Introduction to deal.II and matrix-free generalized eigenvalue problems

Denis Davydov

Philosophy and overview

What is deal.II - differential equations analysis library

- "A C++ library for solving PDEs using adaptive finite elements"
- "We provide the tools, you concentrate on your work!"

Primary aim

- Facilitate the rapid testing and development of FE codes
- Focus: Applied mathematics (as opposed to Engineering applications), agnostic to PDE

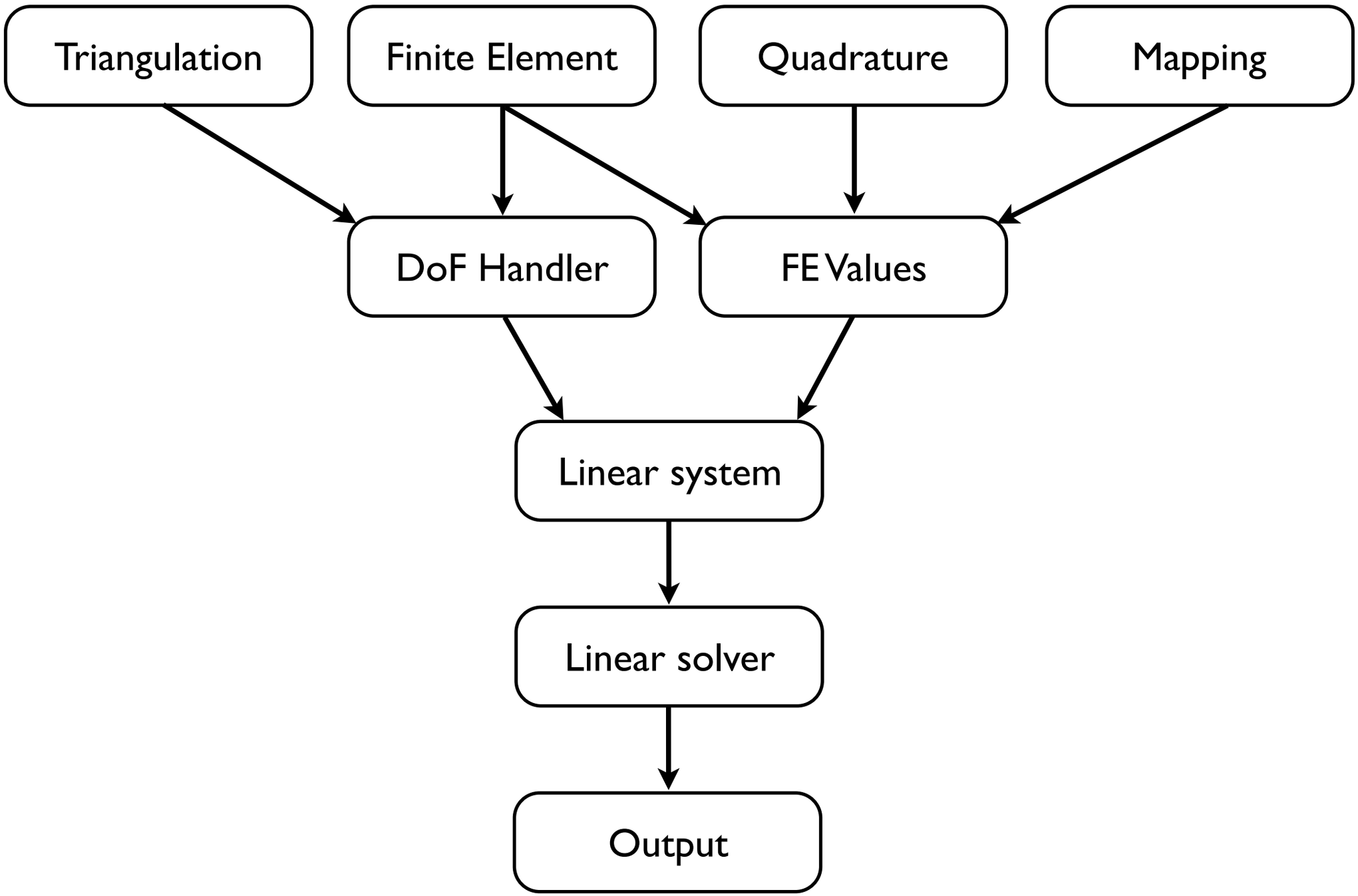
Capabilities

- Spatial independent programming (templates in C++)
- Manifolds (2d structures in 3d space)
- Adaptive mesh refinement (h-, hp-,…)
- Geometric multigrid and matrix-free methods (MPI+TBB+SIMD)
- Large number of continuous and discontinuous finite elements
- Interface to modern external libraries (Trilinos, PETSc, Sundials, Arpack, SLEPc, …)
- Parallelisation (domain decomposition and MPI, multithreading via TBB)
- Numerous assistive tool classes (Grid generator, integrators, run-time input, solution transfer,…)

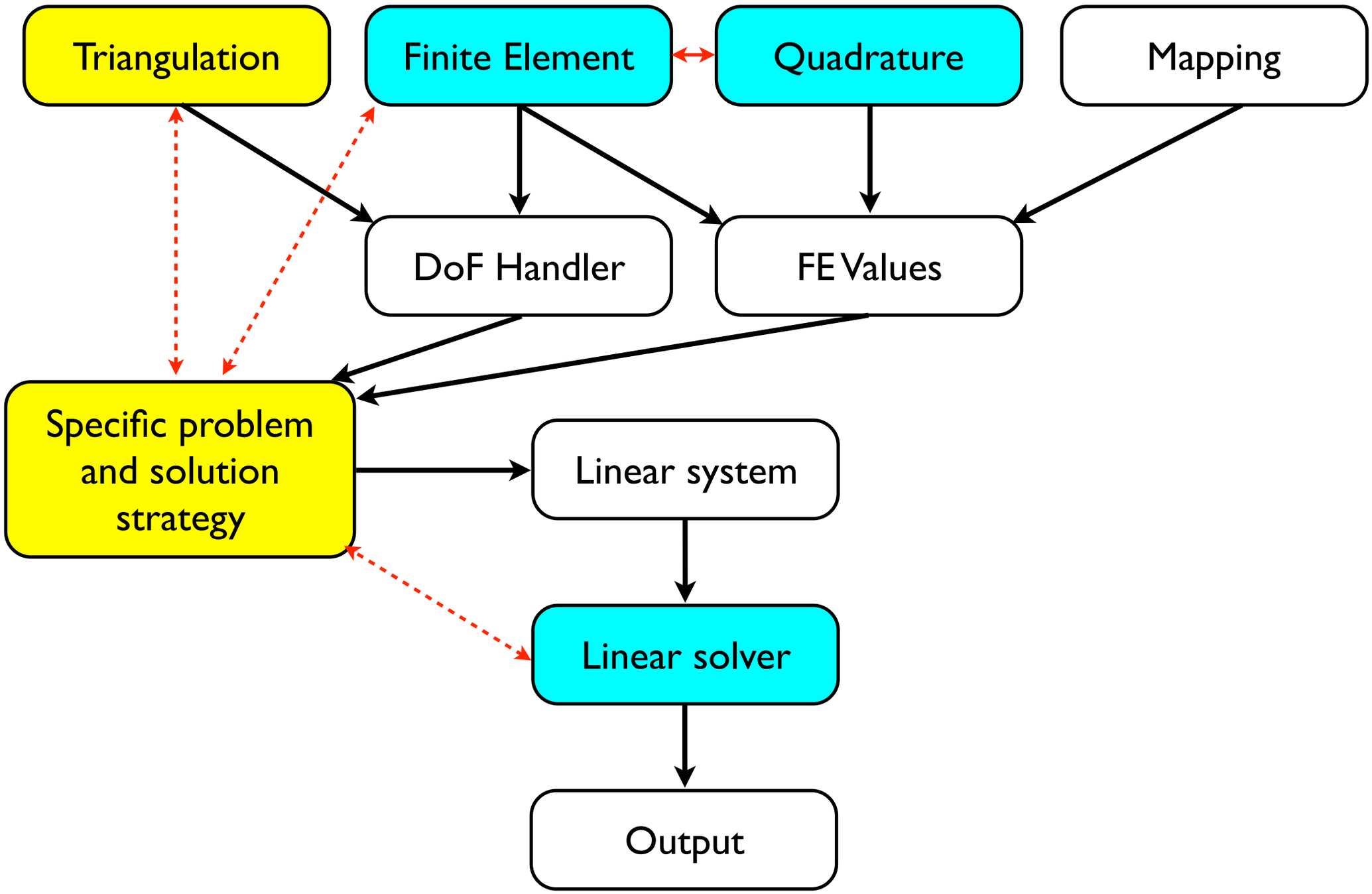
Bonus

- Extensively documented
- Well tested: 9k+ unit tests
- Example programs covering fields of fluid, solid, thermo and quantum mechanics

Structure of a FE problem



Structure of a FE problem



Structure of a FE problem

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)
- **DoFHandler**: enumerate a concrete FE basis using a triangulation so that we can represent solution as $u^h(\boldsymbol{x}) = \sum_i u_i N_i(\boldsymbol{x})$

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)
- **DoFHandler**: enumerate a concrete FE basis using a triangulation so that we can represent solution as $u^h(\boldsymbol{x}) = \sum_i u_i N_i(\boldsymbol{x})$
- **Mapping**: describes how to map shape functions, quadrature points, etc from the unit cell to each cell of a triangulation.

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)
- **DoFHandler**: enumerate a concrete FE basis using a triangulation so that we can represent solution as $u^h(\boldsymbol{x}) = \sum_i u_i N_i(\boldsymbol{x})$
- **Mapping**: describes how to map shape functions, quadrature points, etc from the unit cell to each cell of a triangulation.
- **FEValues**: given a FE, DoFHandler and Mapping, evaluate shape functions or their gradients at quadrature points when mapped to the real space.

Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)
- **DoFHandler**: enumerate a concrete FE basis using a triangulation so that we can represent solution as $u^h(\boldsymbol{x}) = \sum_i u_i N_i(\boldsymbol{x})$
- **Mapping**: describes how to map shape functions, quadrature points, etc from the unit cell to each cell of a triangulation.
- **FEValues**: given a FE, DoFHandler and Mapping, evaluate shape functions or their gradients at quadrature points when mapped to the real space.
- **Linear Algebra**: use bilinear weak form of the problem to assemble the system matrix and the right-hand side of the linear system. There is a zoo of classes to store and manage entries of (sparse) matrices and vectors.

