

```

1  #ifndef SIMPLICIAL_ELEMENT
2  #define SIMPLICIAL_ELEMENT
3
4  #include "Definitions.h"
5  #include "Utilities.h"
6  #include "FiniteDeformationUtilities.h"
7
8  namespace Elements {
9
10
11  template <class MaterialModelType, unsigned int QP, class ElementType, unsigned int
DOFs,
12          template<class> class NodeType = NodeWithId>
13  class IsoparametricElement {
14
15  public:
16
17      struct Properties {
18
19          Properties( const double & density          = 1.,
20                    const double & elementMultiplier = 1.) :
21              _density(density),
22              _elementMultiplier(elementMultiplier){}
23
24          const double _density;
25          const double _elementMultiplier;
26
27      };
28
29
30      static const unsigned int          QuadPoints = QP;
31      static const unsigned int          NumberOfNodes =
ElementType::NumberOfNodes;
32      static const unsigned int          SpatialDimension =
ElementType::SpatialDimension;
33      static const VtkCellType           VtkCellType = ElementType::VtkCellType;
34      static const unsigned int          NumberOfNodesPerBoundary =
ElementType::NumberOfNodesPerBoundary;
35      static const unsigned int          NumberOfBoundaries =
ElementType::NumberOfBoundaries;
36      static const unsigned int          DegreesOfFreedom = DOFs;
37
38      typedef Eigen::Matrix<double, DegreesOfFreedom, 1>          Vector;
39      typedef typename ElementType::Point                        Point;
40      typedef NodeType<Point>                                     Node;
41      typedef array<Vector, NumberOfNodes>                       NodalDisplacements;
42      typedef array<Vector, NumberOfNodes>                       Forces;
43      typedef Matrix<double,
44                    NumberOfNodes*DegreesOfFreedom,
45                    NumberOfNodes*DegreesOfFreedom>             StiffnessMatrix;
46      typedef Matrix<double,
47                    NumberOfNodes*DegreesOfFreedom,
48                    NumberOfNodes*DegreesOfFreedom>             MassMatrix;
49
50      typedef MaterialModelType                                     MaterialModel;
51      typedef typename MaterialModel::DisplacementGradient        DisplacementGradient;
52      typedef typename MaterialModel::Strain                      Strain;
53      typedef typename MaterialModel::Stress                     Stress;
54      typedef typename MaterialModel::TangentMatrix              TangentMatrix;
55      typedef Matrix<double, DegreesOfFreedom, SpatialDimension> StressTensor;
56      typedef SingleElementBoundary<Node, NumberOfNodesPerBoundary> SingleBoundary;
57      typedef AllElementBoundaries<SingleBoundary, NumberOfBoundaries> AllBoundaries;
58
59
60      IsoparametricElement(const array<Node, NumberOfNodes> &          nodes
61                          , const Properties &                      properties
62                          , const ElementType &                     elementType
63                          , const QuadratureRule<SpatialDimension, QP> * quadratureRule,
64                          , const MaterialModel *                   materialModel ) :
65          _properties          (properties)
66          , _quadratureRule    (quadratureRule)
67          , _materialModel     (materialModel) {
68

```

```

69     array<Point, NumberOfNodes> nodalPoints;
70
71     // NOTE: nodes contains the global id's of all NumberOfNodes nodes of THIS element
72     // NOTE: i.e. NumberOfNodes is not the mesh's total number of nodes, but the
73     //       number of nodes in the element
74     for (unsigned int nodeIndex = 0; nodeIndex < NumberOfNodes; ++nodeIndex) {
75         _nodeIds[nodeIndex] = nodes[nodeIndex]._id;
76         _nodePositions[nodeIndex] = nodes[nodeIndex]._position;
77     }
78
79
80     _volume = 0.;
81
82     // Quadrature Points : go through all of them and pre-evaluate
83     //                     a) shapeFunctions
84     //                     b) shapeFunctionDerivatives
85     //                     c) weighted Jacobian (quadrature weight x J-determinant)
86     //                     Why? Well they are defined with respect to the reference
87     //                     configuration, which doesn't change, so why re-evaluate
88     //                     every single time, when they're constant anyways, right? :)
89     for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
90         const Point & quadPoint = _quadratureRule->_points[qpIndex];
91
92         // Set Jacobian
93         Eigen::Matrix<double, SpatialDimension, SpatialDimension> jacobian
94             = Matrix<double, SpatialDimension, SpatialDimension>::Zero();
95
96         array<Point, NumberOfNodes> shapeFunctionDerivatives =
97             elementType.computeShapeFunctionDerivatives(quadPoint);
98         for (unsigned int nodeId = 0; nodeId < NumberOfNodes; ++nodeId) {
99             for (unsigned int i = 0; i < SpatialDimension; ++i) {
100                 for (unsigned int j = 0; j < SpatialDimension; ++j) {
101                     jacobian(i, j) += shapeFunctionDerivatives[nodeId](i) *
102                                     nodes[nodeId]._position(j);
103                 }
104             }
105
106             // Translate from 'barycentric coordinates' to 'physical coordinate system'
107             _shapeFunctions[qpIndex] // NOTE: This by itself is an array!
