Lab 03

Setup of a FEM problem. deal.II step-3, step-4.

Advanced Topic in Scientific Computing - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa

17 Oct 2024

Introduction

The `step-3` tutorial demonstrates a complete finite element simulation pipeline: from mesh generation and degree of freedom setup to solving the system and visualizing results. The key components introduced are the finite element basis functions, quadrature rules for numerical integration, and iterative solvers for linear systems.

Main function: `run()`

```
void Step3::run()
{
  make_grid();
  setup_system();
  assemble_system();
  solve();
  output_results();
}
```

The Poisson equation solver class

```
class Step3 {
public:
  Step3();

  void run();

private:
  void make_grid();
  void setup_system();
  void assemble_system();
  void solve();
  void output_results() const;

  Triangulation<2> triangulation;
  const FE_Q<2> fe;
  DoFHandler<2> dof_handler;

  SparsityPattern sparsity_pattern;
  SparseMatrix<double> system_matrix;

  Vector<double> solution;
  Vector<double> system_rhs;
};
```

Constructor

```
Step3::Step3()
  : fe(1), // Polynomial degree.
    dof_handler(triangulation)
{}
```

Creating the triangulation

```
void Step3::make_grid()
{
  GridGenerator::hyper_cube(triangulation, -1, 1);
  triangulation.refine_global(5);

  std::cout << "Number of active cells: "
            << triangulation.n_active_cells()
            << std::endl;
}
```

Setting up the system

```
void Step3::setup_system()
{
  dof_handler.distribute_dofs(fe);

  std::cout << "Number of degrees of freedom: "
            << dof_handler.n_dofs()
            << std::endl;

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

  system_matrix.reinit(sparsity_pattern);

  solution.reinit(dof_handler.n_dofs());
  system_rhs.reinit(dof_handler.n_dofs());
}
```

Assembling the linear system (1/2)

```
void Step3::assemble_system()
{
  QGauss<2> quadrature_formula(2);
  FEValues<2> fe_values(fe, quadrature_formula,
                        update_values   | update_gradients |
                        update_quadrature_points | update_JxW_values);

  const unsigned int dofs_per_cell = fe.dofs_per_cell;
  FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
  Vector<double> cell_rhs(dofs_per_cell);

  std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
```

Assembling the linear system (2/2)

```
  for (const auto &cell : dof_handler.active_cell_iterators())
  {
    cell_matrix = 0;
    cell_rhs = 0;

    fe_values.reinit(cell);

    for (unsigned int q_point=0; q_point < quadrature_formula.size(); ++q_point)
    {
      // Integration logic here.
    }

    cell->get_dof_indices(local_dof_indices);
    for (unsigned int i=0; i<dofs_per_cell; ++i)
      for (unsigned int j=0; j<dofs_per_cell; ++j)
        system_matrix.add(local_dof_indices[i],
                          local_dof_indices[j],
                          cell_matrix(i,j));

    for (unsigned int i=0; i<dofs_per_cell; ++i)
      system_rhs(local_dof_indices[i]) += cell_rhs(i);
  }
}
```

Integration logic

System matrix:

```
for (const unsigned int i : fe_values.dof_indices())
  for (const unsigned int j : fe_values.dof_indices())
    cell_matrix(i, j) +=
      (fe_values.shape_grad(i, q_index) * // grad phi_i(x_q)
       fe_values.shape_grad(j, q_index) * // grad phi_j(x_q)
       fe_values.JxW(q_index));           // dx
```

Right-hand side:

```
for (const unsigned int i : fe_values.dof_indices())
  cell_rhs(i) += (fe_values.shape_value(i, q_index) * // phi_i(x_q)
                  1. *                                // f(x_q)
                  fe_values.JxW(q_index));            // dx
```

Boundary conditions (1/2)

Boundary conditions are applied in the `step-3.cc` file using two key functions from deal.II:

1. **`VectorTools::interpolate_boundary_values`**: This function is used to enforce Dirichlet boundary conditions by specifying values at the boundary.

```
VectorTools::interpolate_boundary_values(dof_handler, types::boundary_id(0),
                                         BoundaryValuesFunction(),
                                         boundary_values);
```

   - Here, `types::boundary_id(0)` refers to the boundary where the condition is applied, and `BoundaryValuesFunction()` specifies the function that describes the boundary values. The result is stored in `boundary_values`.

Boundary conditions (2/2)

2. `MatrixTools::apply_boundary_values`: This function modifies the system matrix and right-hand side vector to apply the boundary conditions.

```
MatrixTools::apply_boundary_values(boundary_values, system_matrix, solution, system_rhs);
```

   - After determining the boundary values, this function adjusts the system matrix (`system_matrix`), solution vector (`solution`), and right-hand side vector (`system_rhs`) to account for the boundary conditions.

These boundary conditions ensure that the solution satisfies the prescribed values at the boundaries of the domain, typically for Dirichlet conditions where the solution is fixed at specific points.

Solving the linear system

```
void Step3::solve()
{
  SolverControl solver_control(1000, 1e-12);
  SolverCG<> solver(solver_control);

  solver.solve(system_matrix, solution, system_rhs,
               PreconditionIdentity());

  std::cout << solver_control.last_step()
            << " CG iterations needed to obtain convergence." << std::endl;
}
```

Output results

```
void Step3::output_results() const
{
  DataOut<2> data_out;

  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution, "solution");

  data_out.build_patches();

  std::ofstream output("solution.vtk");
  data_out.write_vtk(output);
}
```

Assignments

0. Learn how to visualize the solution using Paraview or VisIt.
1. Impose Dirichlet condition $u = 0$ only on the left boundary.
2. Impose Dirichlet conditions $u = 0$ on left, bottom, top boundaries and $u = 1$ on the right boundary.
3. Use HDF5 for output (useful for parallel processing).
4. Run the tutorial in 3D using dimension-independent programming, as reported in `step-4`.
5. (**Optional**) Convert to a simplex mesh (replace `FE_Q` with `FE_SimplexP` and `QGauss` with `QGaussSimplex`).
6. (**Bonus**) Use the method of manufactured solutions to construct a problem with exact solution. Compute the L^2 and H^1 errors and verify convergence estimates w.r.to the mesh size h.