 0);

        // Add to node (Internal forces are subtracted)
        node[n].addInternalForce(global_force.times(-1.0));
    }
}


// Definition of strain vector is [exx eyy ezz gammaxy gammayz gammxz]
// The strain is derived as strain = B  d where d is an array of node
// displacements d = [u1 v1 w1 u2 v2 w2 .... w8]. The B-matrix is
// derived from three main matrices, H, J and D according to: strain =
// HPN where P is the expanded version of the inverted Jacobian J, N is
// an expanded version of D.  J is called the Jacobian. It consists of
// an array M which includes the node coordinates and the D matrix
// again as: J = DM.  H is a coupling matrix. H is defined in the
// setInitialConditions method since it is not dependent on any
// variable. Note, this method is run several times; i is
// the number of the integration point run.
public void calculateStrain(double tstep, int i)
{
    // Calculate the D-matrix
    calculateD(xsi[i], phi[i], etha[i]);

    // Create the N matrix by expanding the D matrix
    calculateN();

    // Update M (the node coordinate matrix)
    for (j = 0; j < 8; j++)
    {
        M.set(j, 0, node[j].getX_pos());
        M.set(j, 1, node[j].getY_pos());
        M.set(j, 2, node[j].getZ_pos());
    }

    // Determine the displacement matrix (difference between old,
    // one timestep ago and new node positions)
    for (j = 0; j < 8; j++)
    {
        d.set(3 * j, 0, node[j].getX_dpos());
        d.set((3 * j) + 1, 0, node[j].getY_dpos());
        d.set((3 * j) + 2, 0, node[j].getZ_dpos());
    }

    // Now, do the matrix calculation to derive the jacobian.
```

```
        J[i] = D.times(M);

        // Now, create the P matrix by first inverting it..
        J_inv = J[i].inverse();

        // ... and then expanding it
        P.setMatrix(0, 2, 0, 2, J_inv);
        P.setMatrix(3, 5, 3, 5, J_inv);
        P.setMatrix(6, 8, 6, 8, J_inv);

        // Now, calculate the B matrix
        B[i] = H.times(P.times(N));

        // Calculate the strain increment array for this integration point
        dstrain[i] = B[i].times(d);
    }


    // This method calculates the stresses in each integration point based
    // on old strain, strain increment and old stress. The material law
    // will update the strain and stress matrices to new values using
    // constitutive law and strain increment dstrain. The stresses are:
    // [Sxx Syy Szz Sxy Syz Sxz] transposed (or [sigmaxx sigmayy sigmazz
    // tauxy tauyz tauxz] transposed). Note tauxz = tauzx and so on.
    public void calculateStress(int i, double timestep)
    {
        // i is the current integration point.
        material[i].calculateStressThreeDimensional(
            strain[i], dstrain[i], stress[i], timestep);
    }


    public void setInitialConditions()
    {
        double t = 1.0 / Math.sqrt(3);
        for (i = 0; i < number_of_integration_points; i++)
            material[i] = (Material) material[0].clone();
        for (i = 0; i < number_of_integration_points; i++)
            material[i].setInitialConditions();
        xsi[0] = -t; xsi[1] = -t; xsi[2] = -t; xsi[3] = -t;
        xsi[4] = t; xsi[5] = t; xsi[6] = t; xsi[7] = t;

        etha[0] = -t; etha[1] = -t; etha[2] = t; etha[3] = t;
        etha[4] = -t; etha[5] = -t; etha[6] = t; etha[7] = t;

        phi[0] = t; phi[1] = -t; phi[2] = -t; phi[3] = t;
        phi[4] = t; phi[5] = -t; phi[6] = -t; phi[7] = t;

        W[0] = 1.0; W[1] = 1.0; W[2] = 1.0; W[3] = 1.0;
        W[4] = 1.0; W[5] = 1.0; W[6] = 1.0; W[7] = 1.0;

        H.set(0, 0, 1.0); H.set(1, 4, 1.0);
        H.set(2, 8, 1.0);
        H.set(3, 1, 1.0); H.set(3, 3, 1.0);
        H.set(4, 5, 1.0); H.set(4, 7, 1.0);
        H.set(5, 2, 1.0); H.set(5, 6, 1.0);
```

```
        for (i = 0; i < 9; i++)
            for (j = 0; j < 24; j++)
            {
                N.set(i, j, 0.0);
                P.set(i, j, 0.0);
            }
    }
}
```

```
public class Elastoplastic extends Material
{
    public Elastoplastic()
    {
        trial_stress = new Jama.Matrix(6, 1);
        stiffness_matrix_3d = new Jama.Matrix(6, 6);
        stiffness_matrix_plane_stress = new Jama.Matrix(6, 6);
        A = new Jama.Matrix(6, 6);
        I = new Jama.Matrix(6, 6);
        eps = 0;
        eps_vel = 0;
    }

