deal.II Workshop @ hereon/TUHH

Day 3: solid mechanics

Peter Munch[1,2,3], Daniel Paukner[3], Ingo Scheider[3]

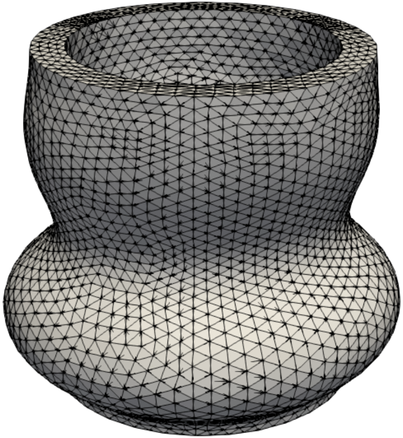
[1]deal.II developer

[2]Institute for Computational Mechanics, Technical University of Munich, Germany

[3]Kontinuumssimulationen, Institut für Werkstoffsystem-Modellierung, Helmholtz-Zentrum hereon, Germany

April 21, 2021

Motivation



$$\mathrm{Div}(\underline{\boldsymbol{F}} \cdot \underline{\boldsymbol{S}}(\underline{\boldsymbol{E}})) + \rho_0 \underline{\hat{\boldsymbol{b}}} = 0$$

FEM

How can deal.II help?

Organization: timetable

Organization:

- 9:00-12:00 (Monday-Wednesday): presentation/workshop with open end
- 9:00-9:30 (Monday): presentation round
- 9:00-9:30 (Tuesday, Wednesday): question time

Topics:

- Monday: introduction into FEM, overview of deal.II, mesh handling
- Tuesday: Poisson problem (heat-conduction problem)
- Wednesday: phase-field methods for sintering (guest presentation), solid mechanics

Part 1:

Wrap up of day 2

General questions

- mesh smoothing via `Mesquite`
- non-linear solver via `SUNDIAL`'s `KINSOL` package ▷ *see also step-77 (WIP)*

```
SUNDIALS::KINSOL<Vector<double>> nonlinear_solver(additional_data);

nonlinear_solver.reinit_vector       = [&](Vector<double> &x) {/*...*/};
nonlinear_solver.residual            = [&](const auto &evaluation_point, auto &residual) {/*...*/};
nonlinear_solver.setup_jacobian      = [&](const auto &current_u, const auto & /*current_f*/) {/*...*/};
nonlinear_solver.solve_with_jacobian = [&](const auto &rhs, auto& dst, const auto tolerance) {/*...*/};

nonlinear_solver.solve(current_solution);
```

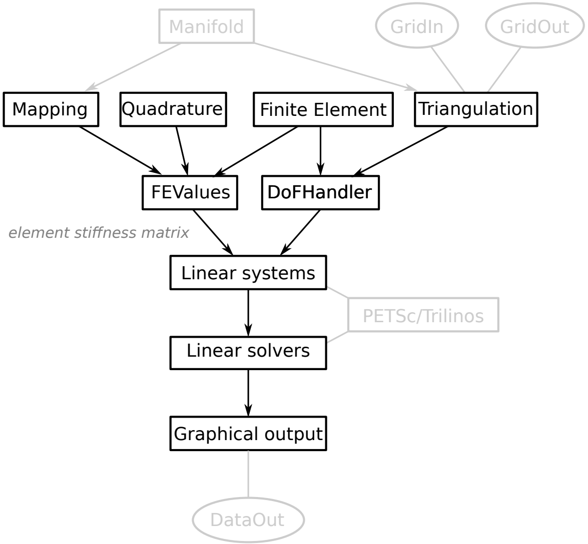
- applying concentrated forces at nodes
- quadratic serendipity element vs. Lagrange element: 8 vs. 9 DoFs

Main modules

needed from a FEM library:

- mesh handling
- finite elements
- quadrature rules
- mapping rules
- assembly procedure
- linear solver

deal.II main modules $\rightarrow$



Task 2: solution

Solve:

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} + \boldsymbol{g}$$

with:

$$\mathbf{K}_{ij}^{(e)} = \sum_q (\nabla N_{iq}, \nabla N_{jq}) \cdot |J_q| \times w_q, \ \mathbf{f}_i^{(e)} = \sum_q (N_{iq}, f) \cdot |J_q| \times w_q, \ \mathbf{g}_i^{(e)} = \sum_q (N_{iq}, g) \cdot |J_q| \times w_q$$

tasks:

- a) implement element stiffness matrix and right-hand-side vector
- b) modify DBC such that $h \neq 0$
- c) implement NBC such that $g \neq 0$

Task 2: solution (cont.)

