```cpp
#ifndef MATERIAL_MODEL_HYPERELASTIC
#define MATERIAL_MODEL_HYPERELASTIC

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

namespace MaterialModels {

class MaterialModelHyperelastic {

  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)

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


  // Constructor
  MaterialModelHyperelastic(const double lambda, const double mu):
    // TODO: Set the Lame parameters _lambda and _mu based on the input
    // 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(lambda),
    _mu(mu){

    }


  Strain
  computeStrain(const DisplacementGradient & displacementGradient) const {

    // TODO: Evaluate the strain vector (here: F) 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> identityMatrix3x3 = Matrix<double,3,3>::Identity();
    Matrix<double,9,1> identityMatrixVectorForm9x1 = Utilities::convertTensorFromStandardToVoigt<3>(identityMatrix3x3);
    for (int i = 0; i<9; i++){
      strain (i,0) += identityMatrixVectorForm9x1(i,0) + displacementGradient(i,0);
    };

    // 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 lines
    //ignoreUnusedVariables(displacementGradient);
    //ignoreUnusedVariables(reconverted3x3Matrix);

    return strain;

  };


  double
  computeEnergy(const DisplacementGradient & displacementGradient) const {

    // Evaluate F based on displacementGradient using computeStrain
    Strain strainVector = computeStrain(displacementGradient);
    StandardStrainMatrix F =
    Utilities::convertTensorFromVoigtToStandard<3>(strainVector);


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

    // TODO: If necessary, evaluate useful scalars such as the I1 invariant
    //       and the volumetric expansion J based on strainMatrix

    StandardStrainMatrix FMatrixTranspose = F.transpose();
    double determinantOfF = F.determinant();
    Matrix<double, 3, 3> CMatrix;
    /*for (int i = 0; i <3; i++){
      for (int j = 0; j<3; j++){
        CMatrix (i,j) = F(i,j)* FMatrixTranspose(i,j);
      }
    }*/
    CMatrix = FMatrixTranspose*F;
    double traceOfC = 0.0;
    for (int i = 0; i < 3; i++){
      for (int j = 0; j<3; j++){
        if (i ==j){
          traceOfC += CMatrix(i,j);
        }
      }
    }
    // traceOfC = CMatrix.trace()
    //double determinantOfF = 0.0;

    // TODO: Evaluate the energy density
    double energyDensity = 0.0;
    energyDensity = _mu * (traceOfC - 3)/2 +
    _lambda*log(determinantOfF)*log(determinantOfF)/2 - _mu * log(determinantOfF);


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

    // Return
    return energyDensity;
  }


  Stress
  computeStress(const DisplacementGradient & displacementGradient) const {

    // TODO : Evaluate F based on displacementGradient using computeStrain
    Strain strainVector = Strain::Zero();
    strainVector = computeStrain(displacementGradient);
    StandardStrainMatrix F =
    Utilities::convertTensorFromVoigtToStandard<3>(strainVector);


    // TODO: If necessary, evaluate useful scalars such as the I1 invariant
```

```
    //       and the volumetric expansion J or tensors such as the right Cauchy-Green
    //       tensor C or the inverse-transpose of F
    double determinantOfF = F.determinant();
    StandardStrainMatrix inverseTransposeOfF = F.inverse().transpose();

    // TODO: Based on the possibly-useful scalars and tensors, you've obtained above,
    //       determine the stress tensor.
    Stress stress = Stress::Zero();
    StandardStrainMatrix stressMatrixP = Utilities::convertTensorFromVoigtToStandard<3>(stress);
    stressMatrixP = _mu * F + _lambda * log (determinantOfF) * inverseTransposeOfF - _mu * inverseTransposeOfF;
    stress = Utilities::convertTensorFromStandardToVoigt<3>(stressMatrixP);

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


    // Return
    return stress;
  };


  TangentMatrix
  computeTangentMatrix(const DisplacementGradient & displacementGradient) const {

    // TODO : Evaluate F based on displacementGradient using computeStrain
    Strain strainVector = Strain::Zero();
    strainVector = computeStrain(displacementGradient);
    StandardStrainMatrix F = Utilities::convertTensorFromVoigtToStandard<3>(strainVector);

    // TODO: If necessary, evaluate useful scalars such as the I1 invariant
    //       and the volumetric expansion J or tensors such as the left Cauchy-Green
    //       tensor C or the inverse-transpose of F
    double determinantOfF = F.determinant();
    StandardStrainMatrix inverseOfF = F.inverse();




    // TODO: We keep our lives simple and define a 4th order tensor as an array first.
    //       1st: Set the right dimensions (i.e. replace the 1's by the correct dimension)
    //       2nd: Set all entries of tangentMatrixAsArray. You may have to use nested for-loops.
    array<array<array<array<double,3>,3>,3>,3> tangentMatrixAsArray;
    for (int i = 0; i<3; i++){
        for (int J = 0; J<3; J++){
            for (int k = 0; k<3; k++){
                for (int L = 0; L<3; L++){
                    if ((i == k) && (J == k)){
                        tangentMatrixAsArray [i][J][k][L] = _mu *(1 + inverseOfF(J,k) * inverseOfF(L,i)) + _lambda * (inverseOfF(L,k) * inverseOfF(J,i) - log (determinantOfF)*inverseOfF(J,k) * inverseOfF(L,i));
                    }
                    else {
                        tangentMatrixAsArray [i][J][k][L] = _mu *(inverseOfF(J,k) * inverseOfF(L,i)) + _lambda * (inverseOfF(L,k) * inverseOfF(J,i) - log (determinantOfF)*inverseOfF(J,k) * inverseOfF(L,i));
                    }
                }
            }
        }
    }

    // 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,strainVector);


    // Return
    return tangentMatrix;
  };



  private:
  const double _lambda;
  const double _mu;

  };
}
#endif // MATERIAL_MODEL_HYPERELASTIC
```