    // This method updates the stress and strain given the strain, strain
    // increase and stress. It is quite simple since it is one-dimensional.
    public void calculateStressOneDimensional(Jama.Matrix strain,
              Jama.Matrix dstrain, Jama.Matrix stress, double timestep)
    {
        // Update the strain
        strain.plusEquals(dstrain);

        // Calculate trail stress (let that be default)
        stress.set(
            0, 0, stress.get(0, 0) + (youngs_modulus * dstrain.get(0, 0)));

        // Calculate yield function
        double yield_function = Math.abs(stress.get(0, 0)) -
            this.yieldStress(eps, eps_vel);

        // Check if above yield stress
        if (yield_function > 0)
        {
            // Update effective plastic strain
            eps_vel = Math.abs(dstrain.get(0, 0) / timestep);
            eps += Math.abs(dstrain.get(0, 0));

            // Update the stress. The effective strain is the same as
            // epsilonxx since it is one-dimensional.
            stress.set(0, 0, yieldStress(eps, eps_vel));
        }
    }


    // This method calculates a new stress based on a given strain, strain
    // increase and old stress matrix for an element. Note that the strain
    // is not corrected! This method works only if the material object has
    // a "memory" since values from the previous timestep of the effective
```

```
// plastic strain is used in the function. The previous timestep means
// the previous time that this method was called.
public void calculateStressThreeDimensional(Jama.Matrix strain,
         Jama.Matrix dstrain, Jama.Matrix stress, double timestep)
{
    // Determine the stress increase elastically to get the trial
    // stress and put it back in the stress matrix
    stress.plusEquals(stiffness_matrix_3d.times(dstrain));

    // Calculate the deviatoric stresses (note the sign of pressure!)
    double pressure = -(stress.get(0, 0) + stress.get(1, 0) +
                        stress.get(2, 0)) / 3.0;
    stress.set(0, 0, stress.get(0, 0) + pressure);
    stress.set(1, 0, stress.get(1, 0) + pressure);
    stress.set(2, 0, stress.get(2, 0) + pressure);

    // Determine yield function
    double vm_stress = Math.sqrt(
            ((3.0 / 2.0) * ((stress.get(0, 0) * stress.get(0, 0)) +
            (stress.get(1, 0) * stress.get(1, 0)) +
            (stress.get(2, 0) * stress.get(2, 0)))) +
            (3.0 * ((stress.get(3, 0) * stress.get(3, 0)) +
            (stress.get(4, 0) * stress.get(4, 0)) +
            (stress.get(5, 0) * stress.get(5, 0)))));

    // Now check if the new stress estimation is above the yield
    // surface and needs correction
    if (vm_stress > yieldStress(eps, eps_vel))
    {
        // Compute plastic strain increment and increment eps
        eps_vel = ((vm_stress - yieldStress(eps, eps_vel)) / (((3.0 *
                    youngs_modulus) / (2.0 * (1.0 + nu))) +
                    yieldStressDerivate(eps, eps_vel))) / timestep;
        eps += (eps_vel * timestep);

        // Scale back the estimated trial stress to the yield surface
        stress.timesEquals(yieldStress(eps, eps_vel) / vm_stress);
    }

    // Convert stress back from deviatoric format.
    stress.set(0, 0, stress.get(0, 0) - pressure);
    stress.set(1, 0, stress.get(1, 0) - pressure);
    stress.set(2, 0, stress.get(2, 0) - pressure);
}


// This method calculates a stress based on a given strain, strain
// increase, and stress for an element. This is the default input for a
// material law.  All matrices are 6x1: strain = [epsilonx, epsilony,
// epsilonz, gammaxy, gammayz, gammazx] transposed; dstrain = same
// format as strain but latest increment; stress = [sigmax, sigmay,
// sigmaz, tauxy, tauyz, tauzx] transposed
public void calculateStressTwoDimensionalPlaneStress(Jama.Matrix
    strain, Jama.Matrix dstrain, Jama.Matrix stress, double timestep)
{

    // Determine the stress increase elastically to get the trial
```

```
        // stress and put it back in the stress matrix
        stress.plusEquals(stiffness_matrix_plane_stress.times(dstrain));

        // Determine yield function
        double vm_stress = Math.sqrt(
                ((stress.get(0, 0) * stress.get(0, 0)) +
                (stress.get(1, 0) * stress.get(1, 0))) -
                (stress.get(0, 0) * stress.get(1, 0)) +
                (3.0 * (stress.get(3, 0) * stress.get(3, 0))));

