```cpp
#ifndef MATERIAL_MODEL_LINEAR_ELASTIC
#define MATERIAL_MODEL_LINEAR_ELASTIC

#include "/src/Definitions.h"
#include "PJ2Utilities.h"
#include "Eigen/Eigen"
#include <iostream>
#include <Eigen/Dense>
using namespace Eigen;
using namespace std;

namespace MaterialModels {

class MaterialModelLinearElastic {

public:

  // TODO: Given that DisplacementGradient, Stress, Strain and TangentMatrix are in Voigt-notation,
  //       determine the number of rows and columns of each of those matrices
  // REMINDER: The second template parameter for Matrix is the number or rows, the third template
  //           parameter denotes the number of columns
  typedef Matrix<double, 9, 1>  DisplacementGradient;
  typedef Matrix<double, 9, 1>  Stress;
  typedef Matrix<double, 9, 1>  Strain;
  typedef Matrix<double, 3, 3>  StandardStrainMatrix;
  typedef Matrix<double, 9, 9>  TangentMatrix;

  // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  // CAREFUL: After changing the dimensions, please make sure to also change the hard-coded 1's below. (Otherwise you will get a large Eigen-mistake)

  // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


  MaterialModelLinearElastic(const double youngsModulus, const double poissonsRatio):
    // TODO: Set the Lame parameters _lambda and _mu based on the input youngsModulus
    //       and poissonsRatio
    // CAUTION: _lambda and _mu are endowed with the 'const'-qualifier and hence can only
    //           be defined in this way as part of the constructor.
    _lambda((poissonsRatio*youngsModulus)/((1+poissonsRatio)*(1-2*poissonsRatio))),
    _mu    (youngsModulus/(1+poissonsRatio)/2){

    //ignoreUnusedVariables(youngsModulus,poissonsRatio); // (You can remove this line as
                                                          //  soon as you finished and you're
                                                          //  happy with your constructor)

    }


  Strain
  computeStrain(const DisplacementGradient & displacementGradient) const {

    // TODO: Evaluate the strain vector (here: epsilon) based on the displacement gradient.
    // NOTE: The functionality to convert from standard to Voigt notation can be found
    //       in the Utilities namespace (see PJ2Utilities.h) (it might come in handy here...)
    Strain strain = Strain::Zero();
    Matrix<double,3,3> standardDisplacementGradientMatrix
      = Utilities::convertTensorFromVoigtToStandard<3>(displacementGradient);

    StandardStrainMatrix standardStrainMatrix=  (standardDisplacementGradientMatrix
```

```
      + standardDisplacementGradientMatrix.transpose())/2;

      strain = Utilities::convertTensorFromStandardToVoigt<3>(standardStrainMatrix);


      // HINT: We will show once, how to convert a 3x3 matrix to a 9x1 matrix.
      //       You can remove this if you feel like you've understood the concept.
      //Matrix<double,3,3> some3x3Matrix = Matrix<double,3,3>::Identity();
      //Matrix<double,9,1> some9x1Matrix
      //  = Utilities::convertTensorFromStandardToVoigt<3>(some3x3Matrix);
      //Matrix<double,3,3> reconverted3x3Matrix
      //  = Utilities::convertTensorFromVoigtToStandard<3>(some9x1Matrix);


      // TODO: Remove the following ignoreUnusedVariables line
      //ignoreUnusedVariable(displacementGradient);
      //ignoreUnusedVariable(reconverted3x3Matrix);


      return strain;
    };


    double
    computeEnergy(const DisplacementGradient & displacementGradient) const {

      // Evaluate strain in vector-form from displacementGradient using the
      // computeStrain function you've defined before:
      Strain strainVector = computeStrain(displacementGradient);
      Matrix <double,3,3>  epsilon =
      Utilities::convertTensorFromVoigtToStandard<3>(strainVector);

      // TODO: Remove the following ignoreUnusedVariables lines
      //ignoreUnusedVariables(epsilon);

      // TODO: If necessary, based on strain matrix, evaluate useful scalars such as the
      //       trace of the strain
      double TraceOfStrain = 0.0;
      for (int i=0; i<3; i++){
          for (int j=0; j<3; j++){
              if (i == j){
                  TraceOfStrain += epsilon(i,j);
              }
          }
      }
      double epsilonDotEpsilon = 0.0;
       for (int i=0; i<3; i++){
          for (int j=0; j<3; j++){
              epsilonDotEpsilon += epsilon(i,j) * epsilon(i,j);
          }
       }


      // TODO: Evaluate the energy density
