```
#ifndef SIMPLICIAL_ELEMENT
#define SIMPLICIAL_ELEMENT

#include "/src/Definitions.h"
#include "/src/Utilities.h"
#include "/src/FiniteDeformationUtilities.h"

namespace Elements {


template <class MaterialModelType, unsigned int QP, class ElementType, unsigned int DOFs,
          template<class> class NodeType = NodeWithId> // keep ElementType and QP general
class IsoparametricElement {

public:

  struct Properties {

    Properties(const double & elementMultiplier = 1.) :
      _elementMultiplier(elementMultiplier) {
    }

    const double _elementMultiplier;
  };


  static const unsigned int                   QuadPoints = QP;
  static const unsigned int                NumberOfNodes = ElementType::NumberOfNodes;
  static const unsigned int              SpatialDimension = ElementType::SpatialDimension;
  static const VTKCellType                   VtkCellType = ElementType::VtkCellType;
  static const unsigned int      NumberOfNodesPerBoundary = ElementType::NumberOfNodesPerBoundary;
  static const unsigned int            NumberOfBoundaries = ElementType::NumberOfBoundaries;
  static const unsigned int              DegreesOfFreedom = DOFs;

  typedef Eigen::Matrix<double, DegreesOfFreedom, 1>        Vector;
  typedef typename ElementType::Point                       Point;
  typedef NodeType<Point>                                   Node;
  typedef array<Vector, NumberOfNodes>                      NodalDisplacements;
  typedef array<Vector, NumberOfNodes>                      Forces;
  typedef Matrix<double,
                 NumberOfNodes*DegreesOfFreedom,
                 NumberOfNodes*DegreesOfFreedom>            StiffnessMatrix;

  typedef MaterialModelType                                 MaterialModel;
  typedef typename MaterialModel::DisplacementGradient      DisplacementGradient;
  typedef typename MaterialModel::Strain                    Strain;
  typedef typename MaterialModel::Stress                    Stress;
  typedef typename MaterialModel::TangentMatrix             TangentMatrix;
  typedef Matrix<double,DegreesOfFreedom,SpatialDimension>  StressTensor;
  typedef SingleElementBoundary<Node, NumberOfNodesPerBoundary>    SingleBoundary;
  typedef AllElementBoundaries<SingleBoundary, NumberOfBoundaries> AllBoundaries;


  //TODO: create the constructor
  IsoparametricElement(const array<Node, NumberOfNodes> &           nodes         ,
                       const Properties &                           properties    ,
                       const ElementType &                          elementType   ,
                       const QuadratureRule<SpatialDimension, QP> * quadratureRule,
                       const MaterialModel *                        materialModel ) :
      _properties     (properties)    ,
      _quadratureRule (quadratureRule),
      _materialModel  (materialModel) {

    ignoreUnusedVariables(nodes);
    ignoreUnusedVariables(elementType);
    // cout<<"the first quadrature point in the reference configuration"<<
    (*_quadraqureRule)._point[0]
```

```
67        // TODO: We want to copy the information of nodes in terms of ID's and positions
68        //       into _nodeIds and _nodePositions, respectively
69        // NOTE: nodes contains the global id's of all NumberOfNodes nodes of THIS element
70        // NOTE: i.e. NumberOfNodes is not the mesh's total number of nodes, but the
71        //       number of nodes in the element
72        for (unsigned int nodeIndex = 0; nodeIndex < NumberOfNodes; ++nodeIndex) {
73          _nodeIds[nodeIndex]       = 0;
74          _nodeIds[nodeIndex]       = nodes[nodeIndex]._id;
75          _nodePositions[nodeIndex] = Point::Zero();
76          _nodePositions[nodeIndex] = nodes[nodeIndex]._position;
77        }
78
79
80
81        //TODO: The aim is to take the shape functions and shapefunction-derivatives as
82        //      provided by elementType and to appropriately store them inside
83        //      _shapeFunctions and _shapeFunctionDerivatives. Using the jacobian and
84        //      the quadrature weights, we are further going to set _weightedJacobian
85        double volume = 0.;
86        ignoreUnusedVariables(volume);
87
88        for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
89
90          // TODO: Read in the qpth quadrature point from _quadratureRule
91          // NOTE: _quadratureRule is an address to a QuadratureRule, and the class
92          //       QuadratureRule has the public member _points which is an array that
93          //       stores all QPs' position
94          //cout << "HINT: This is the position of the first quadraturePoint: " << endl;
95          //cout << _quadratureRule->_points[0] << endl;
96
97          const Point quadPoint;
98          array <Point, QP> quadPoints;
99          quadPoints[qpIndex] = _quadratureRule->_points[qpIndex];
100
101
102         // Initialization of jacobian, incl. initial zero'ing
103         Matrix<double, SpatialDimension, SpatialDimension> jacobian
104           = Matrix<double, SpatialDimension, SpatialDimension>::Zero();
105
106         // TODO: Read out the shapeFunctionDerivatives provided by elementType for this
107         //       particular quadrature point.
