```
#ifndef SOLVER_CLASS_H
#define SOLVER_CLASS_H
#include "Definitions.h"
#include "Utilities.h"
#include "Assembler.h"
#include <Eigen/LU>

template <class Assembler0,class Assembler1, class Assembler2>
class SolverImplicitDynamics {

  public:

  typedef typename Assembler0::ElementVector ElementVector;

  // "Special-Case"-Constructor
  SolverImplicitDynamics(Assembler0 & assembler0,
                         const double timestep    ,
                         const double dampingAlpha,
                         const double dampingBeta ,
                         const double newmarkBeta ,
                         const double newmarkGamma):
      _assembler0(assembler0  ),
      _assembler1(Assembler1(assembler0.getNumberOfNodes())),
      _assembler2(Assembler2(assembler0.getNumberOfNodes())),
      _dampingAlpha(dampingAlpha),
      _dampingBeta (dampingBeta ),
      _newmarkBeta (newmarkBeta ),
      _newmarkGamma(newmarkGamma){
  }

  // "Special-Case"-Constructor
  SolverImplicitDynamics(Assembler0 & assembler0,
                         Assembler1 & assembler1,
                         const double timestep    ,
                         const double dampingAlpha,
                         const double dampingBeta ,
                         const double newmarkBeta ,
                         const double newmarkGamma):
      _assembler0(assembler0),
      _assembler1(assembler1),
      _assembler2(Assembler2(assembler0.getNumberOfNodes())),
      _timestep    (timestep    ),
      _dampingAlpha(dampingAlpha),
      _dampingBeta (dampingBeta ),
      _newmarkBeta (newmarkBeta ),
      _newmarkGamma(newmarkGamma){
  }

  // "General-Case"-Constructor
  SolverImplicitDynamics(Assembler0 & assembler0,
                         Assembler1 & assembler1,
                         Assembler2 & assembler2,
                         const double timestep    ,
                         const double dampingAlpha,
                         const double dampingBeta ,
                         const double newmarkBeta ,
                         const double newmarkGamma):
      _assembler0(assembler0),
      _assembler1(assembler1),
      _assembler2(assembler2),
      _timestep    (timestep    ),
      _dampingAlpha(dampingAlpha),
      _dampingBeta (dampingBeta ),
      _newmarkBeta (newmarkBeta ),
      _newmarkGamma(newmarkGamma){
  }



  void
  computeNewmarkUpdate(const vector<EssentialBoundaryCondition> & essentialBCs,
                       vector<ElementVector> &       currentNodalDisplacement ,
                       vector<ElementVector> &       currentNodalVelocity     ,
```

```
                        vector<ElementVector> &       currentNodalAcceleration ,
                        const unsigned int            maxIterations = 1000,
                        const double                  tolerance = 1e-4    ,
                        const bool                    verbose = true      ) {

    // Solving for the updated displacements using Newton-Raphson iterations
    if (verbose) {
      printf("Implicit Dynamics solver trying to achieve a tolerance of %e in %u "
             "maximum iterations\n", tolerance, maxIterations);
    }

    // Some parameters
    size_t DegreesOfFreedom = Assembler0::DegreesOfFreedom;
    size_t numberOfDOFs     = currentNodalDisplacement.size()*DegreesOfFreedom;

    // TODO: create three VectorXd's currentDisplacement, currentVelocity and
    currentAcceleration
    //       these should remain unchanged so you can even define them as "const"

    VectorXd currentDisplacement(numberOfDOFs);
    for (unsigned int nodeIndex = 0; nodeIndex < currentNodalDisplacement.size();
    nodeIndex++) {
      for (unsigned int dofIndex = 0; dofIndex < DegreesOfFreedom; dofIndex++) {
        currentDisplacement(nodeIndex * DegreesOfFreedom + dofIndex) =
        currentNodalDisplacement[nodeIndex](dofIndex); // ...
      }
    }
    VectorXd currentVelocity(numberOfDOFs);
    for (unsigned int nodeIndex = 0; nodeIndex < currentNodalDisplacement.size();
    nodeIndex++) {
      for (unsigned int dofIndex = 0; dofIndex < DegreesOfFreedom; dofIndex++) {
        currentVelocity(nodeIndex * DegreesOfFreedom + dofIndex) =
        currentNodalVelocity[nodeIndex](dofIndex); // ...
      }
    }
    VectorXd currentAcceleration(numberOfDOFs);
    for (unsigned int nodeIndex = 0; nodeIndex < currentNodalDisplacement.size();
    nodeIndex++) {
      for (unsigned int dofIndex = 0; dofIndex < DegreesOfFreedom; dofIndex++) {
        currentAcceleration(nodeIndex * DegreesOfFreedom + dofIndex) =
        currentNodalAcceleration[nodeIndex](dofIndex); // ...
      }
    }
    // ...

