deal.II Workshop @ hereon/TUHH

Day 2: Poisson problem

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

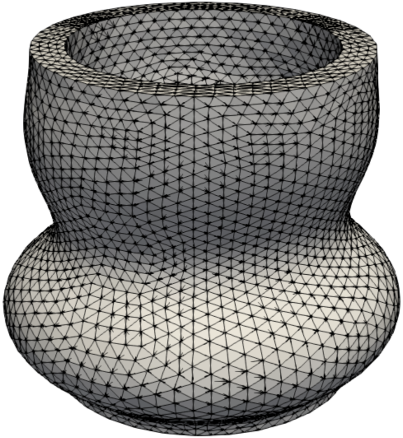
[1] deal.II developer

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

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

April 20, 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
- Wednesday: solid mechanics, particles

Part 1:

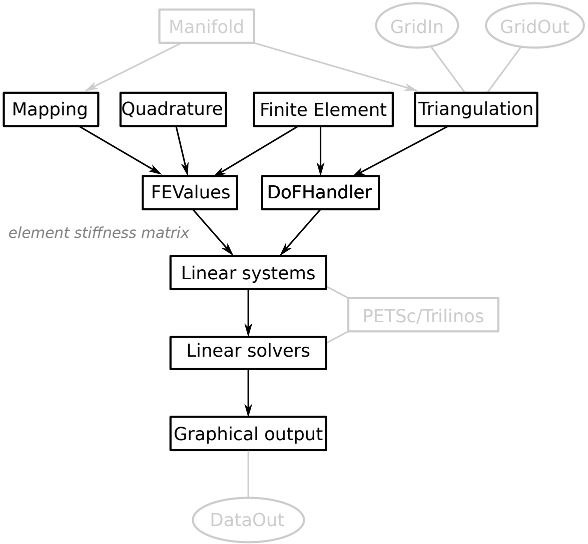
Wrap up of day 1

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 1a: solution

```
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_in.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/numerics/data_out.h>
#include <fstream>

using namespace dealii;

const int dim = 2;

int
main()
{
  Triangulation<dim> tria;

  // read mesh with GridIn
  GridIn<dim> grid_in(tria);
  grid_in.read("beam.msh");

  // write mesh with GridOut in VTK format
  std::ofstream output_1("task-1a-grid.vtk");
  GridOut grid_out;
  grid_out.write_vtk(tria, output_1);
  }
}
```

Task 1b: solution

```
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_in.h>
#include <deal.II/grid/grid_out.h>
#include <fstream>

using namespace dealii;
const int dim = 2;

int
main()
{
  Triangulation<dim> tria;

  // ...
  for (const auto &cell : tria.active_cell_iterators()) {
      std::cout << cell->center() << std::endl;

      cell->set_material_id(cell->center()[0] > 2.5);

      for (const auto &face : cell->face_iterators())
        if (face->at_boundary()) {
            if (face->center()[0] == 0.0) face->set_boundary_id(0);
            else if (face->center()[0] == 5.0) face->set_boundary_id(1);
            else if (face->center()[1] == 0.0) face->set_boundary_id(2);
            else if (face->center()[1] == 1.0) face->set_boundary_id(3);
          }
    }
  // ...
}
```

Part 2:

Solving the Poisson problem with deal.II

Poisson problem

Strong form of the Poisson problem:

$$
\begin{aligned}
-\nabla \cdot \nabla u &= f && \text{in } \Omega = (0,1) \times (0,1),\\
u &= h && \text{on } \Gamma_D = \{x = 0, y \in (0,1)\},\\
\nabla u(x,y) \cdot \underline{\boldsymbol{n}} &= g && \text{on } \Gamma_N = \{x = 1, y \in (0,1)\},\\
\nabla u(x,y) \cdot \underline{\boldsymbol{n}} &= 0 && \text{else.}
\end{aligned}
$$

Steps:

a. definition of the function spaces
b. derivation of the weak form
c. spatial discretization + computation of the element stiffness matrix
d. assembly and set-up of the linear equation system

Poisson problem (cont.)