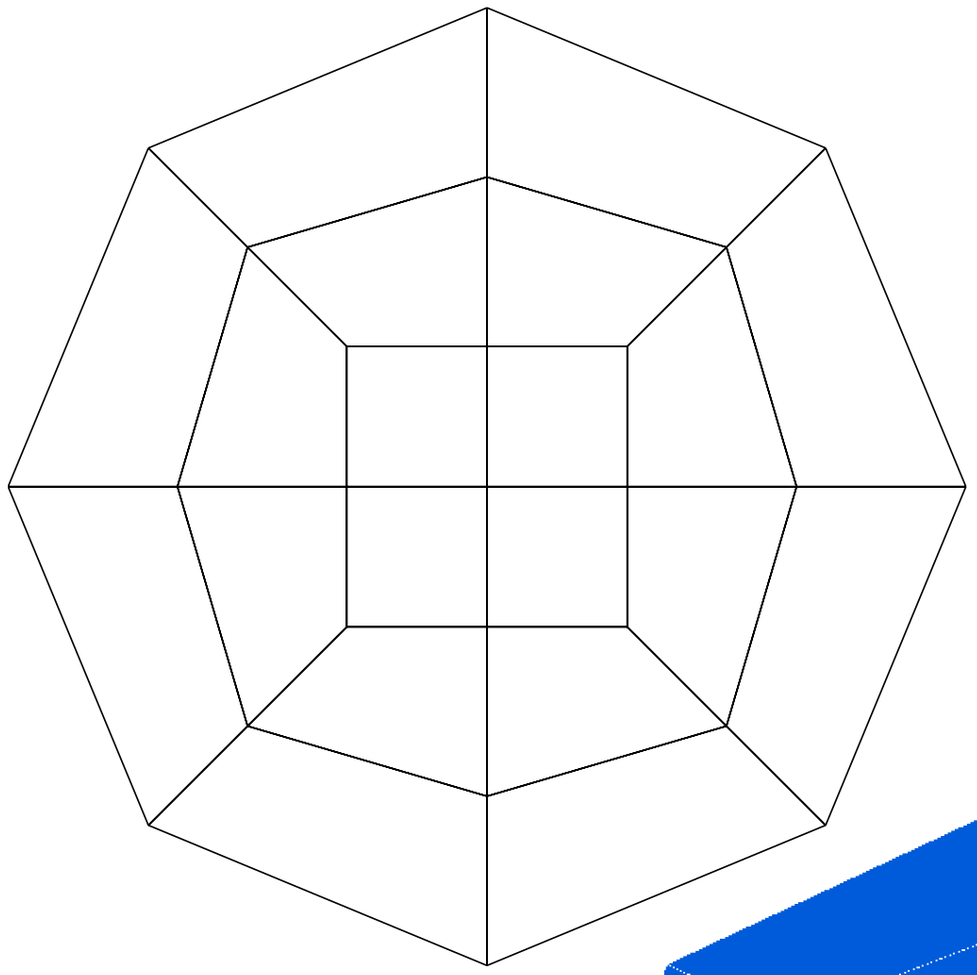
Structure of a FE problem

- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)
- **DoFHandler**: enumerate a concrete FE basis using a triangulation so that we can represent solution as $u^h(\boldsymbol{x}) = \sum_i u_i N_i(\boldsymbol{x})$
- **Mapping**: describes how to map shape functions, quadrature points, etc from the unit cell to each cell of a triangulation.
- **FEValues**: given a FE, DoFHandler and Mapping, evaluate shape functions or their gradients at quadrature points when mapped to the real space.
- **Linear Algebra**: use bilinear weak form of the problem to assemble the system matrix and the right-hand side of the linear system. There is a zoo of classes to store and manage entries of (sparse) matrices and vectors.
- **Linear Solvers**: variety of (iterative) solvers to solve the linear system.

Structure of a FE problem

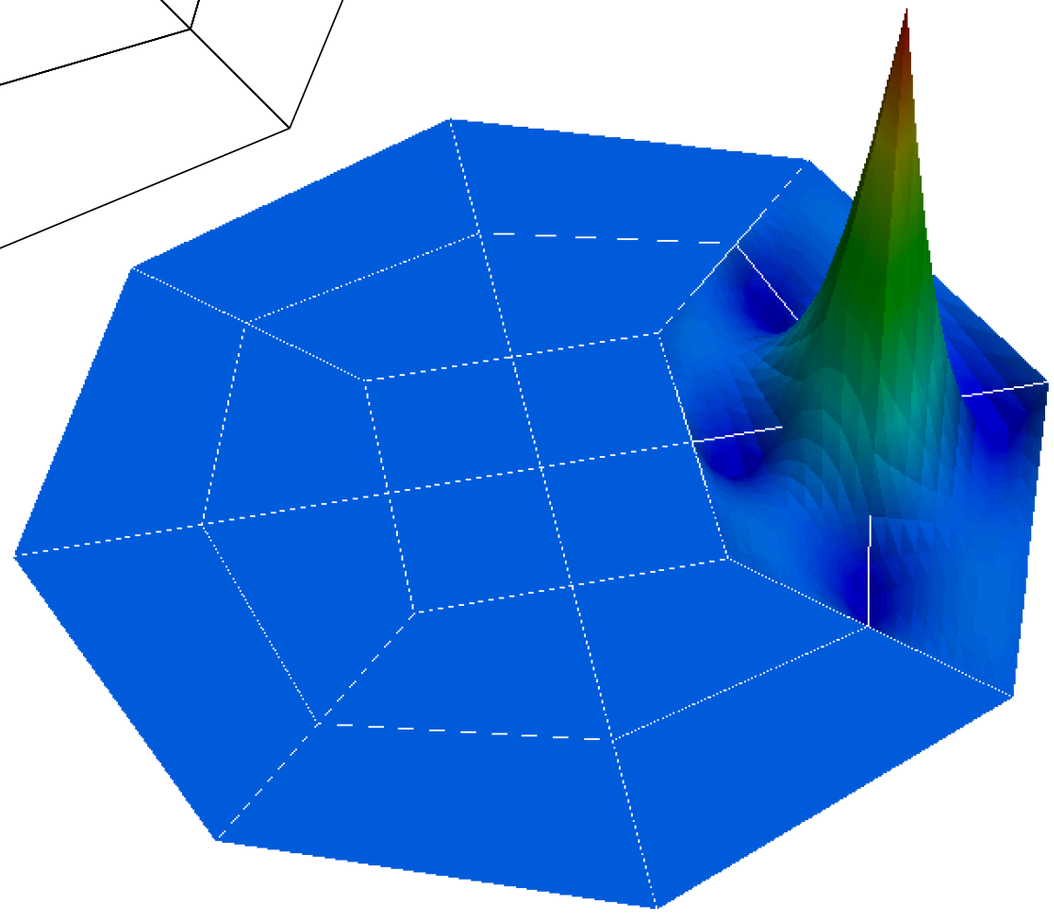
- **Triangulation**: collection of cells; stores geometric and topologic properties of mesh;
- **Finite Element**: describes the finite element space as defined on the unit cell [0,1]^dim. Allow evaluation of values/gradients of shape functions at points on the unit cell.
- **Quadrature**: location of quadrature points on the unit cell and their weights (used during numerical integration)
- **DoFHandler**: enumerate a concrete FE basis using a triangulation so that we can represent solution as $u^h(\boldsymbol{x}) = \sum_i u_i N_i(\boldsymbol{x})$
- **Mapping**: describes how to map shape functions, quadrature points, etc from the unit cell to each cell of a triangulation.
- **FEValues**: given a FE, DoFHandler and Mapping, evaluate shape functions or their gradients at quadrature points when mapped to the real space.
- **Linear Algebra**: use bilinear weak form of the problem to assemble the system matrix and the right-hand side of the linear system. There is a zoo of classes to store and manage entries of (sparse) matrices and vectors.
- **Linear Solvers**: variety of (iterative) solvers to solve the linear system.
- **Output**: post-processing and visualisation of the solution

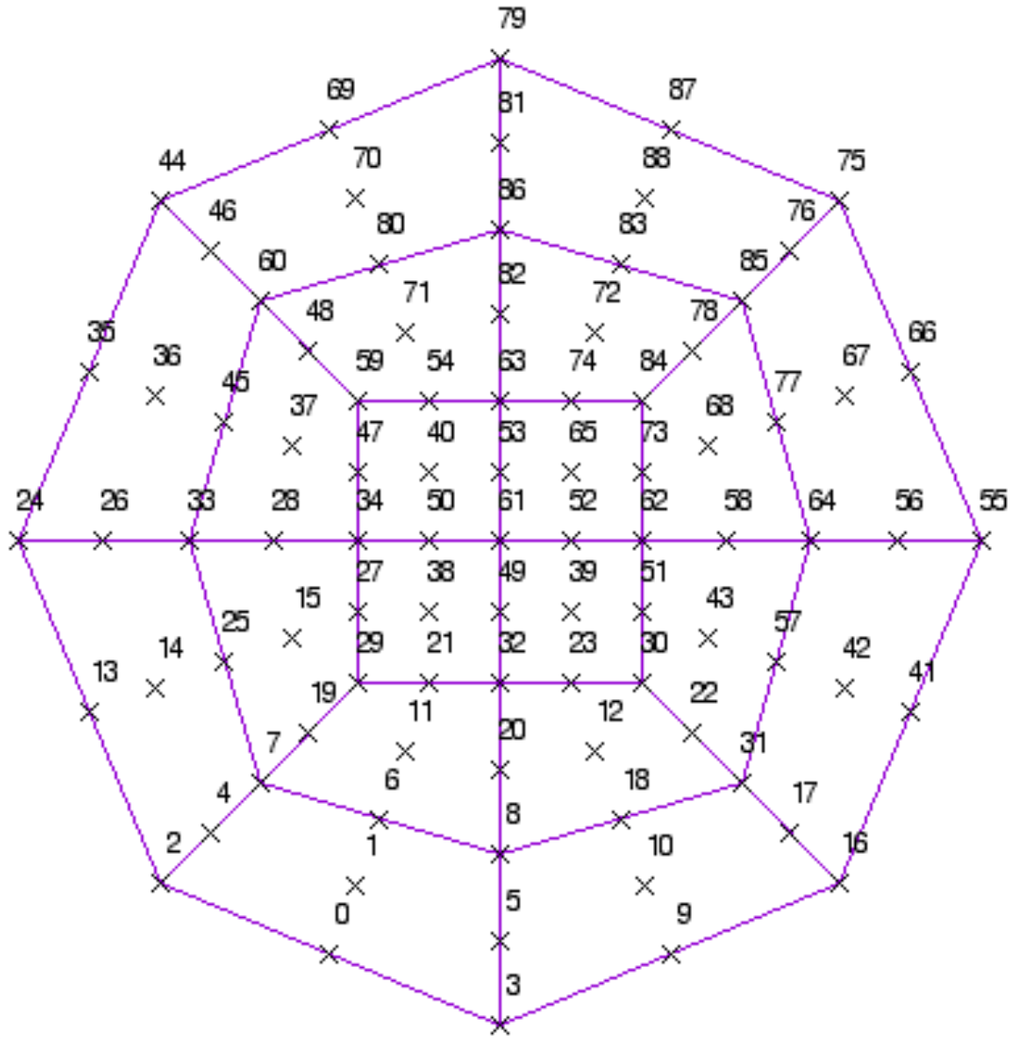
Triangulation + FE –> DoFHandler



+ quadratic FE =

(functional space
on a reference cell)





FE space

Laplace problem

POISSON'S EQUATION

Strong form

$$-\nabla \cdot c(\mathbf{x}) \nabla \varphi(\mathbf{x}) = f(\mathbf{x}) \quad \text{in} \quad \Omega$$

$$\varphi = \bar{\varphi} \quad \text{on} \quad \partial\Omega$$

Weak form

$$\int_\Omega c \nabla \delta\varphi \cdot \nabla \varphi - \int_{\delta\Omega} \delta\varphi \mathbf{n} \cdot \nabla \varphi = \int_\Omega \delta\varphi f \quad \forall \delta\varphi$$

Laplace problem

Poisson's equation

Strong form

$$-\nabla \cdot c(\mathbf{x}) \nabla \varphi(\mathbf{x}) = f(\mathbf{x}) \quad \text{in} \quad \Omega$$

$$\varphi = \bar{\varphi} \quad \text{on} \quad \partial\Omega$$

Weak form

$$\int_\Omega c \nabla \delta\varphi \cdot \nabla \varphi - \int_{\delta\Omega} \delta\varphi \mathbf{n} \cdot \nabla \varphi = \int_\Omega \delta\varphi f \quad \forall \delta\varphi$$

FE Approximation for discrete problem

Test function: $\delta\varphi^h(\mathbf{x}) = \sum_i N_i(\mathbf{x}) \, \delta\varphi_i$