Task 2a with $f(\underline{\boldsymbol{x}}) = ||\underline{\boldsymbol{x}}||$:

- initialization of `FEValues`:

```
FEValues<dim> fe_values(mapping, fe, quad,
                        update_values | update_gradients | update_JxW_values | update_quadrature_points);
```

- computation of element stiffness matrix and right-hand-side vector :

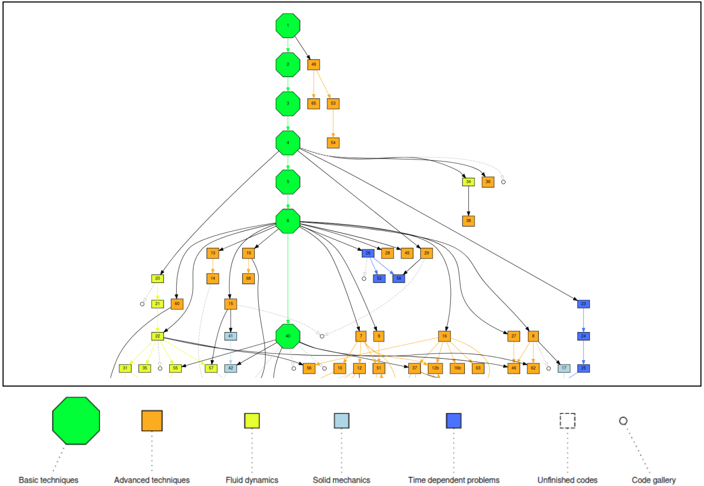
```
      // loop over cell dofs
      for (const auto q : fe_values.quadrature_point_indices())
        {
          for (const auto i : fe_values.dof_indices())
            for (const auto j : fe_values.dof_indices())
              cell_matrix(i, j) +=
                (fe_values.shape_grad(i, q) * // grad phi_i(x_q)
                 fe_values.shape_grad(j, q) * // grad phi_j(x_q)
                 fe_values.JxW(q));           // dx

          for (const unsigned int i : fe_values.dof_indices())
            cell_rhs(i) += (fe_values.shape_value(i, q) *            // phi_i(x_q)
                            fe_values.quadrature_point(q).norm() * // f(x_q)=||x_q||
                            fe_values.JxW(q));                     // dx
        }
```

Part 2:

Solid mechanics in deal.II

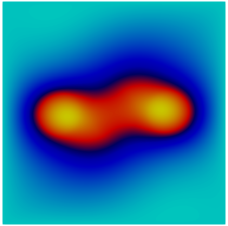
Further examples

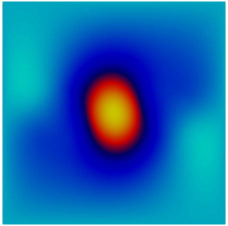
Many tutorials and code-gallery programs give good starting points for solid mechanics:



Further examples (cont.)

Tutorial: step-8

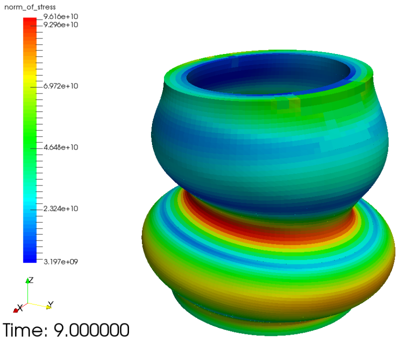




- linear elasticity
- dimension-independent
- Hooke's law
- parallelization in step-17 with PETSc

Further examples (cont.)