108         // NOTE: The reference element is defined in elementType, an object passed
          to_char_type
109         //       this constructor
110         array<Point, NumberOfNodes> shapeFunctionDerivatives;
111         shapeFunctionDerivatives =
          elementType.computeShapeFunctionDerivatives(quadPoints[qpIndex]);
112
113         // TODO: In a (nested) for-loop, evaluate the entries of the jacobian as
          dervived in
114         //       lecture
115         for (unsigned int nodeId = 0; nodeId < NumberOfNodes; ++nodeId) {
116           for (unsigned int i = 0; i < SpatialDimension; ++i) {
117             for (unsigned int j = 0; j < SpatialDimension; ++j) {
118               jacobian(i, j) += _nodePositions[nodeId](j) *
                  shapeFunctionDerivatives[nodeId](i);
119             }
120           }
121         }
122
123         // TODO: Define the qpIndex'th entry of _shapeFunctions and
          _shapeFunctionDerivatives
124         // HINT: As you can see from the equations given in the assignment sheet,
125         //       there's not really much to do for _shapeFunctions, while there is
126         //       a tad bit more to do for _shapeFunctionDerivatives
127         // HINT: _shapeFunctions and _shapeFunctionDerivatives are arrays containing
          arrays
128         // HINT: someMatrix*someArrayofMatrices is an allowed operation thanks to the
129         //       the std- and Eigen-lib.
130
131         _shapeFunctions[qpIndex] =
          elementType.computeShapeFunctions(quadPoints[qpIndex]);
132
```

```
133         for (unsigned int nodeId = 0; nodeId < NumberOfNodes; ++nodeId)
134         {
135           _shapeFunctionDerivatives[qpIndex][nodeId] = jacobian.inverse() *
              shapeFunctionDerivatives[nodeId];
136         }
137
138         // TODO: Set _weightedJacobian[qpIndex]. This should basically equate to J *
            w_k on
139         //       the HW sheet - w_k are the quadrature weights, which for example for
            the
140         //       constant element (i.e. one quadrature point) would just simply be '1'
141         //cout << "HINT: This is the weight associated to the first quadrature point "
            << endl;
142         //cout << _quadratureRule->_weights[0] << endl;
143
144         _weightedJacobian[qpIndex] = jacobian.determinant() *
            _quadratureRule->_weights[qpIndex] ; // ...
145
146         // TODO: (Accumulatively) update volume, describing the total "weighted
            Jacobian"
147         volume += _weightedJacobian[qpIndex]; // ...
148
149       }
150
151       // TODO: Finally, based on the volume we evaluated in the above for-loop, we now
152       //       evaluate _nodalWeights
153       for (unsigned int nodeIndex = 0; nodeIndex < NumberOfNodes; nodeIndex++) {
154         _nodalWeights[nodeIndex] = 1.11; // ...
155       }
156     }
157
158     //TODO: compute displacement gradient u_{i,j} in Voigt notation at all quadrature
      points
159     array<DisplacementGradient, QP>
160     computeDispGradsAtGaussPoints(const NodalDisplacements & displacements) const {
161
162       // TODO: Initialization of an array displacementGradients of displament
163       //       gradients at all quadrature points
164       array<DisplacementGradient, QP> displacementGradients;
165
166         // TODO: Based on the discrete set of displacements at the nodes and using
167       //       the shape function derivatives, evaluate the displacement gradients
168       //       at the Gauss-points
169       // NOTE: None of the entries of displacementGradients has been zeroed yet...
170       for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
171         for (unsigned int i = 0; i < DegreesOfFreedom; i++) {
172           for (unsigned int j = 0; j < SpatialDimension; j++) {
173               displacementGradients[qpIndex](i*SpatialDimension+j) = 0.0;
174           }
175         }
176       }
177       for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
178         for (unsigned int i = 0; i < DegreesOfFreedom; i++) {
179           for (unsigned int j = 0; j < SpatialDimension; j++) {
180             for (unsigned int nodeId = 0; nodeId < NumberOfNodes; nodeId++) {
181               displacementGradients[qpIndex](i*SpatialDimension+j) +=
                  displacements[nodeId](i) *
                  _shapeFunctionDerivatives[qpIndex][nodeId](j); // ... (a lot to be
                  changed here!)