        // Now check if the new stress estimation is above the yield
        // surface and needs correction
        if (vm_stress > yieldStress(eps, eps_vel))
        {
            // Compute plastic strain increment and increment eps
            eps_vel = (vm_stress - yieldStress(eps, eps_vel)) /
                      (youngs_modulus * timestep);
            eps += (eps_vel * timestep);

            // Scale back the estemated trial stress to the yield surface
            stress.timesEquals(yieldStress(eps, eps_vel) / vm_stress);

            // The strain increment in thickness direction is in this case
            // just to keep the volume constant.
            dstrain.set(2, 0, -(dstrain.get(0, 0) + dstrain.get(1, 0)));
        } else
        {
            // If elastic, update the strain increment in thickness
            // direction elastically
            dstrain.set(2, 0,
                -(nu / (1.0-nu)) * (dstrain.get(0,0) + dstrain.get(1,0)));
        }

        // Finally, update the strain
        strain.plusEquals(dstrain);
    }


    public void setInitialConditions()
    {
        // Initialize all the stiffness matrices
        c = youngs_modulus / ((1.0 + nu) * (1.0 - (2.0 * nu)));
        c2 = youngs_modulus / (1.0 - (nu * nu));
        G = youngs_modulus / (2.0 * (1.0 + nu));

        stiffness_matrix_3d.set(0, 0, (1.0 - nu) * c);
        stiffness_matrix_3d.set(0, 1, nu * c);
        stiffness_matrix_3d.set(0, 2, nu * c);
        stiffness_matrix_3d.set(1, 0, nu * c);
        stiffness_matrix_3d.set(1, 1, (1.0 - nu) * c);
        stiffness_matrix_3d.set(1, 2, nu * c);
        stiffness_matrix_3d.set(2, 0, nu * c);
        stiffness_matrix_3d.set(2, 1, nu * c);
        stiffness_matrix_3d.set(2, 2, (1.0 - nu) * c);
        stiffness_matrix_3d.set(3, 3, G / 2.0);
        stiffness_matrix_3d.set(4, 4, G / 2.0);
        stiffness_matrix_3d.set(5, 5, G / 2.0);
```

```
        stiffness_matrix_plane_stress.set(0, 0, c2);
        stiffness_matrix_plane_stress.set(0, 1, nu * c2);
        stiffness_matrix_plane_stress.set(1, 0, nu * c2);
        stiffness_matrix_plane_stress.set(1, 1, c2);
        stiffness_matrix_plane_stress.set(3, 3, ((1.0 - nu) * c2) / 2.0);
        stiffness_matrix_plane_stress.set(4,4,((5.0/6.0)*(1.0-nu)*c2)/2.0);
        stiffness_matrix_plane_stress.set(4, 5, 0);
        stiffness_matrix_plane_stress.set(5,5,((5.0/6.0)*(1.0-nu)*c2)/2.0);

        // A matrix for material law plane stress
        A.set(0, 0, 1.0); A.set(0, 1, -0.5); A.set(1, 0, -0.5);
        A.set(1, 1, 1.0); A.set(3, 3, 3.0);

        // I matrix (Unity matrix) for material law plane stress
        I.set(0, 0, 1.0); I.set(1, 1, 1.0); I.set(2, 2, 1.0);
        I.set(3, 3, 1.0); I.set(4, 4, 1.0); I.set(5, 5, 1.0);
    }


    // This function returns the yield stress as a function of effective
    // plastic strain
    private double yieldStress(double plastic_strain, double strain_vel)
    {
        if (yield_stress.isAConstant())
            return yield_stress.value(plastic_strain) +
                                     (factor * plastic_strain);
        else if (V_is_set == 0)
            return yield_stress.value(plastic_strain);
        else
            return yield_stress.value(plastic_strain, strain_vel);
    }


    // This function returns the yield stress derivate (increase) as a
    // function of effective plastic strain
    private double yieldStressDerivate(
        double plastic_strain, double strain_vel)
    {
        if (yield_stress.isAConstant())
            return factor;
        else if (V_is_set == 0)
            return yield_stress.derivate(plastic_strain);
        else
            return yield_stress.derivate(plastic_strain, strain_vel);
    }
}
```

# Post-processing

The output of the processor is a file with the displacements of the nodes and stresses and strains of the elements, printed at the specified time intervals. The file can be opened by a post-processor program, such as GiD (in post-processing mode), and the displacements at each time interval can be used to play an animation, and the stresses and strains can be used for further analysis in the post-processor.

