```cpp
#ifndef SIMPLICIAL_ELEMENT
#define SIMPLICIAL_ELEMENT

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

namespace Elements {


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

public:

  struct Properties {

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

    const double _density;
    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 Matrix<double,
                 NumberOfNodes*DegreesOfFreedom,
                 NumberOfNodes*DegreesOfFreedom>            MassMatrix;

  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;


  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) {

```

```
  array<Point, NumberOfNodes> nodalPoints;

  // NOTE: nodes contains the global id's of all NumberOfNodes nodes of THIS element
  // NOTE: i.e. NumberOfNodes is not the mesh's total number of nodes, but the
  //       number of nodes in the element
  for (unsigned int nodeIndex = 0; nodeIndex < NumberOfNodes; ++nodeIndex) {
    _nodeIds[nodeIndex]       = nodes[nodeIndex]._id;
    _nodePositions[nodeIndex] = nodes[nodeIndex]._position;
  }


  _volume = 0.;

  // Quadrature Points : go through all of them and pre-evaluate
  //                      a) shapeFunctions
  //                      b) shapeFunctionDerivatives
  //                      c) weighted Jacobian (quadrature weight x J-determinant)
  //                     Why? Well they are defined with respect to the reference
  //                     configuration, which doesn't change, so why re-evaluate
  //                     every single time, when they're constant anyways, right? :)
  for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
    const Point & quadPoint = _quadratureRule->_points[qpIndex];

    // Set Jacobian
    Eigen::Matrix<double, SpatialDimension, SpatialDimension> jacobian
      = Matrix<double, SpatialDimension, SpatialDimension>::Zero();

    array<Point, NumberOfNodes> shapeFunctionDerivatives =
      elementType.computeShapeFunctionDerivatives(quadPoint);
    for (unsigned int nodeId = 0; nodeId < NumberOfNodes; ++nodeId) {
      for (unsigned int i = 0; i < SpatialDimension; ++i) {
        for (unsigned int j = 0; j < SpatialDimension; ++j) {
          jacobian(i, j) += shapeFunctionDerivatives[nodeId](i) *
          nodes[nodeId]._position(j);
        }
      }
    }

    // Translate from 'barycentric coordinates' to 'physical coordinate system'
    _shapeFunctions[qpIndex]            // NOTE: This by itself is an array!
      = elementType.computeShapeFunctions(quadPoint);
    _shapeFunctionDerivatives[qpIndex] = jacobian.inverse() *
    shapeFunctionDerivatives;

    // Basically J * w_k on the HW sheet - w_k are the quadrature weights, which for
    // the constant element (i.e. one quadrature point) would just simply be '1'
    _weightedJacobian[qpIndex]
      = fabs(jacobian.determinant()) * _quadratureRule->_weights[qpIndex];
    _volume += _weightedJacobian[qpIndex];
  }
  for (unsigned int nodeIndex = 0; nodeIndex < NumberOfNodes; nodeIndex++) {
    _nodalWeights[nodeIndex] = _volume/NumberOfNodes;
  }
}




// compute consistent mass matrix through numerical quadrature
MassMatrix
computeConsistentMassMatrix() const {

    MassMatrix consistentMassMatrix = MassMatrix::Zero();

    // TODO: Fill in the consistent mass matrix
    for (unsigned int i = 0; i < NumberOfNodes*DegreesOfFreedom; i++){
        for (unsigned int j = 0; j < NumberOfNodes*DegreesOfFreedom; j++){
            if (i == j){
                consistentMassMatrix (i,j) = 2.0;
            }
            else if ((i%DegreesOfFreedom)==(j%DegreesOfFreedom)){
                consistentMassMatrix (i,j) = 1.0;
            }
```

```
          }
      }
      consistentMassMatrix = (consistentMassMatrix * _density * _volume) / (NumberOfNodes*DegreesOfFreedom);

      return consistentMassMatrix;
  }

  array<DisplacementGradient, QP>
  computeDispGradsAtGaussPoints(const NodalDisplacements & displacements) const {


    // Initialize and zero array of displacment gradients at quadrature points
    // NOTE: For array's, the fill-function looks somewhat different compared to
    //       the equivalent for std::vector's. You need to hand over the pointer
    //       to the first and last element in the array.