182             }
183           }
184         }
185       }
186
187       // Return
188       return displacementGradients;
189     }
190
191     //TODO: compute the potential energy of the element by summation over all
      quadrature points
192     double
193     computeEnergy(const NodalDisplacements &  displacements ) const {
194
```

```
195       ignoreUnusedVariables(time);
196       ignoreUnusedVariables(displacements);
197
198       // TODO: Using the computeDispGradsAtGaussPoints function you just defined above,
199       //       obtain the displacement gradients at all quadrature points
200       const array<DisplacementGradient, QP> displacementGradients =
          computeDispGradsAtGaussPoints(displacements); // ...
201
202       // TODO: Sweeping through all quadrature points, evaluate the total energy
203       double energy = 0.;
204       for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
205         energy += _weightedJacobian[qpIndex] *
          _materialModel->computeEnergy(displacementGradients[qpIndex]); // ...
206       }
207
208
209       // Return - Don't forget to multiply by _properties._elementMultiplier
210       return energy * _properties._elementMultiplier;
211     }
212
213     //TODO: compute the nodal forces of the element by summation over all quadrature
          points
214     Forces
215     computeForces(const NodalDisplacements &  displacements ) const {
216
217       ignoreUnusedVariables(time);
218       ignoreUnusedVariables(displacements);
219       const array<DisplacementGradient, QP> displacementGradients =
          computeDispGradsAtGaussPoints(displacements);
220       // Initialization of forcesAtNodes
221       Forces forcesAtNodes;
222       for (size_t nodeId = 0; nodeId < NumberOfNodes; nodeId++){
223           forcesAtNodes[nodeId] = Vector::Zero();
224       }
225
226       // TODO: Based on the displacementGradients at the Q.P., evaluate the forces at
          all nodes
227       // NOTE: None of the entries of forcesAtNodes has been zeroed yet...
228       for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
229         Stress  stress = _materialModel->computeStress(displacementGradients[qpIndex]);
230         Matrix<double,SpatialDimension,SpatialDimension> matrixStress
231         = Utilities::convertTensorFromVoigtToStandard<SpatialDimension>(stress);
232         for (size_t nodeId = 0; nodeId < NumberOfNodes; nodeId++){
233
234           forcesAtNodes[nodeId] +=  matrixStress * _weightedJacobian[qpIndex] *
              _shapeFunctionDerivatives[qpIndex][nodeId];// ...
235         }
236       }
237
238       // Return - don't forget to multiply by _properties._elementMultiplier
239       return forcesAtNodes * _properties._elementMultiplier;
240     }
241
242     //TODO: compute the stiffness matrix of the element by summation over all
          quadrature points
243     StiffnessMatrix
244     computeStiffnessMatrix(const NodalDisplacements & displacements ) const {
245
246       ignoreUnusedVariables(time);
247       ignoreUnusedVariables(displacements);
248
249       // Initialize zero-stiffness matrix
250       Matrix<double, NumberOfNodes*DegreesOfFreedom, NumberOfNodes*DegreesOfFreedom>
          stiffness
251         = Matrix<double, NumberOfNodes*DegreesOfFreedom,
          NumberOfNodes*DegreesOfFreedom>::Zero();
252
253       // TODO: Using the computeDispGradsAtGaussPoints function you just defined above,
254       //       obtain the displacement gradients at all quadrature points
255       const array<DisplacementGradient, QP> displacementGradients =
          computeDispGradsAtGaussPoints(displacements);
256
257       // TODO: Based on the displacementGradients at the Q.P., evaluate the several
```

```
    entries
258 //       of the stiffness matrix by sweeping through all Q.P.s and evaluating
    the local
259 //       incremental stiffness matrix
260 // HINT: The function convertStandardIndicesToVoigtIndex in the Utilities
    namespace might
261 //       proof useful here (you can find it defined in
    FiniteDeformationUtilities.h) in
262 //       the /src/ directory
263
264   for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
265     TangentMatrix tangentMatrix =
      _materialModel->computeTangentMatrix(displacementGradients[qpIndex]);
266
      array<array<array<array<double,SpatialDimension>,SpatialDimension>,SpatialDimens
      ion>,SpatialDimension> tangentMatrixAsArray =
      Utilities::convertFourthOrderTensorFromVoigtToStandard<SpatialDimension>(tangent
      Matrix);
267     for (unsigned int nodeId_a = 0; nodeId_a < NumberOfNodes; nodeId_a++){