108                 = elementType.computeShapeFunctions(quadPoint);
109             _shapeFunctionDerivatives[qpIndex] = jacobian.inverse() *
110                 shapeFunctionDerivatives;
111
112             // Basically J * w_k on the HW sheet - w_k are the quadrature weights, which for
113             // the constant element (i.e. one quadrature point) would just simply be '1'
114             _weightedJacobian[qpIndex]
115                 = fabs(jacobian.determinant()) * _quadratureRule->_weights[qpIndex];
116             _volume += _weightedJacobian[qpIndex];
117         }
118         for (unsigned int nodeIndex = 0; nodeIndex < NumberOfNodes; nodeIndex++) {
119             _nodalWeights[nodeIndex] = _volume/NumberOfNodes;
120         }
121     }
122
123
124
125     // compute consistent mass matrix through numerical quadrature
126     MassMatrix
127     computeConsistentMassMatrix() const {
128
129         MassMatrix consistentMassMatrix = MassMatrix::Zero();
130
131         // TODO: Fill in the consistent mass matrix
132         for (unsigned int i = 0; i < NumberOfNodes*DegreesOfFreedom; i++){
133             for (unsigned int j = 0; j < NumberOfNodes*DegreesOfFreedom; j++){
134                 if (i == j){
135                     consistentMassMatrix (i,j) = 2.0;
136                 }
137                 else if ((i%DegreesOfFreedom)==(j%DegreesOfFreedom)){
138                     consistentMassMatrix (i,j) = 1.0;
139                 }

```

```

140     }
141 }
142 consistentMassMatrix = (consistentMassMatrix * _density * _volume) /
    (NumberOfNodes*DegreesOfFreedom);
143
144     return consistentMassMatrix;
145 }
146
147 array<DisplacementGradient, QP>
148 computeDispGradsAtGaussPoints(const NodalDisplacements & displacements) const {
149
150
151     // Initialize and zero array of displacment gradients at quadrature points
152     // NOTE: For array's, the fill-function looks somewhat different compared to
153     //         the equivalent for std::vector's. You need to hand over the pointer
154     //         to the first and last element in the array.
155     array<DisplacementGradient, QP> displacementGradients;
156     std::fill(displacementGradients.begin(), displacementGradients.end(),
        DisplacementGradient::Zero());
157
158     // Based on the discrete set of displacements at the nodes and using the shape
159     // function derivatives, evaluate the displacement gradients at the Gauss-points
160     // -> for reference, check out the last equation on p. 1 of the HW-sheet
161     for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
162         for (unsigned int i = 0; i < DegreesOfFreedom; i++) {
163             for (unsigned int j = 0; j < SpatialDimension; j++) {
164                 for (unsigned int nodeId = 0; nodeId < NumberOfNodes; nodeId++) {
165                     displacementGradients[qpIndex](SpatialDimension*i + j) +=
166                         displacements[nodeId](i) *
167                         _shapeFunctionDerivatives[qpIndex][nodeId](j);
168                 }
169             }
170         }
171
172     return displacementGradients;
173 }
174
175
176 double
177 computeEnergy(const NodalDisplacements & displacements ) const {
178
179     ignoreUnusedVariables(time);
180
181     // Based on the displacementGradients at the Q.P., first evaluate the stress
182     // tensor at all Q.P.