Tutorial: step-18

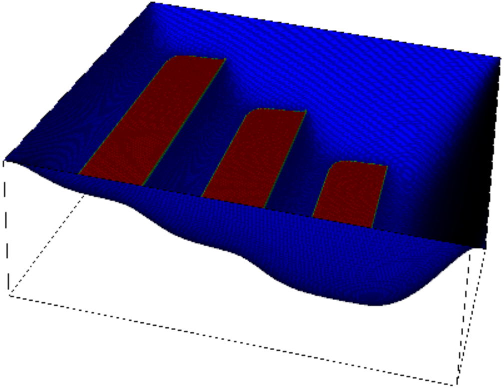


- quasistatic but time-dependent elasticity problem for large deformations with a Lagrangian mesh-movement approach
- buckling

Warning: The model considered here has little to do with reality!

Further examples (cont.)

Tutorial: step-41

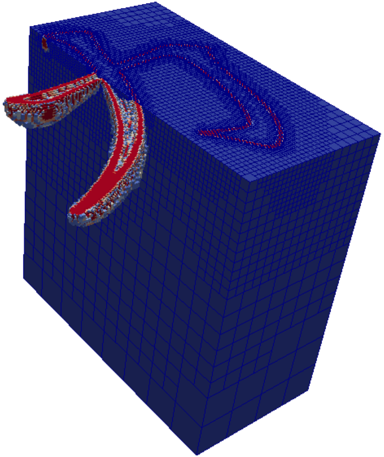


- elastic clamped membrane is deflected by gravity force, but is also constrained by an obstacle

  ... *a.k.a. obstacle problem*

Further examples (cont.)

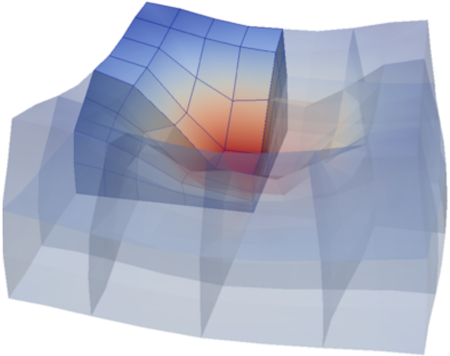
Tutorial: step-42



- deformation by rigid obstacle (contact problem)
- elasto-plastic material behavior with isotropic hardening

Further examples (cont.)

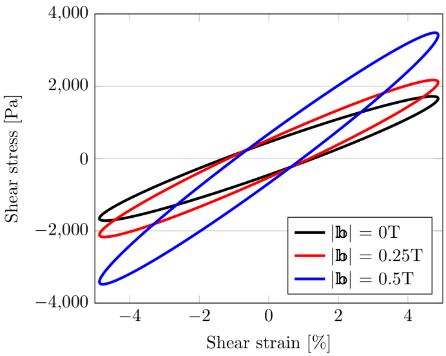
Tutorial: step-44



- three-field formulation
- fully nonlinear (geometrical and material) response of an isotropic continuum body
- quasi-incompressible neo-Hookean
- locking-free

Further examples (cont.)

Tutorial: step-71 (WIP)[1]

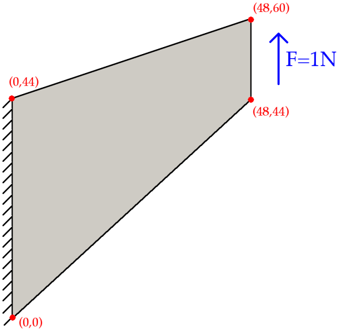


- automatic differentiation
- magneto-elastic constitutive law
- magneto-viscoelastic constitutive law

[1] https://github.com/dealii/dealii/pull/10392

Further examples (cont.)

Tutorial: step-73 (WIP)[2]



- automatic and symbolic differentiation
- finite-strain elasticity
- Cook's membrane

[2] https://github.com/dealii/dealii/pull/10394

Further examples (cont.)