    // TODO: further define a VectorXd newDisplacement, which is the solution, we will
    //       iteratively (try to) improve

    VectorXd newDisplacement(numberOfDOFs);
    newDisplacement.fill(0);

    // TODO: Boundary conditions I - Solution
    for (size_t bcIndex = 0; bcIndex < essentialBCs.size(); ++bcIndex) {

      const EssentialBoundaryCondition & bc = essentialBCs[bcIndex];
      const size_t dofIndex = bc._nodeId * DegreesOfFreedom + bc._coordinate;
      currentDisplacement(dofIndex)    = bc._constraint;

    }

    // TODO: newDisplacement in vector<ElementVector> form - needed in this form to
    evaluate
    //       stiffness, forces, etc. etc.
    vector<ElementVector> nodalNewDisplacements
            =
            Utilities::distributeGlobalVectorToLocalVectors<Assembler0>(currentDisplac
            ement);

    // TODO: Evaluate the consistent mass matrix and damping matrix
    Eigen::MatrixXd consistentMassMatrix(numberOfDofs, numberOfDofs);
    consistentMassMatrix = _assembler0.assembleConsistentMassMatrix() +
    _assembler1.assembleConsistentMassMatrix() +
    _assembler2.assembleConsistentMassMatrix();
```

```
135       Eigen::MatrixXd stiffnessMatrix(numberOfDofs, numberOfDofs);
136       stiffnessMatrix = _assembler0.assembleStiffnessMatrix(nodalNewDisplacements) +
          _assembler1.assembleStiffnessMatrix(nodalNewDisplacements) +
          _assembler2.assembleStiffnessMatrix(nodalNewDisplacements);
137       Eigen::MatrixXd dampingMatrix(numberOfDofs, numberOfDofs);
138       dampingMatrix = dampingAlpha * consistentMassMatrix + dampingBeta *
          stiffnessMatrix;
139
140       //TODO: evaluate the effective force vector
141       VectorXd effectiveForceVector;
142       VectorXd globalForceVector;// ...
143       VectorXd rVector;
144       globalForceVector = _assembler0.assembleForceVector(nodalNewDisplacements) +
          _assembler1.assembleForceVector(nodalNewDisplacements) +
          _assembler2.assembleForceVector(nodalNewDisplacements);
145       rVector = consistentMassMatrix.solve((1/(_newmarkBeta*_timestep*_timestep) *
          currentNodalDisplacement) + 1/(_newmarkBeta*_timestep) * currentNodalVelocity +
          (1/(2*_newmarkBeta) -1)*currentNodalAcceleration)
146                 +
                    dampingMatrix.solve(_newmarkGamma/(_newmarkBeta*_timestep)*currentNodalD
                    isplacement + (_newmarkGamma/_newmarkBeta -1)*currentNodalVelocity +
                    _timestep*(_newmarkGamma/(2*_newmarkBeta)-1)*currentNodalAcceleration);
147       effectiveForceVector =
          ((1/(_newmarkBeta*_timestep*_timestep)*consistentMassMatrix)+((_newmarkGamma/(_new
          markBeta*_timestep))*dampingMatrix)).slove(nodalNewDisplacements)
148                           + globalForceVector
149                           - rVector;
150
151       // TODO: Boundary conditions II - Force
152       for (size_t bcIndex = 0; bcIndex < essentialBCs.size(); ++bcIndex) {
153
154         // ...
155         const EssentialBoundaryCondition & bc = essentialBCs[bcIndex];
156         effectiveForceVector(bc._nodeId * DegreesOfFreedom + bc._coordinate) = 0.0;
157
158       }
159
160       // TODO: Evaluate the residual based on the effectiveForceVector incl. BCs
161       double residue = 0.0;
162       residue = effectiveForceVector.norm();
163       if (verbose == true){
164         printf("Initial residue = %9.3e\n", residue);
165       }
166
167       // While the residue > tolerance compute Newton-Raphson iterations
168       unsigned int numberOfIterations = 0;
169
170       MatrixXd effectiveTangentMatrix;
171
172       while( (residue > tolerance) && (numberOfIterations < maxIterations) ) {
173
174         // TODO: Set the efficient tangentMatrix
175         // ...
176         effectiveTangentMatrix = stiffnessMatrix +
          (consistentMassMatrix/(_newmarkBeta*_timestep*_timestep) + _newmarkGamma *
          dampingMatrix/(_newmarkBeta*_timestep));
177         // Boundary condition III - Effective Tangent Matrix
178         for (size_t bcIndex = 0; bcIndex < essentialBCs.size(); ++bcIndex) {
179
180           // ...
181           const EssentialBoundaryCondition & bc = essentialBCs[bcIndex];
182           effectiveTangentMatrix.row(bc._nodeId * DegreesOfFreedom +
          bc._coordinate).fill(0.0);
183           effectiveTangentMatrix(bc._nodeId * DegreesOfFreedom + bc._coordinate,
          bc._nodeId * DegreesOfFreedom + bc._coordinate) = 1.0;
184         }
185
186         // Update newDisplacement using the Newmark method update rule
187         newDisplacement -= effectiveTangentMatrix.lu().solve(effectiveForceVector);
188
189         // TODO :Boundary conditions IV - Solution
190         for (size_t bcIndex = 0; bcIndex < essentialBCs.size(); ++bcIndex) {
191
```

```
192           // ...
