Project #5

assigned: Thursday, November 23rd, 2017
due: Thursday, December 7th, 2017, 17:00
please hand in to one of the TAs in LEE N203

Problem 1: assembler (25 points).

Let us write a class `Assembler` which realizes the assembly of finite element quantities into their respective global quantities, as discussed in class. The new class should have the following methods:

- The *constructor* receives and stores the vector of all n_e elements and the total number of nodes, N.
- Method `assembleEnergy` receives the global vector $\boldsymbol{U}^h$ and computes the total energy $I = \sum_{e=1}^{n_e} I_e$ from all individual elements.
- Method `assembleForceVector` receives the global vector $\boldsymbol{U}^h$ and computes the global force vector $\boldsymbol{F} = \{\boldsymbol{F}^1, \ldots, \boldsymbol{F}^N\}$ by assembling the force vectors obtained from each individual element.
- Method `assembleStiffnessMatrix` receives the global vector $\boldsymbol{U}^h$ and computes the global stiffness matrix $\boldsymbol{T}$ by assembling the stiffness matrices $\boldsymbol{T}_e$ from each individual element.

Note that each method receives the *global* displacement vector $\boldsymbol{U}^h = \{\boldsymbol{u}^1, \ldots, \boldsymbol{u}^N\}$, from which we must extract the element displacement vectors $\boldsymbol{U}_e^h = \{\boldsymbol{u}_e^0, \ldots, \boldsymbol{u}_n^1\}$ for each element. Then, you can pass $\boldsymbol{U}_e^h$ to each element and ask for its contributions and assemble those. You can use the provided helper function `distributeGlobalVectorToLocalVectors` to turn global vectors of the type `Eigen::VectorXd` into nodal-wise vectors of type `vector<ElementVector>`, and vice-versa.

Note that since the vectors and matrices can become very large, we pass pointers instead of the actual vectors and matrices. Also, we simply write $\boldsymbol{F}$ for what is in fact $\boldsymbol{F}_{\text{int}} - \boldsymbol{F}_{\text{ext}}$ to avoid confusion with the minus sign.

Problem 2: Newton-Raphson solver (30 points).

Let us write a class `Solver` which takes `Assemblers` and a list of essential boundary conditions and computes the displacement solution by Newton-Raphson iteration, as discussed in class. Note that in case we want several types of elements, we should allow for more than one assembler – let us use *two* here and implement a simple way to deal with only one assembler as a special-case constructor. The `Solver` needs a method `computeSolution` which does the following:

(1) Start with an initial guess $\boldsymbol{U}_0^h$ for the global displacements (e.g. all zero except for essential BCs).
(2) Use both assemblers to compute the global force vector $\boldsymbol{F}(\boldsymbol{U}_n^h)$ and the global stiffness matrix $\boldsymbol{T}(\boldsymbol{U}_n^h)$. For several assemblers, simply add up the contributions from each assembler.
(3) Modify $\boldsymbol{F}$ and $\boldsymbol{T}$ according to the essential boundary conditions, using the method of substitution (i.e., replace rows of $\boldsymbol{T}$ by 0s and a 1 on the diagonal, and replace the corresponding entry in $\boldsymbol{F}$).
(4) Compute the residue $r = |\boldsymbol{F}|$ and check if $r <$ tolerance.

Note: For the above check, you must use the global forces $\boldsymbol{F}$ *with essential BCs replaced by* 0*s* as discussed in class (otherwise, the reaction forces to essential BCs will never go to zero).

(5) If $r >$ tolerance, perform the next step of Newton-Raphson iteration:

$$\boldsymbol{U}^h_{n+1} = \boldsymbol{U}^h_n - \boldsymbol{T}(\boldsymbol{U}^h_n)^{-1}\boldsymbol{F}(\boldsymbol{U}^h_n).$$

Update the displacements, recompute stiffness matrix and force vector, re-apply BCs, and check the residue again.

(6) Stop if either the tolerance is met or the maximum number of iterations (defined in the constructor) is reached.

(7) Return the solution in terms of arrays of node-wise vectors of displacements.