Code gallery:

- elastoplastic torsion
- goal-oriented mesh adaptivity in elastoplasticity problems
- linear elastic active skeletal muscle model
- nonlinear poro-viscoelasticity
- quasi-static finite-strain compressible elasticity
- quasi-static finite-strain quasi-incompressible visco-elasticity
- linear elastoplasticity (WIP)[3] ▷ *history variables:* `CellDataStorage`

[3] `https://github.com/dealii/code-gallery/pull/62`

Part 3:

Theory

Strong form

- geometrically non-linear elasticity (reference configuration):

$$\mathrm{Div}(\underline{\boldsymbol{F}} \cdot \underline{\boldsymbol{S}}(\underline{\boldsymbol{E}})) + \underline{\hat{\boldsymbol{b}}}_0 = 0 \qquad \text{with } \rho_0 = 1$$

  with deformation gradient $\underline{\boldsymbol{F}}$, Green-Lagrange strain $\underline{\boldsymbol{E}}$, 2nd Piola-Kirchhoff stress $\underline{\boldsymbol{S}}$

- geometrically linear elasticity:

$$\mathrm{Div}(\underline{\sigma}) + \underline{\hat{\boldsymbol{b}}} = 0$$

  with $\underline{\sigma} = \underline{\boldsymbol{C}} : \underline{\varepsilon}$ and $\underline{\varepsilon} = \frac{1}{2}\left(\nabla \underline{\boldsymbol{u}} + \nabla \underline{\boldsymbol{u}}^T\right)$

Discrete weak form

Discrete weak form (geometrically linear elasticity):

$$\underline{\underline{\boldsymbol{K}}}\,\underline{\boldsymbol{u}} = \underline{\boldsymbol{F}} \quad \text{with} \quad \underline{\underline{\boldsymbol{K}}}_{ij}^{(e)} = \int\limits_{\Omega^{(e)}} \underline{\boldsymbol{B}}_i : \underline{\underline{\boldsymbol{C}}} : \underline{\boldsymbol{B}}_j \, \mathrm{d}\Omega \quad \text{and} \quad \underline{\boldsymbol{F}}_i^{(e)} = \int\limits_{\Gamma^{(e)}} \underline{\boldsymbol{N}}_i \cdot \underline{\boldsymbol{t}} \, \mathrm{d}\Gamma + \int\limits_{\Omega^{(e)}} \underline{\boldsymbol{N}}_i \cdot \underline{\boldsymbol{f}} \, \mathrm{d}\Omega$$

with $\underline{\boldsymbol{B}}_i = \frac{1}{2}\left(\nabla \underline{\boldsymbol{N}}_i + \nabla \underline{\boldsymbol{N}}_i^T\right)$.

Modifications compared to Poisson problem:

- $\underline{\boldsymbol{u}}$ is vectorial
- computation of $\underline{\underline{\boldsymbol{C}}}$ (Hooke's law)
- computation of $\underline{\boldsymbol{B}}$

more common notation:

$$\underline{\boldsymbol{B}} = \begin{bmatrix} \frac{\partial}{\partial x_1} & 0 \\ 0 & \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} \end{bmatrix} \left[\begin{array}{cc|c|cc} \underline{\boldsymbol{N}}_0 & 0 & \multirow{2}{*}{\dots} & \underline{\boldsymbol{N}}_k & 0 \\ 0 & \underline{\boldsymbol{N}}_0 & & 0 & \underline{\boldsymbol{N}}_k \end{array} \right]$$

... with index related to node

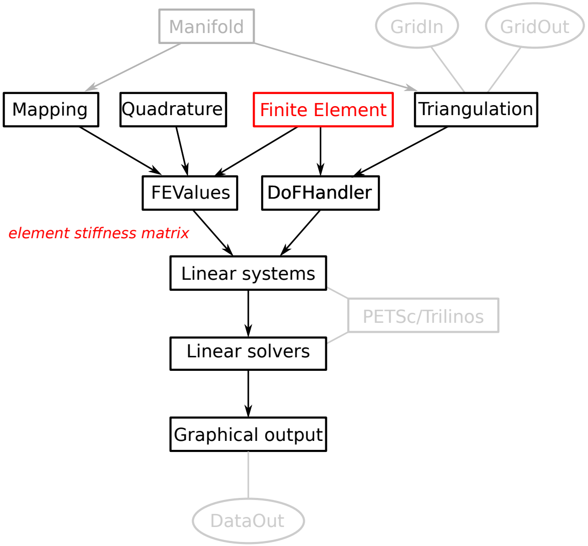
Vectorial finite element

Describe $\vec{u} \in \mathbb{R}^d$ as a **system of scalar Lagrange finite elements**:

$$\underbrace{[\mathcal{Q}_p^d, \ldots, \mathcal{Q}_p^d]}_{\times d}$$

in code:

```
FESystem<dim> fe(FE_Q<dim>(degree), dim);
```



Elastic stiffness tensor

Compute tensor $C_{ijkl} = \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) + \lambda\delta_{ij}\delta_{kl}$:

```
template <int dim>
SymmetricTensor<4, dim>
get_stress_strain_tensor(const double lambda, const double mu)
{
  SymmetricTensor<4, dim> tmp;
  for (unsigned int i = 0; i < dim; ++i)
    for (unsigned int j = 0; j < dim; ++j)
      for (unsigned int k = 0; k < dim; ++k)
        for (unsigned int l = 0; l < dim; ++l)
          tmp[i][j][k][l] = (((i == k) && (j == l) ? mu : 0.0) +
                             ((i == l) && (j == k) ? mu : 0.0) +
                             ((i == j) && (k == l) ? lambda : 0.0));
  return tmp;
}
```

... using SymmetricTensor<4, dim>

Strain tensor

Compute tensor $\underline{\boldsymbol{B}}_{iq}^{(e)} = \frac{1}{2}\left(\nabla \underline{\boldsymbol{N}}_i^{(e)}(\underline{\boldsymbol{x}}_q) + \nabla \underline{\boldsymbol{N}}_i^{(e)}(\underline{\boldsymbol{x}}_q)^T\right)$:

```
template <int dim>
inline SymmetricTensor<2, dim>
get_strain(const FEValues<dim> &fe_values,
           const unsigned int   shape_func,
           const unsigned int   q_point)
{
  SymmetricTensor<2, dim> tmp;

  for (unsigned int i = 0; i < dim; ++i)
    tmp[i][i] = fe_values.shape_grad_component(shape_func, q_point, i)[i];

  for (unsigned int i = 0; i < dim; ++i)
    for (unsigned int j = i + 1; j < dim; ++j)
      tmp[i][j] = (fe_values.shape_grad_component(shape_func, q_point, i)[j] +
                   fe_values.shape_grad_component(shape_func, q_point, j)[i]) /
                  2;

  return tmp;
}
```

... using SymmetricTensor$<$2, dim$>$

Strain tensor (cont.)

Compute tensor $\underline{\boldsymbol{B}}_{iq}^{(e)} = \frac{1}{2}\left(\nabla \underline{\boldsymbol{N}}_i^{(e)}(\underline{\boldsymbol{x}}_q) + \nabla \underline{\boldsymbol{N}}_i^{(e)}(\underline{\boldsymbol{x}}_q)^T\right)$:

```
template <int dim>
inline SymmetricTensor<2, dim>
get_strain(const FEValues<dim> &fe_values,
           const unsigned int   shape_func,
           const unsigned int   q_point)
{
  return fe_values[FEValuesExtractors::Vector()].symmetric_gradient(shape_func, q_point);
}
```

... using SymmetricTensor<2, dim>

On SymmetricTensor

The class SymmetricTensor allows to work in tensor notation with the performance of the Voigt notation[4] due to reduced memory consumption and specialized functions (e.g., double contraction).

E.g., internal representation of SymmetricTensor<2, 3>[5]:

$$\begin{bmatrix} \varepsilon_{00} & \varepsilon_{01} & \varepsilon_{02} \\ \varepsilon_{10} & \varepsilon_{11} & \varepsilon_{12} \\ \varepsilon_{20} & \varepsilon_{21} & \varepsilon_{22} \end{bmatrix} \quad \leftrightarrow \quad \begin{bmatrix} \varepsilon_{00} & \varepsilon_{11} & \varepsilon_{22} & \varepsilon_{01} & \varepsilon_{02} & \varepsilon_{12} \end{bmatrix}$$