```
# At time t = 0.2 . . .
  DISPLACEMENTS 1 0.20009999999999428 2 1 0
1 1.2592840658015651E-5   1.9043052914784243E-7   2.4947241845756814E-6
2 1.2638625020322536E-5   1.5324262666812603E-6   1.017617241719958E-6
. . .
1133 -1.959532980322365E-9   1.41835556632941E-8   3.028223005685504E-9
1134 -2.3442083829650073E-9  1.9582074628488044E-8   1.0881748835345206E-9

LOCAL_STRESSES_ 1 0.10000000000000184 1 2 0 "TypeROD_2"
64 -4.330071339287944E-7
65 5.314430555561958E-7
66 3.7022872921233824E-5
67 -3.343666309569047E-5
68 -2.870428766342823E-5

GLOBAL_STRESSES 1 0.10000000000000184 3 2 0 "TypeSOLID_ISO_6"
1 -2.4176895192112814E-7   -4.2271912491748585E-8  -5.5125793812794384E-8
   4.4443204814866476E-8 -5.4164066610042294E-9  -9.697799375772764E-9
  -2.140599633329062E-7 -4.136068109394219E-8 3.63262844977758E-8
   7.696287734099122E-8  -1.0050725729525792E-9  -4.5668844888623324E-8
  -5.0727231150341134E-8   -1.3986490719532762E-7  5.5774836028136875E-8
   3.827944448435511E-8  8.195379047911982E-9  -6.160461724573802E-8
  -9.565843513446389E-9 1.9921403022428013E-8 3.3193133169731523E-8
   2.1303222051364234E-8 4.206922351679053E-9  -2.563357160693879E-8
  -4.3001227713393064E-8   2.6417121714809953E-8 -2.4930077130699904E-8
   3.7743765376204666E-8 1.1149963375186701E-9 -1.8683540547928262E-8
  -1.402360784795583E-8 2.8596984599602758E-8 6.948214187559012E-8
   7.026343791472033E-8  2.8483122041040583E-8 -3.91111360014616E-8
  4.0504973134570254E-8 -1.1653759217289735E-7  4.2300342593312676E-8
   8.623213739320362E-9  3.768357380130798E-8  -5.546978569679793E-8
  8.039772948738738E-8  4.198008650402279E-8  1.675849899715215E-8  -
  8.353008694602236E-9  1.0738325495917516E-8 -3.5042190112798165E-8
2 -1.096974546238785E-6 -3.2573118792433853E-7  -2.7041137659588864E-7
   1.752085601813217E-7  5.492591039713616E-9  -4.280301950331745E-8
  -9.418887920735715E-7 8.569620802266172E-9  -8.775963156467824E-8
   3.1391220553741746E-7 1.883319751701023E-8  -1.951271790155249E-7
  -3.3910270883012264E-8   9.619737207419715E-8  2.109222517280884E-7
   1.3492189735234164E-7 7.29791016058686E-8 -2.0201867575907375E-7
  -1.5633360068654512E-7   -1.6189110437238627E-7  6.093293326846986E-8
   7.558008126037332E-8  2.6622830456659462E-8 -4.9694516514701033E-8
  -1.8943975254719565E-7   2.698902379997825E-8  -2.21212460240608E-9
   2.2001232296151297E-7 4.7368648551852407E-8 -1.8785599686697298E-8
  -1.334009888679036E-7 2.6224283851380454E-7 -5.067003268832237E-8
   3.5871596814375977E-7 7.159673179969817E-8  -9.17479296363091E-8
  2.1904472322003628E-7 1.1178509912716754E-7 9.92636043512211E-9
   1.6883818289531202E-7 1.2574263603074766E-7 -6.562376091664775E-8
  1.9566838328981313E-7 -4.725638403828106E-8 9.104669475029564E-8
   1.094963668552715E-7  6.849888811909185E-8  7.338568924344147E-9
. . .

LOCAL_STRAINS__ 1 0.20009999999999428 1 2 0 "TypeROD_2"
64 5.4611606199110215E-9
65 -2.255341416854519E-8
66 -9.732184931899127E-8
67 -1.8948021515048487E-7
68 -3.891275819072443E-6
```

```
GLOBAL_STRAINS_ 1 0.20009999999999428 3 2 0 "TypeSOLID_ISO_6"
1 0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
2 0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0
  0.0  0.0  0.0  0.0  0.0  0.0

. . .


# At time t = 0.3 . . .
  DISPLACEMENTS 1 0.30009999999998327 2 1 0
1  4.296516066674646E-5    3.868095184076604E-6    4.471519025938642E-6
2  4.3095470796394864E-5     8.184828400245297E-6    1.7864634771194375E-6

. . .
```