    array<DisplacementGradient, QP> displacementGradients;
    std::fill(displacementGradients.begin(), displacementGradients.end(), DisplacementGradient::Zero());

    // Based on the discrete set of displacements at the nodes and using the shape
    // function derivatives, evaluate the displacement gradients at the Gauss-points
    // -> for reference, check out the last equation on p. 1 of the HW-sheet
    for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
      for (unsigned int i = 0; i < DegreesOfFreedom; i++) {
        for (unsigned int j = 0; j < SpatialDimension; j++) {
          for (unsigned int nodeId = 0; nodeId < NumberOfNodes; nodeId++) {
            displacementGradients[qpIndex](SpatialDimension*i + j) +=
              displacements[nodeId](i) * _shapeFunctionDerivatives[qpIndex][nodeId](j);
          }
        }
      }
    }

    return displacementGradients;
  }


  double
  computeEnergy(const NodalDisplacements &  displacements ) const {

    ignoreUnusedVariables(time);

    // Based on the displacementGradients at the Q.P., first evaluate the stress tensor at all Q.P.
    // and then evaluate the internal forces acting on the NumberOfNodes nodes in the element
    const array<DisplacementGradient, QP> displacementGradients
      = computeDispGradsAtGaussPoints(displacements);
    double energy = 0.;
    for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
      energy += _weightedJacobian[qpIndex] *
        _materialModel->computeEnergy(displacementGradients[qpIndex]);
    }

    return energy * _properties._elementMultiplier;
  }


  Forces
  computeForces(const NodalDisplacements &  displacements ) const {

    ignoreUnusedVariables(time);

    // Initialize array and set all NumberOfNodes entries to the zero-vector
    Forces forcesAtNodes;
    std::fill(forcesAtNodes.begin(), forcesAtNodes.end(), Vector::Zero());

    // Based on the displacementGradients at the Q.P., first evaluate the stress tensor at all Q.P.
    // and then evaluate the internal forces acting on the NumberOfNodes nodes in the element
```

```
    const array<DisplacementGradient, QP> displacementGradients = computeDispGradsAtGaussPoints(displacements);
    for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
      const Stress stressVector =
        _materialModel->computeStress(displacementGradients[qpIndex]);
      const StressTensor stressTensor =
        Utilities::convertTensorFromVoigtToStandard<SpatialDimension,DegreesOfFreedom>(stressVector);
      for (size_t nodeId = 0; nodeId < NumberOfNodes; nodeId++){
        forcesAtNodes[nodeId] += _weightedJacobian[qpIndex] *
          stressTensor * _shapeFunctionDerivatives[qpIndex][nodeId];
      }
    }

    // Don't forget to multiply by _properties._elementMultiplier
    // (1 for 3D, t(hickness) for 2D, (x-sectional) A(rea) for 1D simulations)
    return forcesAtNodes * _properties._elementMultiplier;
  }


  StiffnessMatrix
  computeStiffnessMatrix(const NodalDisplacements & displacements) const {

    ignoreUnusedVariables(time);

    // Initialize zero-stiffness matrix
    Matrix<double, NumberOfNodes*DegreesOfFreedom, NumberOfNodes*DegreesOfFreedom> stiffness
      = Matrix<double, NumberOfNodes*DegreesOfFreedom, NumberOfNodes*DegreesOfFreedom>::Zero();

    // Based on the displacementGradients at the Q.P., first evaluate the tangent matrix at all Q.P.