Solve:

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

with:

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

requires:

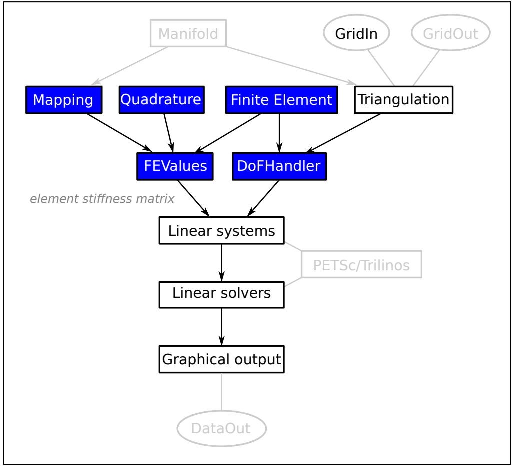
- mesh handling ✓
- finite elements, quadrature rules, mapping rules
- assembly procedure
- linear solver

Finite element, quadrature, mapping

Example configuration:

- finite element (`FE_Q`)
- quadrature (`QGauss`)
- mapping (`MappingQ1`)

```
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/mapping_q.h>
#include <deal.II/base/quadrature_lib.h>

int main()
{
  using namespace dealii;

  MappingQ1<2> mapping;
  FE_Q<2>      fe  (degree);
  QGauss<2>    quad (n_q_points_1D);
}
```



DoFHandler

- responsible for degrees of freedom
- initialization: Triangulation + FiniteElement
- loop over cells like in the case of Triangulation

see: `DoFCellAccessor`

```
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/grid/tria.h>

int main()
{
  using namespace dealii;

  Triangulation<2> tria;
  // ...

  FE_Q<2> fe (/**/);

  DoFHandler<2> dofs(tria);
  dofs.distribute_dofs(fe);

  for(auto & cell : dofs.active_cell_iterators())
  {
    // ...
  }
}
```

Constraints

The `AffineConstraint` class can be used to prescribe relationships of DoFs:

$$x_i = \sum_j a_{ij} x_j + b_i$$

... constraints are considered during assembly!

Following utility function can be used:

- `VectorTools::interpolate_boundary_values()` $\rightarrow$ DBC
- `DoFTools::make_periodicity_constraints()` $\rightarrow$ PBC
- `DoFTools::make_hanging_node_constraints()` $\rightarrow$ AMR

Note: can be used for multi-point constraints (MPC)

FEValues

motivation:

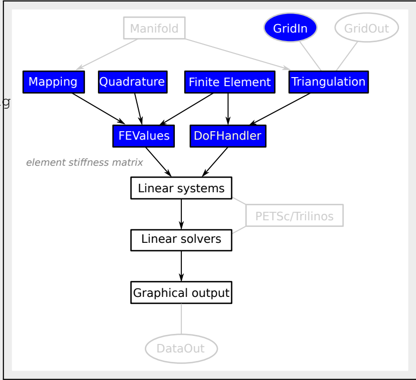
$$\mathbf{K}_{ij}^{(e)} = \sum_q (\nabla N_{iq}, \nabla N_{jq}) \cdot |J_q| \times w_q$$

- `FEValues` provides information at the cell quadrature points
- `UpdateFlags` determines what is needed
  - `update_values` $\rightarrow \underline{\underline{\boldsymbol{N}}}$
  - `update_gradients` $\rightarrow \nabla\underline{\underline{\boldsymbol{N}}}$
- for faces: `FEFaceValues`

```
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/grid/tria.h>

int main()
{
  using namespace dealii;

  Triangulation<2> tria;
  // ...

  MappingQ1<2> mapping;
  FE_Q<2> fe (/**/);
  QGauss quad (/**/);

  DoFHandler<2> dofs(tria);

  FEValues eval(mapping, fe, quad,
    update_values | update_quadrature_points);

  for(auto & cell : dofs.active_cell_iterators())
  {
    fe_values.reinit(cell);

    cout << eval.shape_value(0, 0) << endl;
    cout << eval.quadrature_point (0) << endl;
  }
}
```