183     // and then evaluate the internal forces acting on the NumberOfNodes nodes in
184     // the element
185     const array<DisplacementGradient, QP> displacementGradients
186         = computeDispGradsAtGaussPoints(displacements);
187     double energy = 0.;
188     for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
189         energy += _weightedJacobian[qpIndex] *
190             _materialModel->computeEnergy(displacementGradients[qpIndex]);
191     }
192     return energy * _properties._elementMultiplier;
193 }
194
195 Forces
196 computeForces(const NodalDisplacements & displacements ) const {
197
198     ignoreUnusedVariables(time);
199
200     // Initialize array and set all NumberOfNodes entries to the zero-vector
201     Forces forcesAtNodes;
202     std::fill(forcesAtNodes.begin(), forcesAtNodes.end(), Vector::Zero());
203
204     // Based on the displacementGradients at the Q.P., first evaluate the stress
205     // tensor at all Q.P.
206     // and then evaluate the internal forces acting on the NumberOfNodes nodes in
207     // the element

```

```

206     const array<DisplacementGradient, QP> displacementGradients =
computeDispGradsAtGaussPoints(displacements);
207     for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
208         const Stress stressVector =
209             _materialModel->computeStress(displacementGradients[qpIndex]);
210         const StressTensor stressTensor =
211
212             Utilities::convertTensorFromVoigtToStandard<SpatialDimension, DegreesOfFreedom>
(stressVector);
213         for (size_t nodeId = 0; nodeId < NumberOfNodes; nodeId++){
214             forcesAtNodes[nodeId] += _weightedJacobian[qpIndex] *
215                 stressTensor * _shapeFunctionDerivatives[qpIndex][nodeId];
216         }
217
218         // Don't forget to multiply by _properties._elementMultiplier
219         // (1 for 3D, t(hickness) for 2D, (x-sectional) A(rea) for 1D simulations)
220         return forcesAtNodes * _properties._elementMultiplier;
221     }
222
223
224     StiffnessMatrix
225     computeStiffnessMatrix(const NodalDisplacements & displacements) const {
226
227         ignoreUnusedVariables(time);
228
229         // Initialize zero-stiffness matrix
230         Matrix<double, NumberOfNodes*DegreesOfFreedom, NumberOfNodes*DegreesOfFreedom>
stiffness
231             = Matrix<double, NumberOfNodes*DegreesOfFreedom,
232                 NumberOfNodes*DegreesOfFreedom>::Zero();
233
234         // Based on the displacementGradients at the Q.P., first evaluate the tangent
235         // and then evaluate the stiffness matrix based on those results
236         const array<DisplacementGradient, QP> displacementGradients =
computeDispGradsAtGaussPoints(displacements);
237         for (unsigned int qpIndex = 0; qpIndex < QP; qpIndex++) {
238             const TangentMatrix incrementalStiffnessMatrix =
239                 _materialModel->computeTangentMatrix(displacementGradients[qpIndex]);
240             for (unsigned int nodeId1 = 0; nodeId1 < NumberOfNodes; nodeId1++){
241                 for (unsigned int nodeId2 = 0; nodeId2 < NumberOfNodes; nodeId2++){
242                     for (unsigned int i = 0; i < DegreesOfFreedom; i++){
243                         const unsigned int voigtIndex1i =
244
245                             Utilities::convertStandardIndicesToVoigtIndex<DegreesOfFreedom>(nodeId1,
246                                 i);
247                         for (unsigned int k = 0; k < DegreesOfFreedom; k++){
248                             const unsigned int voigtIndex2k =
249
250                                 Utilities::convertStandardIndicesToVoigtIndex<DegreesOfFreedom>(nodeId
251                                     2, k);
252                             for (unsigned int j = 0; j < SpatialDimension; j++){
253                                 const unsigned int voigtIndexij =