      double energyDensity = 0.0;
      energyDensity = _lambda * (TraceOfStrain*TraceOfStrain)/2 + _mu *
      epsilonDotEpsilon;


      // TODO: Remove the following ignoreUnusedVariables line
      //ignoreUnusedVariables(displacementGradient,strainVector);


      // Return
      return energyDensity;
    }


    Stress
    computeStress(const DisplacementGradient & displacementGradient) const {

      // TODO: Evaluate strain in vector-form from displacementGradient using the
      //       computeStrain function you've defined before
```

```
131       Strain strainVector = Strain::Zero();
132       strainVector = computeStrain(displacementGradient);
133       Matrix <double,3,3> epsilon =
          Utilities::convertTensorFromVoigtToStandard<3>(strainVector);
134
135
136       // TODO: If necessary, based on strain matrix, evaluate useful scalars such as the
137       //       trace of the strain
138
139       double TraceOfStrain = 0.0;
140       for (int i=0; i<3; i++){
141           for (int j=0; j<3; j++){
142               if (i == j){
143                   TraceOfStrain += epsilon(i,j);
144               }
145           }
146       }
147
148
149       // TODO: Evaluate the 2nd order stress tensor in Voigt-form (!)
150       Stress stress = Stress::Zero();
151       StandardStrainMatrix sigma =
          Utilities::convertTensorFromVoigtToStandard<3>(stress);
152       for (int i=0; i<3; i++){
153           for (int j=0; j<3; j++){
154               if (i == j){
155                   sigma(i,j) = _lambda * TraceOfStrain + 2 * _mu * epsilon(i,j);
156               }
157               else {
158                   sigma(i,j) = 2 * _mu * epsilon(i,j);
159               }
160
161           }
162       }
163       stress = Utilities::convertTensorFromStandardToVoigt<3>(sigma);
164       // TODO: Remove the following ignoreUnusedVariables line
165       //ignoreUnusedVariables(displacementGradient,strainVector);
166
167       // Return
168       return  stress;
169     };
170
171     TangentMatrix
172     computeTangentMatrix(const DisplacementGradient & displacementGradient) const {
173
174       // TODO: We keep our lives simple and define a 4th order tensor as an array first.
175       //       1st: Set the right dimensions (i.e. replace the 1's by the correct
          dimension)
176       //       2nd: Set all entries of tangentMatrixAsArray. You may have to use
          nested for-loops.
177       array<array<array<array<double,3>,3>,3>,3> tangentMatrixAsArray;
178       for (int i = 0; i<3; i++){
179           for (int j = 0; j<3; j++){
180               for (int k = 0; k<3; k++){
181                   for (int l = 0; l<3; l++){
182                       if ((i==j) && (k==l)){
183                           if (i==k){
184                               tangentMatrixAsArray[i][j][k][l] = _lambda + 2*_mu;
185                           }
186                           else {
187                               tangentMatrixAsArray[i][j][k][l] = _lambda;
188                           }
189                       }
190                       else if ((i!=j )|| (k !=l)){
191                           if ((i ==k) && (j==l) && (i == l) && (j==k)){
192                               tangentMatrixAsArray[i][j][k][l] = 2*_mu;
193                           }
194                           else if ((i ==k) && (j==l)){
195                               tangentMatrixAsArray[i][j][k][l] = _mu;
196                           }
197                           else if ((i == l) && (j==k)){
198                               tangentMatrixAsArray[i][j][k][l] = _mu;
199                           }
```

```
                        else {
                            tangentMatrixAsArray[i][j][k][l] = 0.0;
                        }
                    }
                }
            }
        }
    }

    // TODO: Simply replace the 1 again by the right dimension
    // We'll take care of the rest of the conversion. The functionality to convert between standard
    // and Voigt is found in the Utilities namespace in PJ2Utilities.h.
    TangentMatrix tangentMatrix
      =
      Utilities::convertFourthOrderTensorFromStandardToVoigt<3>(tangentMatrixAsArray);


    // TODO: Remove the following ignoreUnusedVariables line
    ignoreUnusedVariables(displacementGradient);


    return tangentMatrix;
  };


  private:
    const double _lambda;
    const double _mu;
};


} // MaterialModels
#endif // MATERIAL_MODEL_LINEAR_ELASTIC
```