Trial solution: $\varphi^h(\mathbf{x}) = \sum_i N_i(\mathbf{x}) \, \varphi_i$

Laplace problem

Poisson's equation

Strong form

$$-\nabla \cdot c(\mathbf{x}) \nabla \varphi(\mathbf{x}) = f(\mathbf{x}) \quad \text{in} \quad \Omega$$

$$\varphi = \bar{\varphi} \quad \text{on} \quad \partial\Omega$$

Weak form

$$\int_\Omega c \nabla \delta\varphi \cdot \nabla \varphi - \int_{\delta\Omega} \delta\varphi \mathbf{n} \cdot \nabla \varphi = \int_\Omega \delta\varphi f \quad \forall \delta\varphi$$

FE Approximation for discrete problem

Test function: $\delta\varphi^h(\mathbf{x}) = \sum_i N_i(\mathbf{x}) \, \delta\varphi_i$

Trial solution: $\varphi^h(\mathbf{x}) = \sum_i N_i(\mathbf{x}) \, \varphi_i$

Discrete linear system

System: $K_{ij} u_j = F_i$

Stiffness matrix:

$$K_{ij} = \sum_h \int_{\Omega^h} c \nabla N_i \cdot \nabla N_j$$

Force vector:

$$F_i = \sum_h \int_{\Omega^h} N_i f$$

Laplace problem

Poisson's equation

Strong form

$$-\nabla \cdot c(\mathbf{x}) \nabla \varphi(\mathbf{x}) = f(\mathbf{x}) \quad \text{in} \quad \Omega$$

$$\varphi = \bar{\varphi} \quad \text{on} \quad \partial\Omega$$

Weak form

$$\int_\Omega c \nabla \delta\varphi \cdot \nabla \varphi - \int_{\delta\Omega} \delta\varphi \mathbf{n} \cdot \nabla \varphi = \int_\Omega \delta\varphi f \quad \forall \delta\varphi$$

FE Approximation for discrete problem

Test function: $\delta\varphi^h(\mathbf{x}) = \sum_i N_i(\mathbf{x}) \, \delta\varphi_i$

Trial solution: $\varphi^h(\mathbf{x}) = \sum_i N_i(\mathbf{x}) \, \varphi_i$

Discrete linear system

System: $K_{ij} u_j = F_i$

Stiffness matrix:

$$K_{ij} = \sum_h \int_{\Omega^h} c \nabla N_i \cdot \nabla N_j$$

Force vector:

$$F_i = \sum_h \int_{\Omega^h} N_i f$$

Numerical integration

Element stiffness matrix:

$$K_{ij}^h = \sum_q c(\mathbf{x}) \nabla N_i\left(\xi_q^h\right) \cdot \nabla N_j\left(\xi_q^h\right) w_q^h$$

Element force function:

$$f_i^h = \sum_q N_i\left(\xi_q^h\right) \cdot f\left(\mathbf{x}_q^h\right) w_q^h$$

Step-6 tutorial program (main class):

```
template <int dim>
class Step6
{
public:
  Step6 ();
  ~Step6 ();

  void run ();

private:
  void make_mesh();
  void setup_system ();
  void assemble_system ();
  void solve ();
  void refine_grid ();
  void output_results (const unsigned int cycle) const;

  Triangulation<dim>           triangulation;
  const SphericalManifold<dim> boundary;

  FE_Q<dim>             fe;
  DoFHandler<dim>       dof_handler;

  ConstraintMatrix      constraints;

  SparsityPattern       sparsity_pattern;
  SparseMatrix<double>  system_matrix;

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

Step-6 tutorial program (main class):

Standard operations to initialise and solve problem

```
template <int dim>
class Step6
{
public:
  Step6 ();
  ~Step6 ();

  void run ();

private:
  void make_mesh();
  void setup_system ();
  void assemble_system ();
  void solve ();
  void refine_grid ();
  void output_results (const unsigned int cycle) const;

  Triangulation<dim>           triangulation;
  const SphericalManifold<dim> boundary;

  FE_Q<dim>             fe;
  DoFHandler<dim>       dof_handler;

  ConstraintMatrix      constraints;

  SparsityPattern       sparsity_pattern;
  SparseMatrix<double>  system_matrix;

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

Step-6 tutorial program (main class):

Describe domain and FE description of problem
Order of listing is important here.

```
template <int dim>
class Step6
{
public:
  Step6 ();
  ~Step6 ();

  void run ();

private:
  void make_mesh();
  void setup_system ();
  void assemble_system ();
  void solve ();
  void refine_grid ();
  void output_results (const unsigned int cycle) const;

  Triangulation<dim>           triangulation;
  const SphericalManifold<dim> boundary;

  FE_Q<dim>                    fe;
  DoFHandler<dim>              dof_handler;

  ConstraintMatrix             constraints;

  SparsityPattern              sparsity_pattern;
  SparseMatrix<double>         system_matrix;

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

Step-6 tutorial program (main class):

```
template <int dim>
class Step6
{
public:
  Step6 ();
  ~Step6 ();

  void run ();

private:
  void make_mesh();
  void setup_system ();
  void assemble_system ();
  void solve ();
  void refine_grid ();
  void output_results (const unsigned int cycle) const;

  Triangulation<dim>           triangulation;
  const SphericalManifold<dim> boundary;

  FE_Q<dim>                    fe;
  DoFHandler<dim>              dof_handler;

  ConstraintMatrix             constraints;

  SparsityPattern              sparsity_pattern;
  SparseMatrix<double>         system_matrix;

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

Describe linear system and constraints

Driver functions

```
int main ()
{
  Step6<2> laplace_problem_2d;
  laplace_problem_2d.run ();
  return 0;
}
```

```
template <int dim>
void Step6<dim>::run ()
{
  make_mesh();
  for (unsigned int cycle=0; cycle<8; ++cycle)
    {
      std::cout << "Cycle " << cycle << ':' << std::endl;

      if (cycle > 0)
        refine_grid ();

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

      setup_system ();

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

      assemble_system ();
      solve ();
      output_results (cycle);
    }
}
```

Driver functions

Main function creates an instance of the class
Problem solves itself!

```
int main ()
{
  Step6<2> laplace_problem_2d;
  laplace_problem_2d.run ();
  return 0;
}
```

```
template <int dim>
void Step6<dim>::run ()
{
  make_mesh();

  for (unsigned int cycle=0; cycle<8; ++cycle)
    {
      std::cout << "Cycle " << cycle << ':' << std::endl;

      if (cycle > 0)
        refine_grid ();

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

      setup_system ();

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

      assemble_system ();
      solve ();
      output_results (cycle);
    }
}
```

Driver functions

```
int main ()
{
  Step6<2> laplace_problem_2d;
  laplace_problem_2d.run ();
  return 0;
}
```

Order of calls to initialise and run problem
We'll see the same arrangement again...

```
template <int dim>
void Step6<dim>::run ()
{
  make_mesh();

  for (unsigned int cycle=0; cycle<8; ++cycle)
    {
      std::cout << "Cycle " << cycle << ':' << std::endl;

      if (cycle > 0)
        refine_grid ();

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

      setup_system ();

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

      assemble_system ();
      solve ();
      output_results (cycle);
    }
}
```

Constructor and mesh generation

```
template <int dim>
Step6<dim>::Step6 ()
  :
  fe (2),
  dof_handler (triangulation)
{}
```

```
template <int dim>
void
Step6<dim>::make_mesh ()
{
   GridGenerator::hyper_ball (triangulation);

   triangulation.set_all_manifold_ids_on_boundary(0);
   triangulation.set_manifold (0, boundary);

   triangulation.refine_global (1);
}
```

Constructor and mesh generation

Finite element type specified in main class
- Continuous Lagrange polynomial

Choose FE polynomial degree
- Quadratic

```
template <int dim>
Step6<dim>::Step6 ()
  :
  fe (2),
  dof_handler (triangulation)
{}
```

```
template <int dim>
void
Step6<dim>::make_mesh ()
{
   GridGenerator::hyper_ball (triangulation);

   triangulation.set_all_manifold_ids_on_boundary(0);
   triangulation.set_manifold (0, boundary);

   triangulation.refine_global (1);
}
```

Constructor and mesh generation

Associate the DoFHandler with a triangulation object
- The two are now linked until destruction of DoFH.
- One-way association (tria can be reused with other DoFHandlers)
- Doesn't need to know anything about the geometry / mesh at this point

```
template <int dim>
Step6<dim>::Step6 ()
  :
  fe (2),
  dof_handler (triangulation)
{}
```

```
template <int dim>
void
Step6<dim>::make_mesh ()
{
   GridGenerator::hyper_ball (triangulation);

   triangulation.set_all_manifold_ids_on_boundary(0);
   triangulation.set_manifold (0, boundary);

   triangulation.refine_global (1);
}
```

Constructor and mesh generation

```
template <int dim>
Step6<dim>::Step6 ()
  :
  fe (2),
  dof_handler (triangulation)
{}
```

Circle/ball around origin with the unit radius.
Spherical manifold is attached for refinement.

```
template <int dim>
void
Step6<dim>::make_mesh ()
{
    GridGenerator::hyper_ball (triangulation);

    triangulation.set_all_manifold_ids_on_boundary(0);
    triangulation.set_manifold (0, boundary);

    triangulation.refine_global (1);
}
```

setup_system: Data structure initialization

```
template <int dim>
void Step6<dim>::setup_system ()
{
  // Distribute DoFs
  dof_handler.distribute_dofs (fe);

  // Optimise DoF numbering
  DoFRenumbering::Cuthill_McKee (dof_handler);

  // Zero Dirichlet BC and hanging-nodes constraints
  constraints.clear ();
  DoFTools::make_hanging_node_constraints (dof_handler,
                                           constraints);
  VectorTools::interpolate_boundary_values (dof_handler,
                                            0,
                                            Functions::ZeroFunction<dim>(),
                                            constraints);
  constraints.close ();

  // Sparsity of the system matrix:
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler,
                                  dsp,
                                  constraints,
                                  /*keep_constrained_dofs = */ false);
  sparsity_pattern.copy_from(dsp);

  // Initialize matrix
  system_matrix.reinit (sparsity_pattern);

  // Initialize vectors
  solution.reinit (dof_handler.n_dofs());
  system_rhs.reinit (dof_handler.n_dofs());
}
```

setup_system: Data structure initialization

Describe distribution of DoFs
Result determined by
- Triangulation (vertex numbering, refinement)
- FE type

```
template <int dim>
void Step6<dim>::setup_system ()
{
  // Distribute DoFs
  dof_handler.distribute_dofs (fe);

  // Optimise DoF numbering
  DoFRenumbering::Cuthill_McKee (dof_handler);

  // Zero Dirichlet BC and hanging-nodes constraints
  constraints.clear ();
  DoFTools::make_hanging_node_constraints (dof_handler,
                                           constraints);
  VectorTools::interpolate_boundary_values (dof_handler,
                                            0,
                                            Functions::ZeroFunction<dim>(),
                                            constraints);
  constraints.close ();

  // Sparsity of the system matrix:
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler,
                                  dsp,
                                  constraints,
                                  /*keep_constrained_dofs = */ false);
  sparsity_pattern.copy_from(dsp);

  // Initialize matrix
  system_matrix.reinit (sparsity_pattern);

  // Initialize vectors
  solution.reinit (dof_handler.n_dofs());
  system_rhs.reinit (dof_handler.n_dofs());
}
```

setup_system: Data structure initialization

No optimisation of DoF numbering occurs by default
Changing sparse matrix bandwidth affects efficiency of
- LU decomposition
- Gauss-Seidel preconditioners

```
template <int dim>
void Step6<dim>::setup_system ()
{
  // Distribute DoFs
  dof_handler.distribute_dofs (fe);

  // Optimise DoF numbering
  DoFRenumbering::Cuthill_McKee (dof_handler);

  // Zero Dirichlet BC and hanging-nodes constraints
  constraints.clear ();
  DoFTools::make_hanging_node_constraints (dof_handler,
                                           constraints);
  VectorTools::interpolate_boundary_values (dof_handler,
                                            0,
                                            Functions::ZeroFunction<dim>(),
                                            constraints);
  constraints.close ();

  // Sparsity of the system matrix:
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler,
                                  dsp,
                                  constraints,
                                  /*keep_constrained_dofs = */ false);
  sparsity_pattern.copy_from(dsp);

  // Initialize matrix
  system_matrix.reinit (sparsity_pattern);

  // Initialize vectors
  solution.reinit (dof_handler.n_dofs());
  system_rhs.reinit (dof_handler.n_dofs());
}
```

setup_system: Data structure initialization

Temporary object to identify resulting sparsity pattern
- May implement different estimates for maximum number of coupling DoFs (memory efficiency)
- This object unsuitable for actual data storage

```
template <int dim>
void Step6<dim>::setup_system ()
{
  // Distribute DoFs
  dof_handler.distribute_dofs (fe);

  // Optimise DoF numbering
  DoFRenumbering::Cuthill_McKee (dof_handler);

  // Zero Dirichlet BC and hanging-nodes constraints
  constraints.clear ();
  DoFTools::make_hanging_node_constraints (dof_handler,
                                           constraints);
  VectorTools::interpolate_boundary_values (dof_handler,
                                            0,
                                            Functions::ZeroFunction<dim>(),
                                            constraints);
  constraints.close ();

  // Sparsity of the system matrix:
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler,
                                  dsp,
                                  constraints,
                                  /*keep_constrained_dofs = */ false);
  sparsity_pattern.copy_from(dsp);

  // Initialize matrix
  system_matrix.reinit (sparsity_pattern);

  // Initialize vectors
  solution.reinit (dof_handler.n_dofs());
  system_rhs.reinit (dof_handler.n_dofs());
}
```