254                                     Utilities::convertStandardIndicesToVoigtIndex<SpatialDimension>(i,
255                                         j);
256                                 for (unsigned int l = 0; l < SpatialDimension; l++){
257                                     const unsigned int voigtIndexkl =
258
259                                         Utilities::convertStandardIndicesToVoigtIndex<SpatialDimension>(k,
260                                             l);
261                                     stiffness(voigtIndex1i, voigtIndex2k) +=
262                                         _weightedJacobian[qpIndex] *
263                                             incrementalStiffnessMatrix(voigtIndexij, voigtIndexkl) *
264                                             _shapeFunctionDerivatives[qpIndex][nodeId1](j) *
265                                             _shapeFunctionDerivatives[qpIndex][nodeId2](l);
266                                 }
267                             }
268                         }
269                     }
270                 }
271             }
272         }

```

```

263
264     // Don't forget to multiply by _properties._elementMultiplier
265     // (1 for 3D, t(hickness) for 2D, (x-sectional) A(rea) for 1D simulations)
266     return stiffness * _properties._elementMultiplier;
267 }
268
269 array<size_t, NumberOfNodes>
270 getNodeIds() const {
271     return _nodeIds;
272 }
273
274 array<Vector, NumberOfNodes>
275 getNodePositions() const {
276     return _nodePositions;
277 }
278
279 array<double, QP>
280 computeWeightsAtGaussPoints() const {
281     return _weightedJacobian;
282 }
283
284 array<double, NumberOfNodes>
285 computeNodalWeights() const {
286     return _nodalWeights;
287 }
288
289
290 TangentMatrix
291 computeWeightedTangentMatrix(const NodalDisplacements & displacements) const {
292
293     TangentMatrix weightedTangentMatrix = TangentMatrix::Zero();
294
295     const array<DisplacementGradient, QP> displacementGradients =
296         computeDispGradsAtGaussPoints(displacements);
297     for (unsigned int qpIndex = 0; qpIndex < QP; qpIndex++) {
298         const TangentMatrix incrementalStiffnessMatrix =
299             _materialModel->computeTangentMatrix(displacementGradients[qpIndex]);
300         weightedTangentMatrix += _weightedJacobian[qpIndex] *
301             incrementalStiffnessMatrix;
302     }
303     return weightedTangentMatrix;
304 }
305
306 array<Stress, QP>
307 computeStressesAtGaussPoints(const NodalDisplacements & displacements) const {
308     array<Stress, QP> stresses;
309     const array<DisplacementGradient, QP> displacementGradients =
310         computeDispGradsAtGaussPoints(displacements);
311     for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
312         stresses[qpIndex] =
313             _materialModel->computeStress(displacementGradients[qpIndex]);
314     }
315     return stresses;
316 }
317
318 array<Strain, QP>
319 computeStrainsAtGaussPoints(const NodalDisplacements & displacements) const {
320     array<Strain, QP> strains;
321     const array<DisplacementGradient, QP> displacementGradients =
322         computeDispGradsAtGaussPoints(displacements);
323     for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
324         strains[qpIndex] =
325             _materialModel->computeStrain(displacementGradients[qpIndex]);
326     }
327     return strains;
328 }
329
330 private:
331
332 double _volume;

```

```

331     const Properties
332     array<size_t, NumberOfNodes>
333     array<Point, NumberOfNodes>
334     const QuadratureRule<SpatialDimension, QP> *
335     const MaterialModel *
336     array<double, QP>
337     array<array<double, NumberOfNodes>, QP>
338     array<array<Point, NumberOfNodes>, QP>
339     array<double, NumberOfNodes>
340     AllBoundaries
341 };
342
343 }
344
345 #endif //SIMPLICIAL_ELEMENT
346

```