Example

```
const unsigned int dim = 2, degree = 3, n_global_refinements = 3;

Triangulation<dim> tria; GridGenerator::hyper_cube(tria, 0, 1, true);
tria.refine_global(n_global_refinements);

FE_Q<dim> fe(degree); QGauss<dim> quad(degree + 1); MappingQ1<dim> mapping

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

// deal with boundary conditions
AffineConstraints<double> constraints;
VectorTools::interpolate_boundary_values(
  mapping, dof_handler, 0, Functions::ZeroFunction<dim>(), constraints);
constraints.close();

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

FEValues<dim> fe_values(mapping, fe, quad, update_default /*TODO*/);
```

$$\sum_q (\nabla N_{iq}, \nabla N_{jq}) \cdot |J_q| \times w_q, \qquad \sum_q (N_{iq}, f) \cdot |J_q| \times w_q$$

Example (cont.)

```
// loop over all cells
for (const auto &cell : dof_handler.active_cell_iterators())
  {
    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 (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) += 0.0; // TODO

        for (const unsigned int i : fe_values.dof_indices())
          cell_rhs(i) += 0.0; // TODO
      }

    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);
  }
```

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

$$\sum_q (N_{iq}, f) \cdot |J_q| \times w_q \rightarrow \mathbf{f}_{i}^{(e)}$$

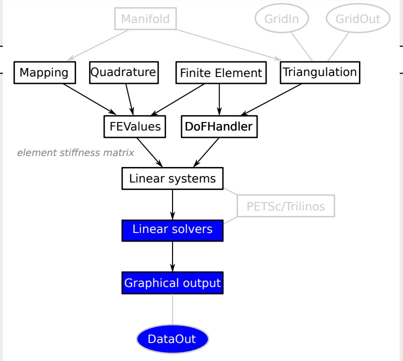
$$\underset{e}{\mathsf{A}}\, \mathbf{K}^{(e)}, \underset{e}{\mathsf{A}}\, \mathbf{f}^{(e)}$$

Example (cont.)

```
// solve linear equation system
ReductionControl         reduction_control(100, 1e-10, 1e-4);
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);
```

$$\mathbf{K}\mathbf{x} = \mathbf{f} \rightarrow \mathbf{x} = \mathbf{K}^{-1}\mathbf{f}$$

```
// output results
DataOutBase::VtkFlags flags;
flags.write_higher_order_cells = true;

DataOut<dim> data_out;
data_out.set_flags(flags);
data_out.attach_dof_handler(dof_handler);
data_out.add_data_vector(dof_handler, x, "solution");
data_out.build_patches(mapping, degree + 1);

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



Task 2

- task 2a) compute element-stiffness matrix and right-hand side for $g = h = 0$
- task 2b) set $g = 1$ ... *hint: take a look at* `Functions` *namespace*
- task 2c) set $h = 1$ ... *hint: use* `FEFaceValues`

Optional:

- make g and h depend on $\underline{\boldsymbol{x}}$
- play with solver and preconditioner
- implement mass-matrix operator (v, u) and Helmholtz operator $(v, u) + (\nabla v, \nabla u)$
- make the code work for triangles
  (hints: `FE_SimplexP`, `QGaussSimplex`, `MappingFE(FE_SimplexP(1))`)

Task 2: hint

mass matrix operator: $\mathbf{K}_{ij}^{(e)} = \sum_q (N_{iq}, N_{jq}) \cdot |J_q| \times w_q$

```
fe_values.reinit(cell);

for(const auto i : fe_values.dof_indices ())
  for(const auto j : fe_values.dof_indices ())
    for(const auto q : fe_values.quadrature_point_indices ())
      matrix(i,j) += fe_values.shape_value(i, q) * fe_values.shape_value(j, q) * fe_values.JxW(q);
```