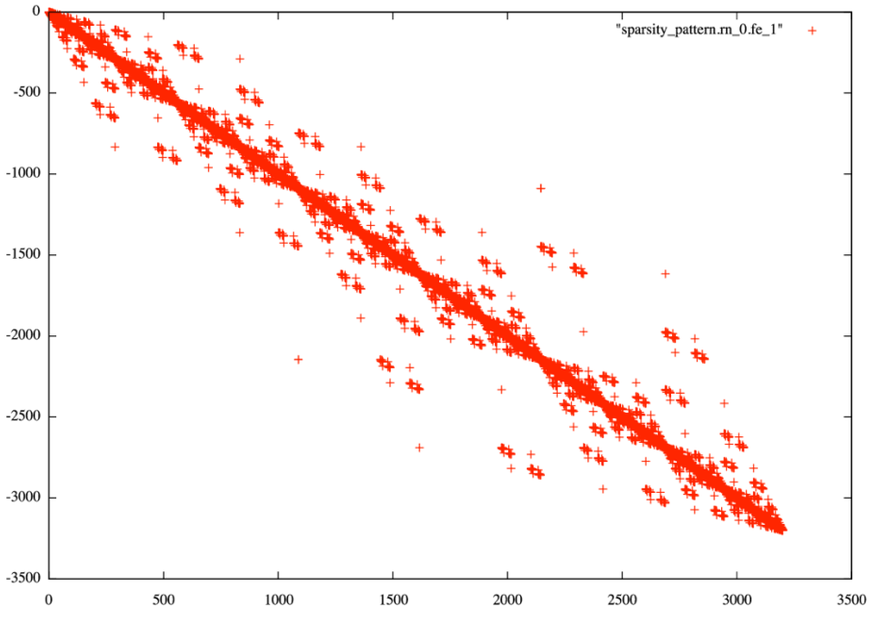
setup_system: Data structure initialization

```
template <int dim>
void Step6<dim>::setup_system ()
{
  // Distribute DoFs
  dof_handler.distribute_dofs (fe);

  // Optimise DoF numbering
  DoFRenumbering::Cuthill_McKee (dof_handler);

  // Zero Dirichlet BC and hanging-nodes constraints
  constraints.clear ();
  DoFTools::make_hanging_node_constraints (dof_handler,
                                           constraints);
  VectorTools::interpolate_boundary_values (dof_handler,
                                            0,
                                            Functions::ZeroFunction<dim>(),
                                            constraints);
  constraints.close ();

  // Sparsity of the system matrix:
  DynamicSparsityPattern dsp(dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler,
                                  dsp,
                                  constraints,
                                  /*keep_constrained_dofs = */ false);
  sparsity_pattern.copy_from(dsp);

  // Initialize matrix
  system_matrix.reinit (sparsity_pattern);

  // Initialize vectors
  solution.reinit (dof_handler.n_dofs());
  system_rhs.reinit (dof_handler.n_dofs());
}
```

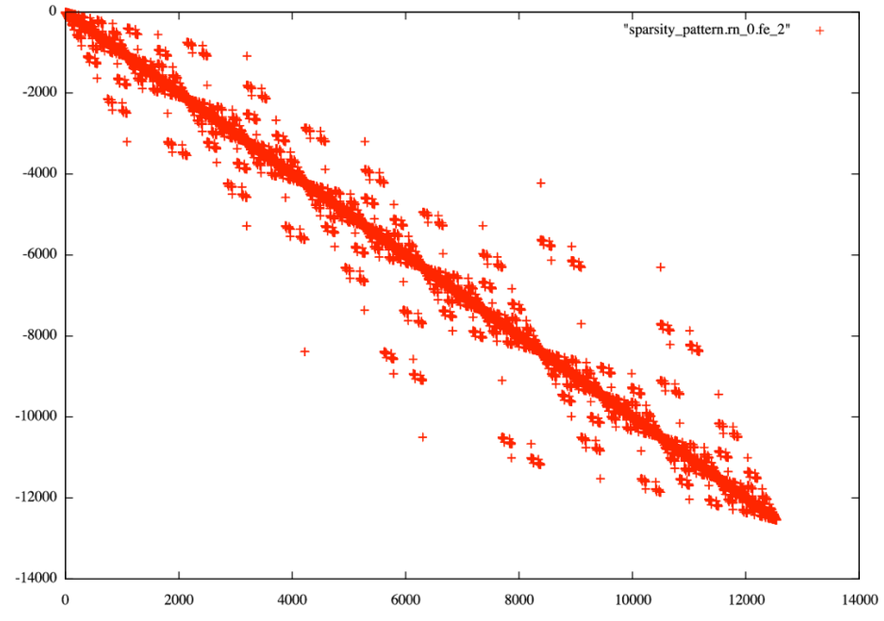
Initialise system matrix and vectors
- Embed sparsity pattern within matrix
- Set vector lengths
- Initialise values of both to zero

setup_system: Data structure initialization

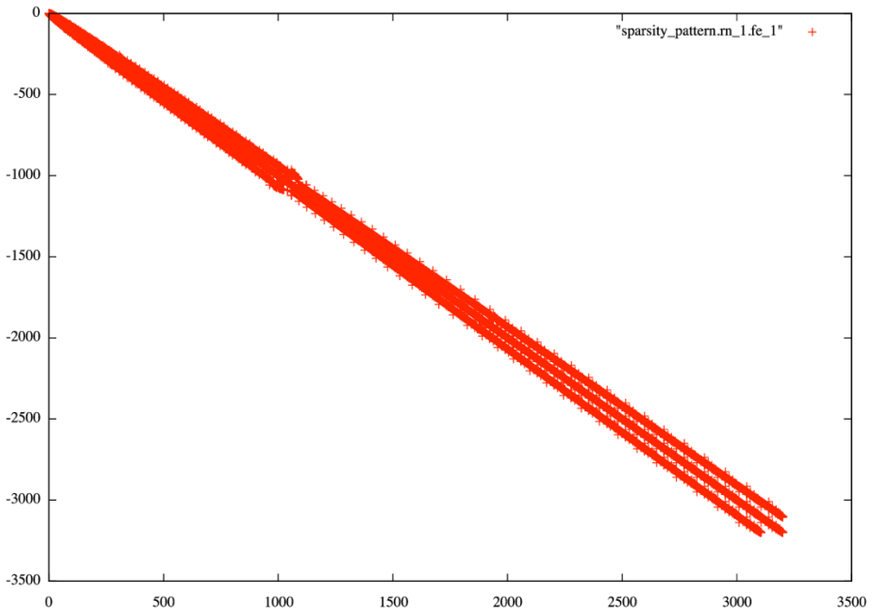
Order 1, No renumbering



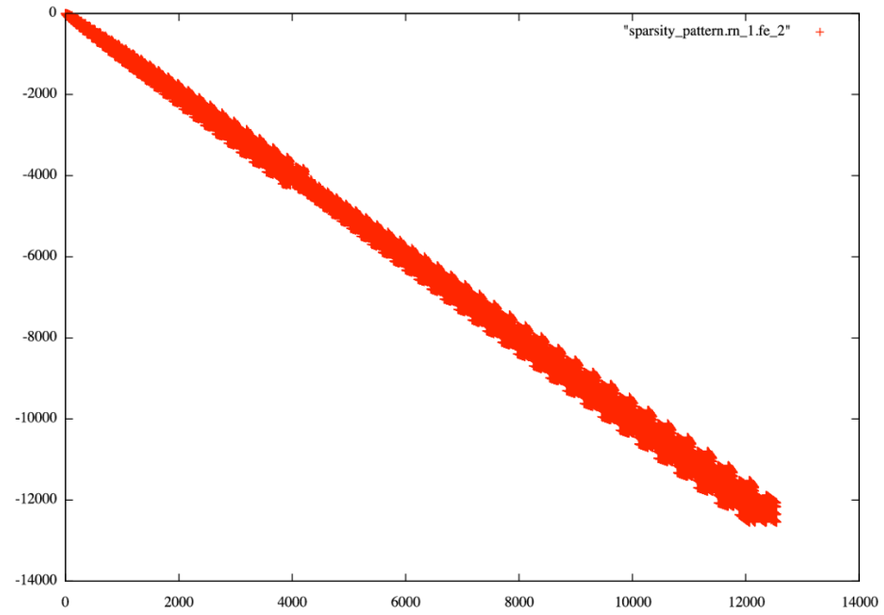
Order 2, No renumbering



Order 1, Cuthill-McKee renumbering



Order 2, Cuthill-McKee renumbering



assemble_system: Administration

```
template <int dim>
void Step6<dim>::assemble_system ()
{
  const QGauss<dim>  quadrature_formula(3);

  FEValues<dim> 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;
  const unsigned int   n_q_points    = quadrature_formula.size();

  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);
```

assemble_system: Administration

Numerical integration: Use Gauss quadrature

```
template <int dim>
void Step6<dim>::assemble_system ()
{
  const QGauss<dim>  quadrature_formula(3);

  FEValues<dim> 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;
  const unsigned int   n_q_points    = quadrature_formula.size();

  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);
```

assemble_system: Administration

Integration assistant
Links Finite Element type, quadrature formula and mapping to cell geometry
Need to tell it which data one wishes to utilise

```
template <int dim>
void Step6<dim>::assemble_system ()
{
  const QGauss<dim>  quadrature_formula(3);

  FEValues<dim> 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;
  const unsigned int   n_q_points    = quadrature_formula.size();

  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);
```

assemble_system: Administration

Use object function calls to dynamically return critical information

```
template <int dim>
void Step6<dim>::assemble_system ()
{
  const QGauss<dim>  quadrature_formula(3);

  FEValues<dim> 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;
  const unsigned int   n_q_points    = quadrature_formula.size();

  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);
```

assemble_system: Administration

```
template <int dim>
void Step6<dim>::assemble_system ()
{
  const QGauss<dim>  quadrature_formula(3);

  FEValues<dim> 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;
  const unsigned int   n_q_points    = quadrature_formula.size();

  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);
```

Create local (full) storage objects for element contributions to global matrix / vector

assemble_system: Administration

```
template <int dim>
void Step6<dim>::assemble_system ()
{
  const QGauss<dim>  quadrature_formula(3);

  FEValues<dim> 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;
  const unsigned int   n_q_points    = quadrature_formula.size();

  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);
```