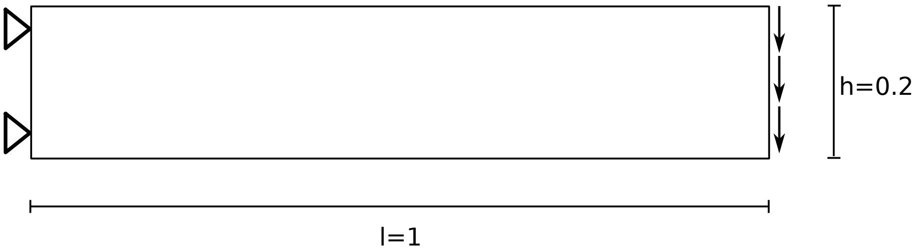
[4] https://www.dealii.org/developer/doxygen/deal.II/namespacePhysics_1_1Notation.html

[5] https://github.com/dealii/dealii/blob/8e208ae9dca8349c52b230514722463eb6fd51f4/include/deal.II/base/symmetric_tensor.h#L2419-L2425

Part 4:

Task

Example: beam



Example: beam (cont.)

```
const unsigned int dim = 2, degree = 1, n_refinements = 0;

// create mesh, select relevant FEM ingredients, and set up DoFHandler
Triangulation<dim> tria;
GridGenerator::subdivided_hyper_rectangle(
  tria, {10, 2}, Point<dim>(0, 0), Point<dim>(1, 0.2), true /*automatically set BIDs*/);
tria.refine_global(n_refinements);

FESystem<dim>          fe(FE_Q<dim>(degree), dim);
QGauss<dim>            quad(degree + 1);
QGauss<dim - 1>        face_quad(degree + 1);
MappingQGeneric<dim> mapping(1);

DoFHandler<dim> dof_handler(tria);
dof_handler.distribute_dofs(fe);

// Create constraint matrix
AffineConstraints<double> constraints;
VectorTools::interpolate_boundary_values(dof_handler,
                                         0 /*left face*/,
                                         Functions::ConstantFunction<dim>(std::vector<double>{0.0, 0.0}),
                                         constraints);
constraints.close();

// compute traction
Tensor<1, dim> traction; traction[0] = +0e9; traction[1] = -1e9;

// compute stress strain tensor
const auto stress_strain_tensor = get_stress_strain_tensor<dim>(9.695e10, 7.617e10);
```

Example: beam (cont.)

```
// initialize vectors and system matrix
Vector<double>       x(dof_handler.n_dofs()), b(dof_handler.n_dofs());
SparseMatrix<double> A;
SparsityPattern      sparsity_pattern;

DynamicSparsityPattern dsp(dof_handler.n_dofs());
DoFTools::make_sparsity_pattern(dof_handler, dsp);
sparsity_pattern.copy_from(dsp);
A.reinit(sparsity_pattern);

// assemble right-hand side and system matrix
FEValues<dim> fe_values(mapping, fe, quad, update_gradients | update_JxW_values);

FEFaceValues<dim> fe_face_values(mapping, fe, face_quad, update_values | update_JxW_values);

FullMatrix<double>                   cell_matrix;
Vector<double>                       cell_rhs;
std::vector<types::global_dof_index> local_dof_indices;
```

Example: beam (cont.)

```
// loop over all cells
for (const auto &cell : dof_handler.active_cell_iterators())
  {
    if (cell->is_locally_owned() == false)
      continue;

    fe_values.reinit(cell);

    const unsigned int dofs_per_cell = cell->get_fe().dofs_per_cell;
    cell_matrix.reinit(dofs_per_cell, dofs_per_cell);
    cell_rhs.reinit(dofs_per_cell);

    // loop over cell dofs
    for (unsigned int i = 0; i < dofs_per_cell; ++i)
      for (unsigned int j = 0; j < dofs_per_cell; ++j)
        for (unsigned int q = 0; q < fe_values.n_quadrature_points; ++q)
          {
            const auto eps_phi_i = get_strain(fe_values, i, q);
            const auto eps_phi_j = get_strain(fe_values, j, q);

            cell_matrix(i, j) += (eps_phi_i * stress_strain_tensor * eps_phi_j ) * fe_values.JxW(q);
          }
```

$$\int_{\Omega^{(e)}} \underline{\boldsymbol{B}}_i^T : \underline{\underline{\boldsymbol{C}}} : \underline{\boldsymbol{B}}_j \, \mathrm{d}\Omega \underline{\boldsymbol{u}}$$

Example: beam (cont.)