    // and then evaluate the stiffness matrix based on those results
    const array<DisplacementGradient, QP> displacementGradients = computeDispGradsAtGaussPoints(displacements);
    for (unsigned int qpIndex = 0; qpIndex < QP; qpIndex++) {
      const TangentMatrix incrementalStiffnessMatrix =
        _materialModel->computeTangentMatrix(displacementGradients[qpIndex]);
      for (unsigned int nodeId1 = 0; nodeId1 < NumberOfNodes; nodeId1++){
        for (unsigned int nodeId2 = 0; nodeId2 < NumberOfNodes; nodeId2++){
          for (unsigned int i = 0; i < DegreesOfFreedom; i++){
            const unsigned int voigtIndex1i =
              Utilities::convertStandardIndicesToVoigtIndex<DegreesOfFreedom>(nodeId1, i);
            for (unsigned int k = 0; k < DegreesOfFreedom; k++){
              const unsigned int voigtIndex2k =
                Utilities::convertStandardIndicesToVoigtIndex<DegreesOfFreedom>(nodeId2, k);
              for (unsigned int j = 0; j < SpatialDimension; j++){
                const unsigned int voigtIndexij =
                  Utilities::convertStandardIndicesToVoigtIndex<SpatialDimension>(i, j);
                for (unsigned int l = 0; l < SpatialDimension; l++){
                  const unsigned int voigtIndexkl =
                    Utilities::convertStandardIndicesToVoigtIndex<SpatialDimension>(k, l);
                  stiffness(voigtIndex1i, voigtIndex2k) +=
                    _weightedJacobian[qpIndex] * incrementalStiffnessMatrix(voigtIndexij,voigtIndexkl) *
                    _shapeFunctionDerivatives[qpIndex][nodeId1](j) * _shapeFunctionDerivatives[qpIndex][nodeId2](l);
                }
              }
            }
          }
        }
      }
    }
```

```
      // Don't forget to multiply by _properties._elementMultiplier
      // (1 for 3D, t(hickness) for 2D, (x-sectional) A(rea) for 1D simulations)
      return stiffness * _properties._elementMultiplier;
    }

    array<size_t, NumberOfNodes>
    getNodeIds() const {
      return _nodeIds;
    }

    array<Vector, NumberOfNodes>
    getNodePositions() const {
      return _nodePositions;
    }

    array<double, QP>
    computeWeightsAtGaussPoints() const {
      return _weightedJacobian;
    }

    array<double, NumberOfNodes>
    computeNodalWeights() const {
      return _nodalWeights;
    }


    TangentMatrix
    computeWeightedTangentMatrix(const NodalDisplacements & displacements) const {

      TangentMatrix weightedTangentMatrix = TangentMatrix::Zero();

      const array<DisplacementGradient, QP> displacementGradients =
      computeDispGradsAtGaussPoints(displacements);
      for (unsigned int qpIndex = 0; qpIndex < QP; qpIndex++) {
        const TangentMatrix incrementalStiffnessMatrix =
          _materialModel->computeTangentMatrix(displacementGradients[qpIndex]);
        weightedTangentMatrix += _weightedJacobian[qpIndex] *
        incrementalStiffnessMatrix;
      }
      return weightedTangentMatrix;
    }


    array<Stress, QP>
    computeStressesAtGaussPoints(const NodalDisplacements & displacements) const {
      array<Stress, QP> stresses;
      const array<DisplacementGradient, QP> displacementGradients =
      computeDispGradsAtGaussPoints(displacements);
      for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
        stresses[qpIndex] =
          _materialModel->computeStress(displacementGradients[qpIndex]);
      }
      return stresses;
    }


    array<Strain, QP>
    computeStrainsAtGaussPoints(const NodalDisplacements & displacements) const {
      array<Strain, QP> strains;
      const array<DisplacementGradient, QP> displacementGradients =
      computeDispGradsAtGaussPoints(displacements);
      for (unsigned int qpIndex = 0; qpIndex < QP; ++qpIndex) {
        strains[qpIndex] =
        _materialModel->computeStrain(displacementGradients[qpIndex]);
      }
      return strains;
    }


    private:

    double                                        _volume;
```

```
  const Properties                                        _properties;
  array<size_t, NumberOfNodes>                            _nodeIds;
  array<Point, NumberOfNodes>                             _nodePositions;
  const QuadratureRule<SpatialDimension, QP> *            _quadratureRule;
  const MaterialModel *                                   _materialModel;
  array<double, QP>                                       _weightedJacobian;
  array<array<double, NumberOfNodes>, QP>                 _shapeFunctions;
  array<array<Point, NumberOfNodes>, QP>                  _shapeFunctionDerivatives;
  array<double, NumberOfNodes>                            _nodalWeights;
  AllBoundaries                                           _allBoundaries;
};

}

#endif //SIMPLICIAL_ELEMENT

```