Need to record which DoF's are associated with the cell

assemble_system: implementation

$$K_{ij}^h = \sum_q c(\mathbf{x}_q) \nabla N_i \left(\xi_q^h\right) \cdot \nabla N_j \left(\xi_q^h\right) w_q^h$$

$$f_i^h = \sum_q N_i \left(\xi_q^h\right) f\left(\mathbf{x}_q\right) w_q^h$$

```
typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
      {
        const double current_coefficient = coefficient<dim>
                                           (fe_values.quadrature_point (q_index));
        for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              cell_matrix(i,j) += (current_coefficient *
                                   fe_values.shape_grad(i,q_index) *
                                   fe_values.shape_grad(j,q_index) *
                                   fe_values.JxW(q_index));

            cell_rhs(i) += (fe_values.shape_value(i,q_index) *
                            1.0 *
                            fe_values.JxW(q_index));
          }
      }

    cell->get_dof_indices (local_dof_indices);
    constraints.distribute_local_to_global (cell_matrix,
                                            cell_rhs,
                                            local_dof_indices,
                                            system_matrix,
                                            system_rhs);
  }
```

assemble_system: implementation

Loop over all cells (using DoFHandler);
Reinitialise integration helper for the geometry of each cell (can be expensive);
Reset local matrix/vector to zero

$$K_{ij}^h = \sum_q c(\mathbf{x}_q) \nabla N_i \left(\xi_q^h\right) \cdot \nabla N_j \left(\xi_q^h\right) w_q^h$$

$$f_i^h = \sum_q N_i \left(\xi_q^h\right) f\left(\mathbf{x}_q\right) w_q^h$$

```
typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
      {
        const double current_coefficient = coefficient<dim>
                                           (fe_values.quadrature_point (q_index));
        for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              cell_matrix(i,j) += (current_coefficient *
                                   fe_values.shape_grad(i,q_index) *
                                   fe_values.shape_grad(j,q_index) *
                                   fe_values.JxW(q_index));

            cell_rhs(i) += (fe_values.shape_value(i,q_index) *
                            1.0 *
                            fe_values.JxW(q_index));
          }
      }

    cell->get_dof_indices (local_dof_indices);
    constraints.distribute_local_to_global (cell_matrix,
                                            cell_rhs,
                                            local_dof_indices,
                                            system_matrix,
                                            system_rhs);
  }
```

assemble_system: implementation

Loop over all quadrature points, evaluate heterogeneity coefficient, loop over all DoFs

$$K_{ij}^h = \sum_q c(\mathbf{x}_q) \nabla N_i \left(\xi_q^h\right) \cdot \nabla N_j \left(\xi_q^h\right) w_q^h$$

$$f_i^h = \sum_q N_i \left(\xi_q^h\right) f\left(\mathbf{x}_q\right) w_q^h$$

```
typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
      {
        const double current_coefficient = coefficient<dim>
                                           (fe_values.quadrature_point (q_index));
        for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              cell_matrix(i,j) += (current_coefficient *
                                   fe_values.shape_grad(i,q_index) *
                                   fe_values.shape_grad(j,q_index) *
                                   fe_values.JxW(q_index));

            cell_rhs(i) += (fe_values.shape_value(i,q_index) *
                            1.0 *
                            fe_values.JxW(q_index));
          }
      }

    cell->get_dof_indices (local_dof_indices);
    constraints.distribute_local_to_global (cell_matrix,
                                            cell_rhs,
                                            local_dof_indices,
                                            system_matrix,
                                            system_rhs);
  }
```

assemble_system: implementation

Stiffness contribution
- Retrieve shape function gradients at each QP for the i^{th} and j^{th} DoF (rank 1 tensor)
- Get integration weights and mapping Jacobian

$$K_{ij}^h = \sum_q c(\mathbf{x}_q) \nabla N_i \left(\xi_q^h\right) \cdot \nabla N_j \left(\xi_q^h\right) w_q^h$$

$$f_i^h = \sum_q N_i \left(\xi_q^h\right) f\left(\mathbf{x}_q\right) w_q^h$$

```
typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
      {
        const double current_coefficient = coefficient<dim>
                                           (fe_values.quadrature_point (q_index));
        for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              cell_matrix(i,j) += (current_coefficient *
                                   fe_values.shape_grad(i,q_index) *
                                   fe_values.shape_grad(j,q_index) *
                                   fe_values.JxW(q_index));

            cell_rhs(i) += (fe_values.shape_value(i,q_index) *
                            1.0 *
                            fe_values.JxW(q_index));
          }
      }

    cell->get_dof_indices (local_dof_indices);
    constraints.distribute_local_to_global (cell_matrix,
                                            cell_rhs,
                                            local_dof_indices,
                                            system_matrix,
                                            system_rhs);
  }
```

assemble_system: implementation

$$K_{ij}^h = \sum_q c(\mathbf{x}_q) \nabla N_i \left(\xi_q^h\right) \cdot \nabla N_j \left(\xi_q^h\right) w_q^h$$

$$f_i^h = \sum_q N_i \left(\xi_q^h\right) f\left(\mathbf{x}_q\right) w_q^h$$

RHS contribution
- Retrieve shape function value at each QP for the i^{th} DoF (scalar value)
- Get integration weights and mapping Jacobian

```
typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
      {
        const double current_coefficient = coefficient<dim>
                                           (fe_values.quadrature_point (q_index));
        for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              cell_matrix(i,j) += (current_coefficient *
                                   fe_values.shape_grad(i,q_index) *
                                   fe_values.shape_grad(j,q_index) *
                                   fe_values.JxW(q_index));

            cell_rhs(i) += (fe_values.shape_value(i,q_index) *
                            1.0 *
                            fe_values.JxW(q_index));
          }
      }

    cell->get_dof_indices (local_dof_indices);
    constraints.distribute_local_to_global (cell_matrix,
                                            cell_rhs,
                                            local_dof_indices,
                                            system_matrix,
                                            system_rhs);
  }
```

assemble_system: implementation

$$K_{ij}^h = \sum_q c(\mathbf{x}_q) \nabla N_i \left(\xi_q^h\right) \cdot \nabla N_j \left(\xi_q^h\right) w_q^h$$

$$f_i^h = \sum_q N_i \left(\xi_q^h\right) f\left(\mathbf{x}_q\right) w_q^h$$

```
typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
      {
        const double current_coefficient = coefficient<dim>
                                           (fe_values.quadrature_point (q_index));
        for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              cell_matrix(i,j) += (current_coefficient *
                                   fe_values.shape_grad(i,q_index) *
                                   fe_values.shape_grad(j,q_index) *
                                   fe_values.JxW(q_index));

            cell_rhs(i) += (fe_values.shape_value(i,q_index) *
                            1.0 *
                            fe_values.JxW(q_index));
          }
      }

    cell->get_dof_indices (local_dof_indices);
    constraints.distribute_local_to_global (cell_matrix,
                                            cell_rhs,
                                            local_dof_indices,
                                            system_matrix,
                                            system_rhs);
  }
```

Assembly procedure
- Distribute contribution to global matrix / vector and apply constraints while doing so.
- Achieved by knowing which local DoF is related to which global DoF though internal indexing

Solving the linear system

```
template <int dim>
void Step6<dim>::solve ()
{
  SolverControl      solver_control (1000, 1e-12);
  SolverCG<>         solver (solver_control);

  PreconditionSSOR<> preconditioner;
  preconditioner.initialize(system_matrix, 1.2);

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

  constraints.distribute (solution);
}
```

Solving the linear system

Object to dictate stopping criterion to conjugate gradient solver

```
template <int dim>
void Step6<dim>::solve ()
{
  SolverControl      solver_control (1000, 1e-12);
  SolverCG<>         solver (solver_control);

  PreconditionSSOR<> preconditioner;
  preconditioner.initialize(system_matrix, 1.2);

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

  constraints.distribute (solution);
}
```

Solving the linear system

Use SSOR preconditioner

```
template <int dim>
void Step6<dim>::solve ()
{
  SolverControl      solver_control (1000, 1e-12);
  SolverCG<>         solver (solver_control);

  PreconditionSSOR<> preconditioner;
  preconditioner.initialize(system_matrix, 1.2);

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

  constraints.distribute (solution);
}
```

Solving the linear system

```
template <int dim>
void Step6<dim>::solve ()
{
  SolverControl      solver_control (1000, 1e-12);
  SolverCG<>         solver (solver_control);

  PreconditionSSOR<> preconditioner;
  preconditioner.initialize(system_matrix, 1.2);

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

  constraints.distribute (solution);
}
```

Solve the system of equations and distribute constraints

Estimate error and refine

```
template <int dim>
void Step6<dim>::refine_grid ()
{
  Vector<float> estimated_error_per_cell (triangulation.n_active_cells());

  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(3),
                                      typename FunctionMap<dim>::type(),
                                      solution,
                                      estimated_error_per_cell);

  GridRefinement::refine_and_coarsen_fixed_number (triangulation,
                                                   estimated_error_per_cell,
                                                   0.3, 0.03);

  triangulation.execute_coarsening_and_refinement ();
}
```

Estimate error and refine

Get per cell error indicator based on jumps of the solution across faces

```
template <int dim>
void Step6<dim>::refine_grid ()
{
  Vector<float> estimated_error_per_cell (triangulation.n_active_cells());

  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(3),
                                      typename FunctionMap<dim>::type(),
                                      solution,
                                      estimated_error_per_cell);

  GridRefinement::refine_and_coarsen_fixed_number (triangulation,
                                                   estimated_error_per_cell,
                                                   0.3, 0.03);

  triangulation.execute_coarsening_and_refinement ();
}
```

Estimate error and refine

Mark cells for coarsening/refinement using error indicator

```
template <int dim>
void Step6<dim>::refine_grid ()
{
  Vector<float> estimated_error_per_cell (triangulation.n_active_cells());

  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(3),
                                      typename FunctionMap<dim>::type(),
                                      solution,
                                      estimated_error_per_cell);

  GridRefinement::refine_and_coarsen_fixed_number (triangulation,
                                                   estimated_error_per_cell,
                                                   0.3, 0.03);

  triangulation.execute_coarsening_and_refinement ();
}
```

Estimate error and refine

```
template <int dim>
void Step6<dim>::refine_grid ()
{
  Vector<float> estimated_error_per_cell (triangulation.n_active_cells());

  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(3),
                                      typename FunctionMap<dim>::type(),
                                      solution,
                                      estimated_error_per_cell);

  GridRefinement::refine_and_coarsen_fixed_number (triangulation,
                                                   estimated_error_per_cell,
                                                   0.3, 0.03);

  triangulation.execute_coarsening_and_refinement ();
}
```

Refine triangulation

Output results

```
template <int dim>
void Step6<dim>::output_results (const unsigned int cycle) const
{
  // Output object
  DataOut<dim> data_out;
  // Associate DoFHandler and solution vector
  data_out.attach_dof_handler (dof_handler);
  data_out.add_data_vector (solution, "solution");
  // Intermediate format
  data_out.build_patches();
  // Final output
  const std::string filename = "solution_" + Utilities::int_to_string(cycle) + ".vtk";
  std::ofstream output (filename.c_str());
  data_out.write_vtk(output);
}
```

Output results

Object to assist in outputting the solution for visualisation

- Need to let it know about the solution and what to call it
- Note: The solution vector must have scope for the length of existence of the DataOut object

```
template <int dim>
void Step6<dim>::output_results (const unsigned int cycle) const
{
  // Output object
  DataOut<dim> data_out;
  // Associate DoFHandler and solution vector
  data_out.attach_dof_handler (dof_handler);
  data_out.add_data_vector (solution, "solution");
  // Intermediate format
  data_out.build_patches();
  // Final output
  const std::string filename = "solution_" + Utilities::int_to_string(cycle) + ".vtk";
  std::ofstream output (filename.c_str());
  data_out.write_vtk(output);
}
```