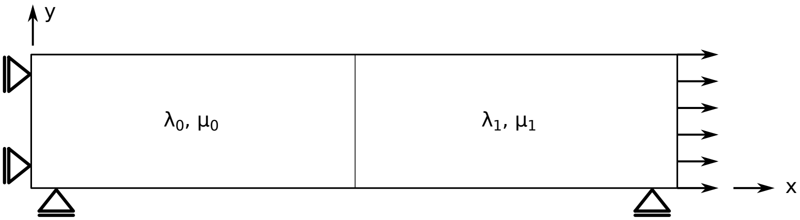
```
      // loop over all cell faces and their dofs
      for (const auto &face : cell->face_iterators())
        {
          // we only want to apply NBC on the right face
          if (!face->at_boundary() || face->boundary_id() != 1)
            continue;

          fe_face_values.reinit(cell, face);

          for (unsigned int q = 0; q < fe_face_values.n_quadrature_points; ++q)
            for (unsigned int i = 0; i < dofs_per_cell; ++i)
              cell_rhs(i) += fe_face_values.shape_value(i, q) *
                             traction[fe.system_to_component_index(i).first] *
                             fe_face_values.JxW(q);
        }

      local_dof_indices.resize(cell->get_fe().dofs_per_cell);
      cell->get_dof_indices(local_dof_indices);

      constraints.distribute_local_to_global(
        cell_matrix, cell_rhs, local_dof_indices, A, b);
    }
```

$$\int_{\Gamma^{(e)}} \underline{\boldsymbol{N}}_i^T \cdot \underline{\boldsymbol{t}} \, \mathrm{d}\Gamma$$

Example: beam (cont.)

```
// solve linear equation system
ReductionControl         reduction_control;
SolverCG<Vector<double>> solver(reduction_control);
solver.solve(A, x, b, PreconditionIdentity());

printf("Solved in %d iterations.\n", reduction_control.last_step());

constraints.distribute(x);

// output results
DataOut<dim> data_out;
data_out.attach_dof_handler(dof_handler);
x.update_ghost_values();
data_out.add_data_vector(dof_handler, x, "solution",
  std::vector<DataComponentInterpretation::DataComponentInterpretation>(
    dim, DataComponentInterpretation::component_is_part_of_vector));
data_out.build_patches(mapping, degree + 1);

std::ofstream output("solution.vtu");
data_out.write_vtu(output);
```

Tasks



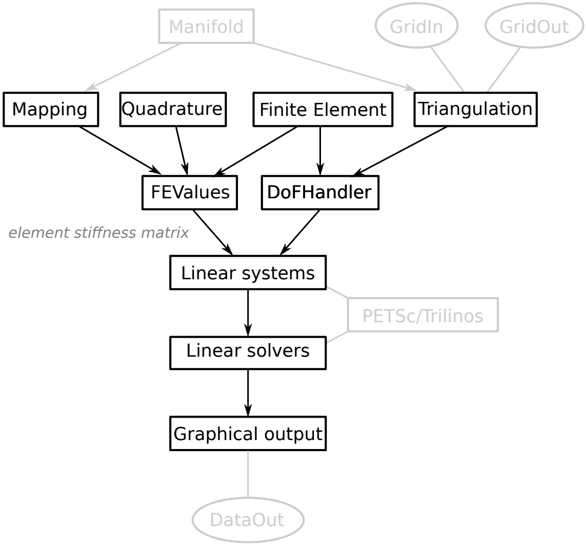
Extend “task-3a-empty.cc” (beam) to simulate a torsion rod:

- symmetric boundary condition on the left and bottom face
- force in x-direction on the right face
- vary material parameters of the rod (left vs. right) - see also Task 1b

Part 5:

Conclusions

Conclusions



general discussion:

- flexibility vs. usability (GUI)
- mathematical vs. physics software
- deal.II as a FEM toolbox?

further features:

- parallelization + AMR
- particles
- interfaces to many libraries

Thanks to Daniel & Ingo!