Hint: we have defined a class `essentialBoundaryCondition` which takes a triple of values $\{a, i, \hat{u}\}$, viz. a nodeId a, a coordinate $i \in [0, d-1]$ and an imposed displacement $\hat{u}$ to be applied to u^a_i. This way we can easily define essential BCs in Main and pass those to the solver to be implemented.

Problem 3: boundary value problem I (20 points).

Let us use all of the classes developed so far to simulate the uniaxial stretching of a cantilever rod of size $L \times H$ in 2D, clamped at its left and right ends (see the sketch below). To this end,

(i) create a 2D *mesh* of triangular elements (see Project #1),

(ii) create a *material model* (our generalized versions of Project #2 work in 2D and 3D),

(iii) create an *assembler* and add all triangular *elements* (see Project #4) from the mesh,

(iv) create *essential BCs* for the left and right edges,

(v) create a *solver* and hand the assembler to the solver,

(vi) *solve* the problem using the soler; once a solution is obtained, *visually display* the stress and strain distributions in the deformed configuration using Paraview. Plot both the discontinuous *element stresses* and the *interpolated stresses* (functionality provided by us).

Verify that the *linear elastic* material model makes the solution converge in a single step, whereas the *Neo-Hookean* model requires several iteration steps. **Simulation parameters** are given in the table below.

Problem 4: boundary value problem II (25 points).

As a second problem, let us simulate the deformation of a truss network under its own weight (using our utilities from Project #3) while fixed at the bottom (see the sketch below). The setup is similar to the above (feel free to copy and paste), and parameters are again given below. To do is:

(i) read in a *mesh* of 2D two-node nonlinear bars (we provide the read-in utility),

(ii) create a *linear elastic 1D material model* (see Project #3),

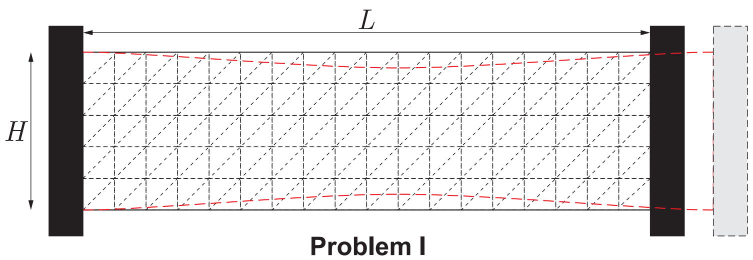
(iii) create an *assembler* and add all bar *elements* (see Project #3) from the mesh,

(iv) create a *second assembler* and add all *external force elements* (see Project #3),

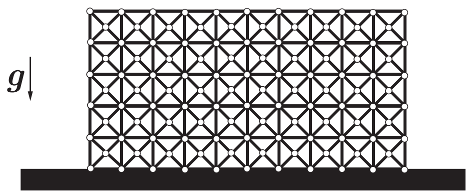
(v) create *essential BCs* for the bottom edge,

(vi) create a solver and hand both assemblers to the solver,

(vii) *solve* the problem using the solver; *visually display* the bar stresses in the deformed configuration using Paraview.



Problem I



Problem II

Please use the following **parameters** for your simulations to receive sensible result graphics:

Problem I:

material model (rubber): μ =1 GPa, λ =4 GPa
model size: L =1 m, H =0.4 m
mesh size: 20×8 elements
stretch: 20%
BCs: leftmost nodes completely fixed, rightmost nodes displacements horizontally and fixed vertically

Problem II:

material model (polymer): E =0.1 GPa, ρ =1.0$\cdot$ 10^5 kg/m^3
gravity: 9.81 m/s^2
BCs: all bottom nodes completely fixed

Note: The value of the density in Problem II does by no means represent a physically realistic value. It is simply chosen so as to allow for reasonably high strains to appear that can be nicely displayed. Furthermore, please note, that in your main file you can set Young's modulus to 0.1 instead of $0.1 \cdot 10^9$ and density accordingly as 1.0$\cdot$ 10^{-4} – simply be reminded that the resulting force for example will then be given in GPa$\cdot$m=GN. Similarly, in Problem I using moduli of 1 and 4 will result in stress values that are also to be interpreted as having units of GPa.

total: 100 points