193           const EssentialBoundaryCondition & bc = essentialBCs[bcIndex];
194           const size_t dofIndex = bc._nodeId * DegreesOfFreedom + bc._coordinate;
195           newDisplacement(dofIndex)    = bc._constraint;
196         }
197
198         // TODO: again convert newDisplacement into nodal form (i.e. update
            nodalNewDisplacements)
199
200         // ...
201         vector<ElementVector> nodalNewDisplacements
202                 =
                  Utilities::distributeGlobalVectorToLocalVectors<Assembler0>(newDisplacem
                  ent);
203
204         // TODO: Evaluate the new damping matrix
205         // ...
206         Eigen::MatrixXd consistentMassMatrix(numberOfDofs, numberOfDofs);
207         consistentMassMatrix = _assembler0.assembleConsistentMassMatrix() +
            _assembler1.assembleConsistentMassMatrix() +
            _assembler2.assembleConsistentMassMatrix();
208         Eigen::MatrixXd stiffnessMatrix(numberOfDofs, numberOfDofs);
209         stiffnessMatrix = _assembler0.assembleStiffnessMatrix(nodalNewDisplacements) +
            _assembler1.assembleStiffnessMatrix(nodalNewDisplacements) +
            _assembler2.assembleStiffnessMatrix(nodalNewDisplacements);
210         Eigen::MatrixXd dampingMatrix(numberOfDofs, numberOfDofs);
211         dampingMatrix = dampingAlpha * consistentMassMatrix + dampingBeta *
            stiffnessMatrix;
212
213         // TODO: evaluate the new effective force vector
214         // ...
215         VectorXd effectiveForceVector; // ...
216         VectorXd globalForceVector;
217         globalForceVector = _assembler0.assembleForceVector(nodalNewDisplacements) +
            _assembler1.assembleForceVector(nodalNewDisplacements) +
            _assembler2.assembleForceVector(nodalNewDisplacements);
218         effectiveForceVector =
            ((1/(_newmarkBeta*_timestep*_timestep)*consistentMassMatrix)+((_newmarkGamma/(_n
            ewmarkBeta*_timestep))*dampingMatrix)).slove(nodalNewDisplacements)
219                               + globalForceVector
220                               - rVector;
221
222
223         // TODO: Boundary conditions V - Force
224         for (size_t bcIndex = 0; bcIndex < essentialBCs.size(); ++bcIndex) {
225
226           // ...
227           const EssentialBoundaryCondition & bc = essentialBCs[bcIndex];
228           effectiveForceVector(bc._nodeId * DegreesOfFreedom + bc._coordinate) = 0.0;
229
230         }
231
232         // TODO: evaluate the residual based on the norm of effectiveForceVector and
            divide it by the numberOfDOFs
233         residue = effectiveForceVector.norm()/numberOfDOFs; // ...
234
235         if (verbose == true) {
236           printf("Newton Raphson iteration %4u, residue = %8.3e\n",
              numberOfIterations, residue);
237         }
238
239         numberOfIterations++;
240
241       }
242
243       // Error check
244       if (numberOfIterations == maxIterations) {
245         throwException("Newton Raphson solver could not converge "
246                        "in %u iterations.\nTolerance: %e \nResidue: %e",
247                        maxIterations,tolerance,residue);
248       }
249
250       // TODO: Update the states, i.e. save the new displacement, velocity and
```

```
      acceleration onto
251   //       currentNodalDisplacement, currentNodalVelocity and
      currentNodalAcceleration
252
253   // ...
254   tempNodalVelocity = currentNodalVelocity;
255   tempNodalAcceleration = currentNodalAcceleration;
256   tempNodalDisplacement = currentNodalDisplacement;
257   currentNodalAcceleration =
      1/(_newmarkBeta*_timestep*_timestep)(nodalNewDisplacements -
      tempNodalDisplacement - _timestep*tempNodalVelocity)  -
      ((1-2*_newmarkBeta)/(2*_newmarkBeta))*tempNodalAcceleration;
258   currentNodalVelocity = (1 - _newmarkGamma/_newmarkBeta)*tempNodalVelocity +
      (_newmarkGamma/(_newmarkBeta*_timestep))*(nodalNewDisplacements -
      tempNodalDisplacement) - _timestep*(_newmarkGamma/(2*_newmarkBeta)
      -1)*tempNodalAcceleration;
259   currentNodalDisplacement = nodalNewDisplacements;
260
261   // TODO: Delete the following when you're done
262   //ignoreUnusedVariables(numberOfDOFs);
263
264 }
265
266
267
268
269 private:
270
271 Assembler0 _assembler0;
272 Assembler1 _assembler1;
273 Assembler2 _assembler2;
274
275 const double _timestep    ;
276 const double _dampingAlpha;
277 const double _dampingBeta ;
278 const double _newmarkBeta ;
279 const double _newmarkGamma;
280
281
282 };
283
284 #endif  // SOLVER_CLASS_H
285
```