Output results

Convert data to an intermediate format
- Allows output of multiple visualisation types
- Only done once all solution data is added

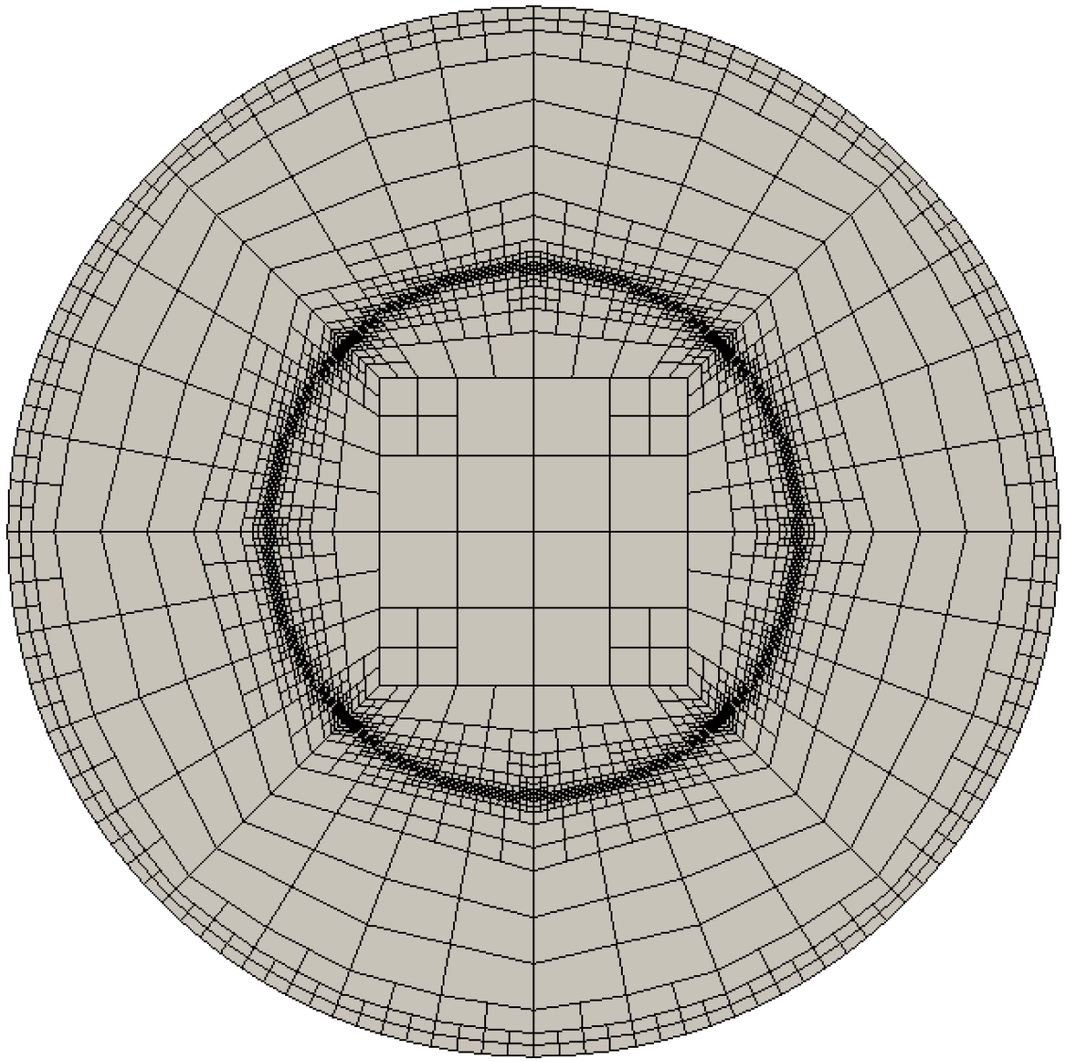
```
template <int dim>
void Step6<dim>::output_results (const unsigned int cycle) const
{
  // Output object
  DataOut<dim> data_out;
  // Associate DoFHandler and solution vector
  data_out.attach_dof_handler (dof_handler);
  data_out.add_data_vector (solution, "solution");
  // Intermediate format
  data_out.build_patches();
  // Final output
  const std::string filename = "solution_" + Utilities::int_to_string(cycle) + ".vtk";
  std::ofstream output (filename.c_str());
  data_out.write_vtk(output);
}
```

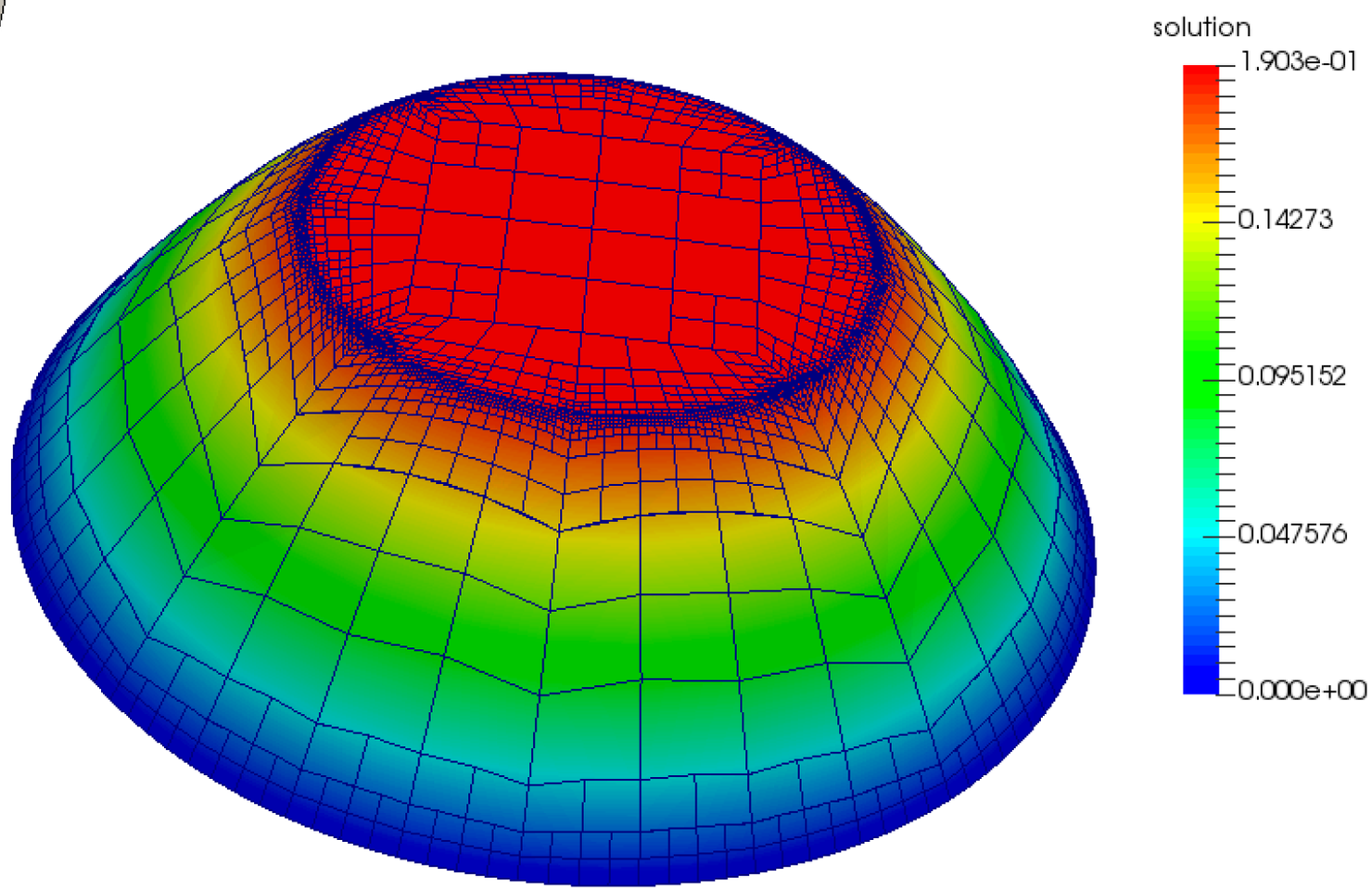
Output results

Output data to file
- VTK format

```
template <int dim>
void Step6<dim>::output_results (const unsigned int cycle) const
{
  // Output object
  DataOut<dim> data_out;
  // Associate DoFHandler and solution vector
  data_out.attach_dof_handler (dof_handler);
  data_out.add_data_vector (solution, "solution");
  // Intermediate format
  data_out.build_patches();
  // Final output
  const std::string filename = "solution_" + Utilities::int_to_string(cycle) + ".vtk";
  std::ofstream output (filename.c_str());
  data_out.write_vtk(output);
}
```

Results





Generalized eigenvalue problems in quantum mechanics solved by matrix-free FEM

Denis Davydov

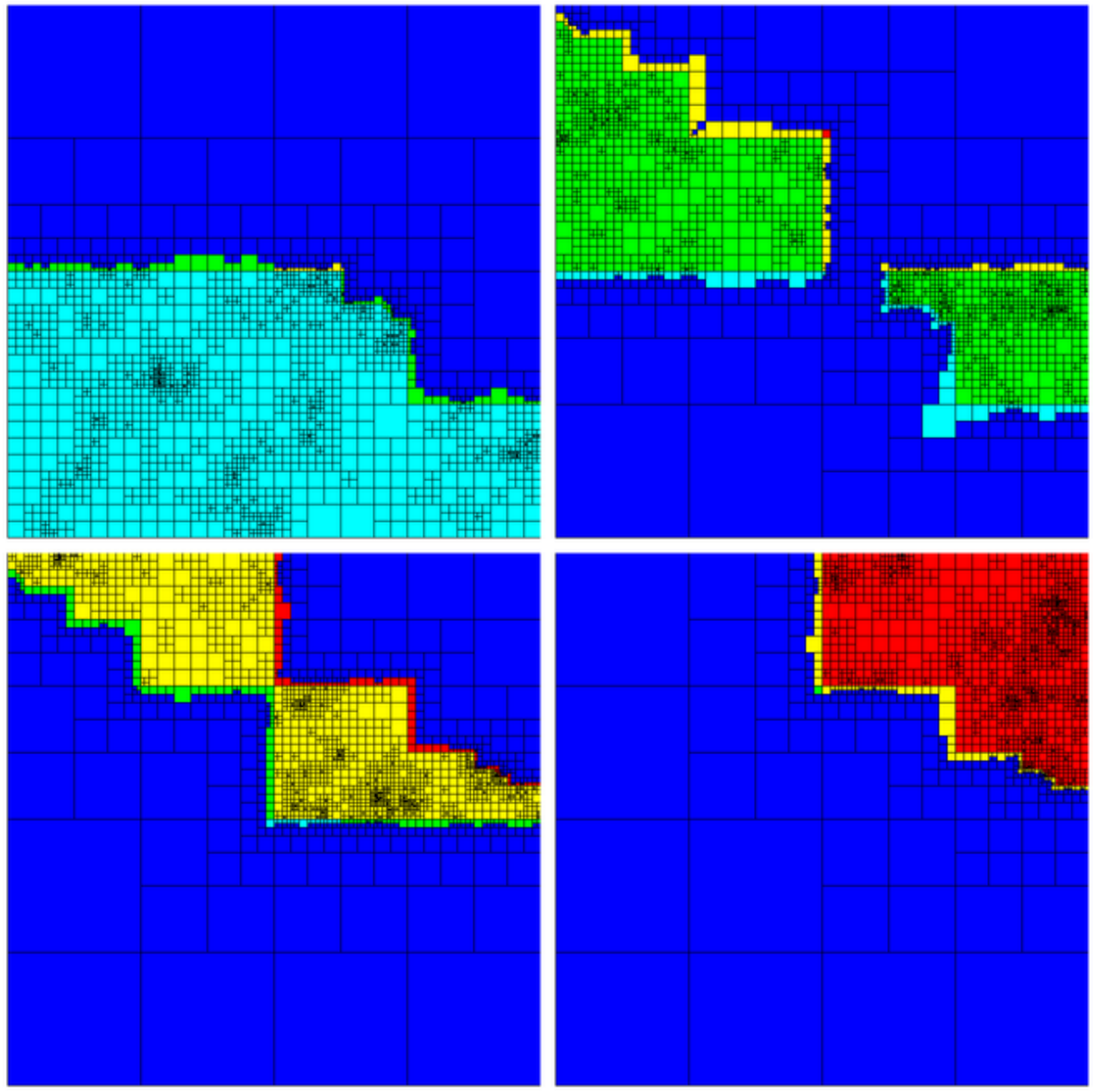
MPI/TBB parallelization in deal.II

parallel::distributed::Triangulation<> store on each processor only a subset of cells: locally owned, ghost and artificial (dark blue).