268       for (unsigned int nodeId_b = 0; nodeId_b < NumberOfNodes; nodeId_b++){
269         for (unsigned int i = 0; i < DegreesOfFreedom; i++){
270           for (unsigned int n = 0; n < SpatialDimension; n++) {
271             for (unsigned int j = 0; j < DegreesOfFreedom; j++){
272               for (unsigned int l = 0; l < DegreesOfFreedom; l++){
273                 stiffness(nodeId_a * DegreesOfFreedom + i, nodeId_b *
                  SpatialDimension + n) += _weightedJacobian[qpIndex] *
                  tangentMatrixAsArray[i][j][n][l] *
                  _shapeFunctionDerivatives[qpIndex][nodeId_a][j] *
                  _shapeFunctionDerivatives[qpIndex][nodeId_b][l];
274               }
275             }
276           }
277         }
278       }
279     }
280   }
281   //need to be check
282   // Return - don't forget to multiply by _properties._elementMultiplier
283   return stiffness * _properties._elementMultiplier;
284 }
285
286 array<size_t, NumberOfNodes>
287 getNodeIds() const {
288   return _nodeIds;
289 }
290
291 array<Vector, NumberOfNodes>
292 getNodePositions() const {
293   return _nodePositions;
294 }
295
296 array<double, QP>
297 computeWeightsAtGaussPoints() const {
298   return _weightedJacobian;
299 }
300
301 array<double, NumberOfNodes>
302 computeNodalWeights() const {
303   return _nodalWeights;
304 }
305
306 //TODO: compute stresses all quadrature points
307 array<Stress, QP>
308 computeStressesAtGaussPoints(const NodalDisplacements & displacements ,
309                              const double               time          ) const {
310
311   // TODO: Using the computeDispGradsAtGaussPoints function you just defined above,
312   //       obtain the displacement gradients at all quadrature points
313   const array<DisplacementGradient, QP> displacementGradients =
      computeDispGradsAtGaussPoints(displacements);
314
315   // TODO: Sweep through all quadrature points, evaluate the stress tensor and save
316   //       it in stresses
```

```
317       array<Stress, QP> stresses;
318       for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
319         stresses[qpIndex] = Stress::Zero();
320         stresses[qpIndex] = _materialModel->computeStress(displacementGradients);//
          need to be check
321       }
322
323       // Return
324       return stresses;
325     }
326
327     //TODO: compute strains (i.e., linearized strain tensors OR deformation gradients)
      all quadrature points
328     array<Strain, QP>
329     computeStrainsAtGaussPoints(const NodalDisplacements & displacements) const {
330
331       // TODO: Using the computeDispGradsAtGaussPoints function you just defined above,
332       //       obtain the displacement gradients at all quadrature points
333       const array<DisplacementGradient, QP> displacementGradients =
          computeDispGradsAtGaussPoints(displacements);
334
335       // TODO: Sweep through all quadrature points, evaluate the strain tensor and save
336       //       it in strains
337       array<Stress, QP> strains;
338       for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
339         strains[qpIndex] = Strain::Zero();
340         Matrix<double,SpatialDimension,SpatialDimension> standardDisplacementGradient
341                 =
                  Utilities::convertTensorFromVoigtToStandard<SpatialDimension>(displaceme
                  ntGradients[qpIndex]);
342         Matrix<double,SpatialDimension,SpatialDimension> standardStrain   =
          standardDisplacementGradient +
          Matrix<double,SpatialDimension,SpatialDimension>::Identity();
343         strains[qpIndex] =
          Utilities::convertTensorFromStandardToVoigt<SpatialDimension>(standardStrain);
344       }
345
346       // Return
347       return strains;
348     }
349
350
351     private:
352
353     const Properties                                _properties;
354     array<size_t, NumberOfNodes>                    _nodeIds;
355     array<Point, NumberOfNodes>                     _nodePositions;
356     const QuadratureRule<SpatialDimension, QP> *    _quadratureRule; //the weights and
      points, * means address of quadratureRule
357     const MaterialModel *                           _materialModel;
358     array<double, QP>                               _weightedJacobian;
359     array<array<double, NumberOfNodes>, QP>         _shapeFunctions;
360     array<array<Point, NumberOfNodes>, QP>          _shapeFunctionDerivatives;
361     array<double, NumberOfNodes>                    _nodalWeights;// can ignore this
362     AllBoundaries                                   _allBoundaries;
363   };
364
365   }
366
367   #endif //SIMPLICIAL_ELEMENT
368
```