Index space for FE fields is split into contiguous blocks using cell partition are termed locally owned dofs.

Matrix assembly loops over locally owned cells and can be further parallelized using Intel TBB via WorkStream class.

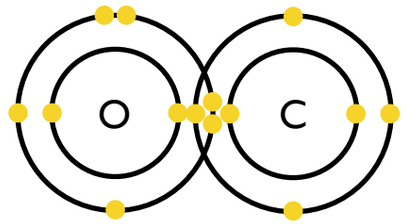
Alternatively one can use matrix-free approach which utilises tensor product nature of both FE space and the quadrature rule. SIMD is employed by working on multiple cells at once.



(courtesy of deal.II authors)

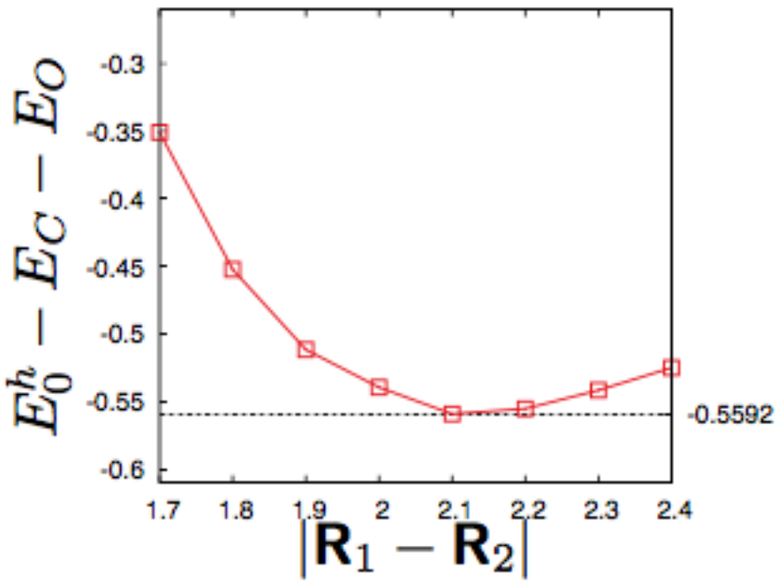
Towards optimisation of nucleus positions

Consider carbon monoxide. Atoms are placed on the x-axis symmetrically about the origin with varying interatomic distance. The same structured mesh is used as an input. **Mesh motion** approach [1,2] is used to deform the mesh so that ions are located at vertices.

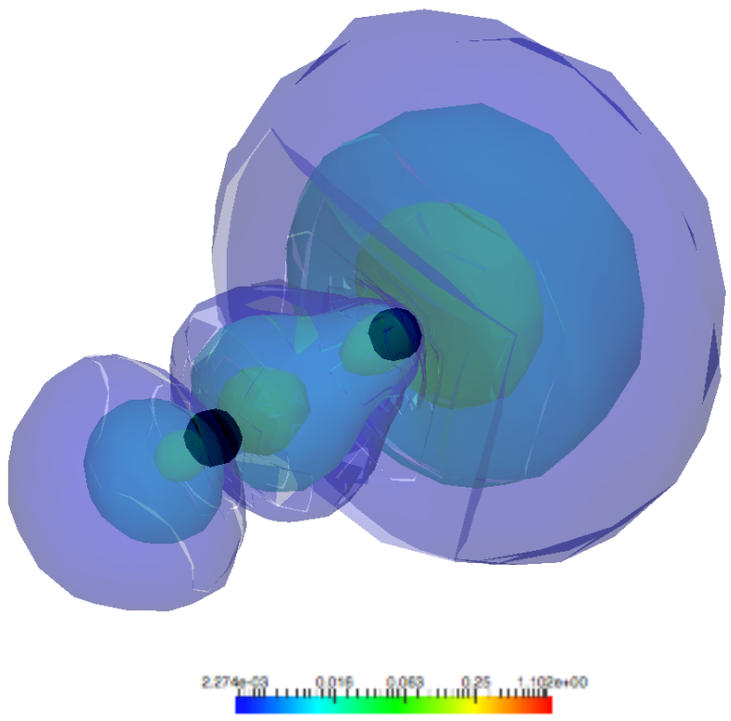


a triple bond

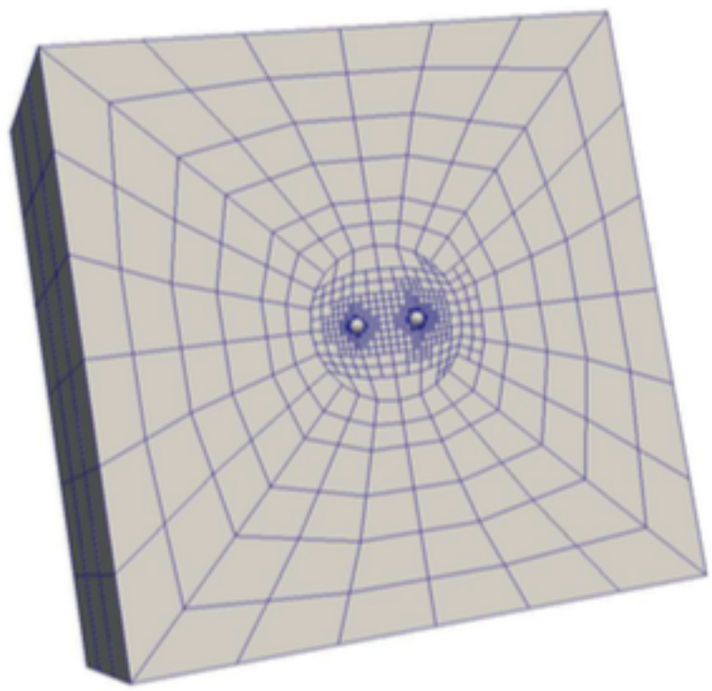
$$^{(+)}:O \equiv C:^{(-)}$$



(a) Bonding energy at the finest mesh.



(b) $|\psi_7(\boldsymbol{x})|^2$

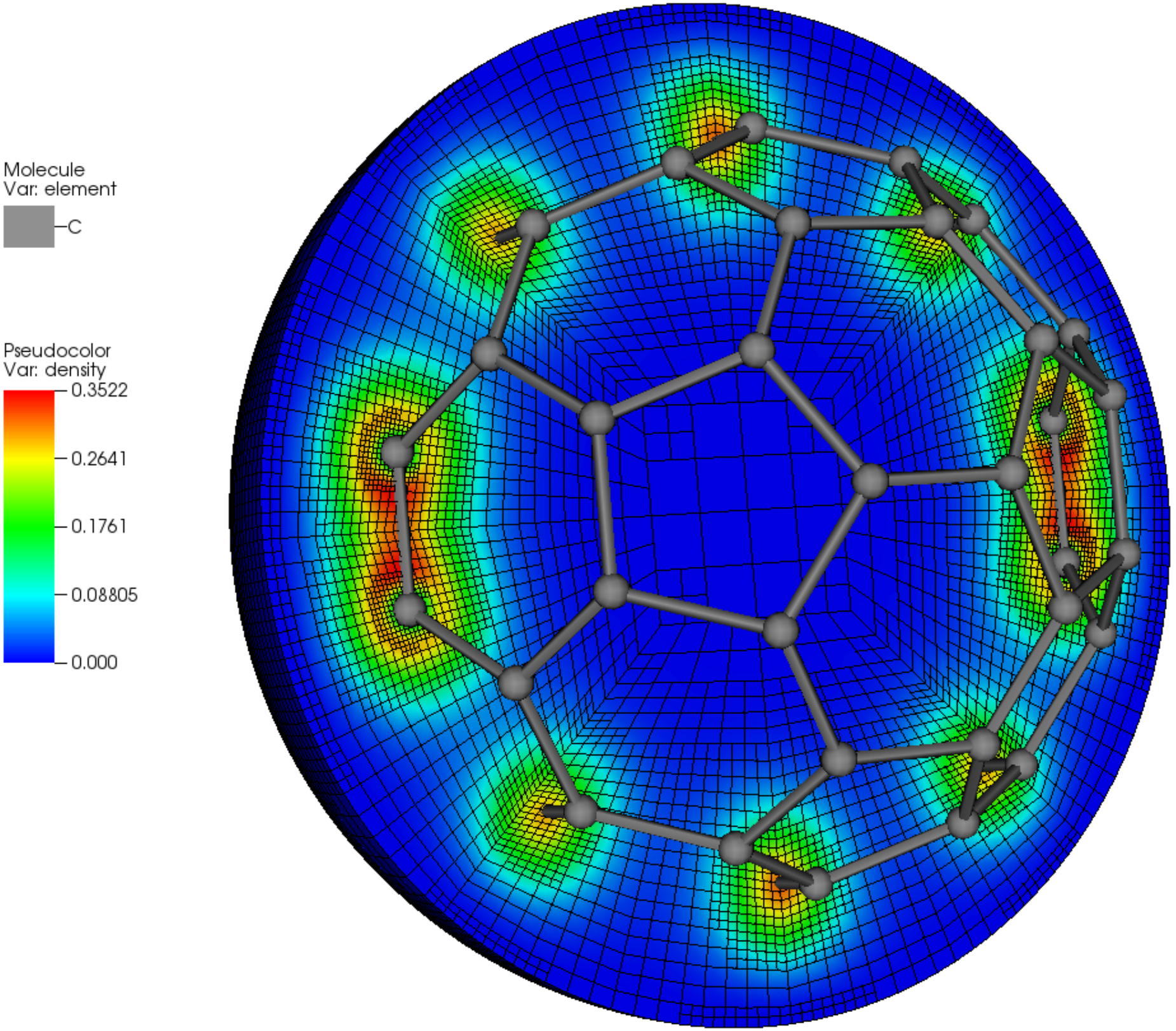


(c) Mesh.

[1] **Davydov, D.**; Young, T. & Steinmann, P. On the adaptive finite element analysis of the Kohn-Sham equations: Methods, algorithms, and implementation. International Journal for Numerical Methods in Engineering, 2016, 106, 863-888

[2] Pelteret, J.-P.; **Davydov, D.**; McBride, A.; Vu, D. K. & Steinmann, P. Computational electro- and magneto-elasticity for quasi-incompressible media immersed in free space International Journal for Numerical Methods in Engineering. Accepted, 2016

Matrix-free numerical example: C60



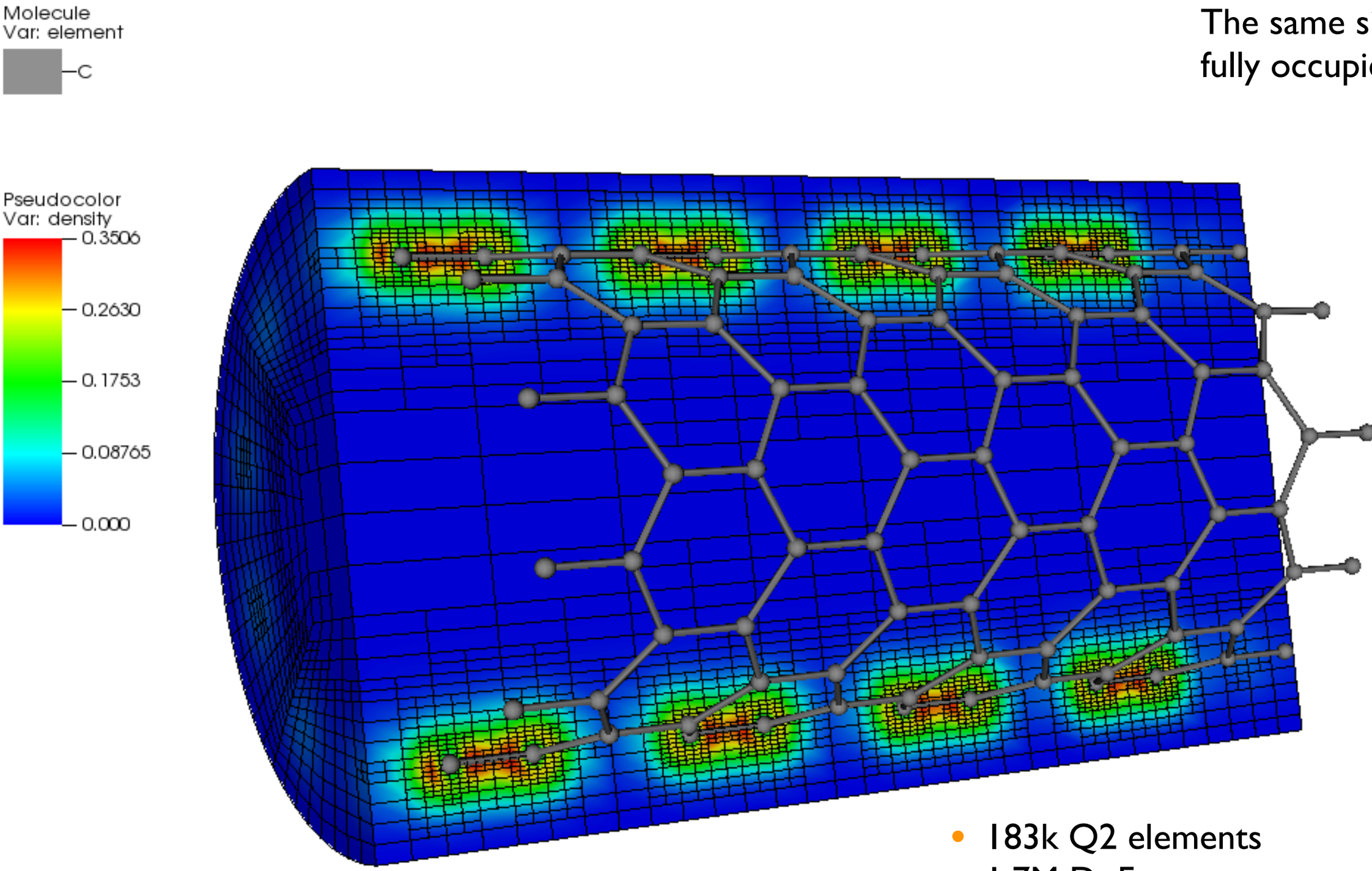
Single node:

- 2 x Intel Xeon 2660v2 IvyBridge
- 2.20GHz
- 10 cores/socket
- 2 threads/core
- 25MB shared cache/socket
- 64GB RAM

FEM details of the finest mesh:

- 420k Q2 elements
- 3.7M DoFs
- 120 orbitals are initialized with Gaussians on bonds
- 3.164e+02s @ 3 SD steps
- convergence based on L2 norm of density increment

Matrix-free numerical example: C160



The same single node, fully occupied.

- 183k Q2 elements
- 1.7M DoFs
- 320 orbitals are initialized randomly in a given radius around each atom.

GHEP with a fixed potential V(x)

Strong form:

$$\left[-\frac{1}{2}\nabla^2 + V(\boldsymbol{x}) \right] \psi_\alpha(\boldsymbol{x}) = \lambda_\alpha \psi_\alpha(\boldsymbol{x}) \quad \text{on } \Omega\,,$$

$$\psi_\alpha(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega,$$

$$\int_\Omega \psi_\alpha(\boldsymbol{x}) \psi_\beta(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \delta_{\alpha\beta}\,.$$

GHEP with a fixed potential V(x)

Strong form:

$$\left[-\frac{1}{2}\nabla^2 + V(\boldsymbol{x}) \right] \psi_\alpha(\boldsymbol{x}) = \lambda_\alpha \psi_\alpha(\boldsymbol{x}) \quad \text{on } \Omega\,,$$

$$\psi_\alpha(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega,$$

$$\int_\Omega \psi_\alpha(\boldsymbol{x})\psi_\beta(\boldsymbol{x})\mathrm{d}\boldsymbol{x} = \delta_{\alpha\beta}\,.$$

Weak form:

$$\int_\Omega \left[\frac{1}{2}\nabla v \cdot \nabla\psi_\alpha + vV\psi_\alpha \right] \mathrm{d}\boldsymbol{x} = \lambda_\alpha \int_\Omega v\psi_\alpha \mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,,$$

$$\int \psi_\alpha\psi_\beta \mathrm{d}\boldsymbol{x} = \delta_{\alpha\beta}\,.$$

(infinite dimensional spaces)

GHEP with a fixed potential V(x)

Strong form:

$$\left[-\frac{1}{2}\nabla^2 + V(\boldsymbol{x}) \right] \psi_\alpha(\boldsymbol{x}) = \lambda_\alpha \psi_\alpha(\boldsymbol{x}) \quad \text{on } \Omega\,,$$

$$\psi_\alpha(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega,$$

$$\int_\Omega \psi_\alpha(\boldsymbol{x})\psi_\beta(\boldsymbol{x})\mathrm{d}\boldsymbol{x} = \delta_{\alpha\beta}\,.$$

Weak form:

$$\int_\Omega \left[\frac{1}{2}\nabla v \cdot \nabla\psi_\alpha + vV\psi_\alpha \right] \mathrm{d}\boldsymbol{x} = \lambda_\alpha \int_\Omega v\psi_\alpha \mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,,$$

(infinite dimensional spaces)

$$\int \psi_\alpha\psi_\beta \mathrm{d}\boldsymbol{x} = \delta_{\alpha\beta}\,.$$

Ritz-Galerkin method: triangulation $\mathcal{P}^h$ of Ω and the associated FE space of continuous piecewise elements of a fixed polynomial degree $\psi_\alpha^h \in V^h \subset H_0^1(\Omega)$

$$\left[\frac{1}{2}\nabla v^h \cdot \nabla\psi_\alpha^h + v^h V\psi_\alpha^h \right] \mathrm{d}\boldsymbol{x} = \lambda_\alpha^h \int_\Omega v^h\psi_\alpha^h \mathrm{d}\boldsymbol{x} \quad \forall\, v^h \in V^h\,,$$

$$\int \psi_\alpha^h\psi_\beta^h \mathrm{d}\boldsymbol{x} = \delta_{\alpha\beta}\,.$$

History detour:

Bubnov-Galerkin method

trial space is build upon the variational space by incorporating generally non-zero Dirichlet boundary conditions. Not necessarily related to minimization of a functional.

Walter Ritz
22.02.1878-07.07.1909

Petrov-Galerkin method

variational and trial spaces are unrelated (i.e. PDE of odd order, collation method, etc). Not necessarily related to minimization of a functional.

Ritz-Galerkin method

Same as Bubnov-Galerkin but in the context of finding a minimum of a functional (variational problems).

Ivan Bubnov
18.01.1872-13.03.1919

Boris Galerkin
04.03.1871-12.06.1945

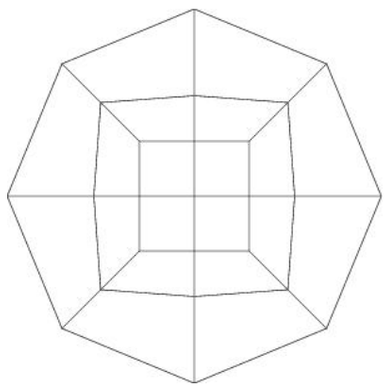
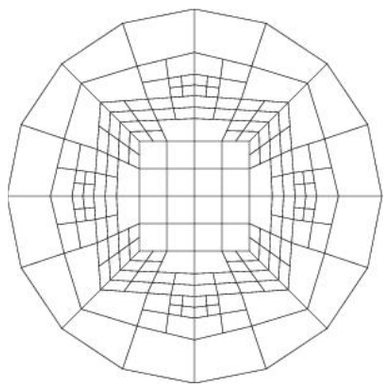
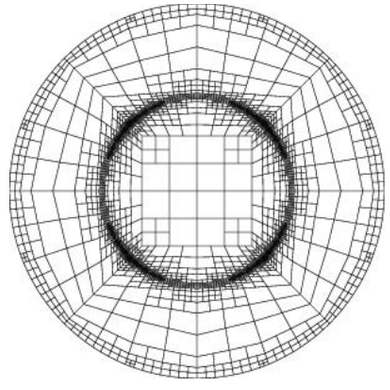
Courant was actually the one who suggested "hat" functions and polynomials on elements.

see Gander, Walker 2012 "From Euler, Ritz, and Galerkin to Modern Computing" (variational calculus started in 1696 from Bernoulli challenging community with the brachistochrone problem) and https://www.unige.ch/~gander/Preprints/RitzTalk.pdf

Richard Courant
08.01.1888-27.01.1972

A posteriori mesh refinement

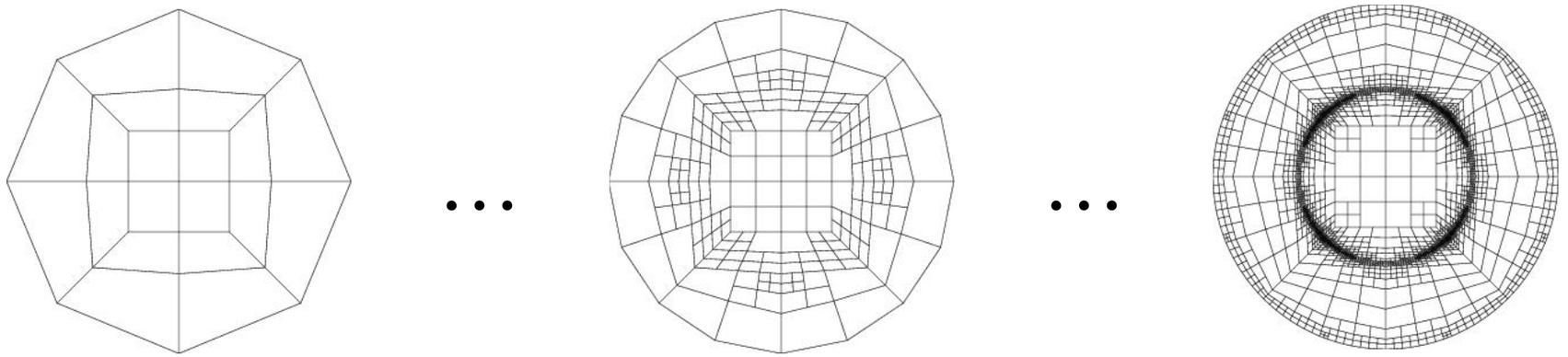
SOLVE - ESTIMATE - MARK - REFINE - SOLVE

(courtesy of deal.II authors)

A posteriori mesh refinement

SOLVE - ESTIMATE - MARK - REFINE - SOLVE



(courtesy of deal.II authors)

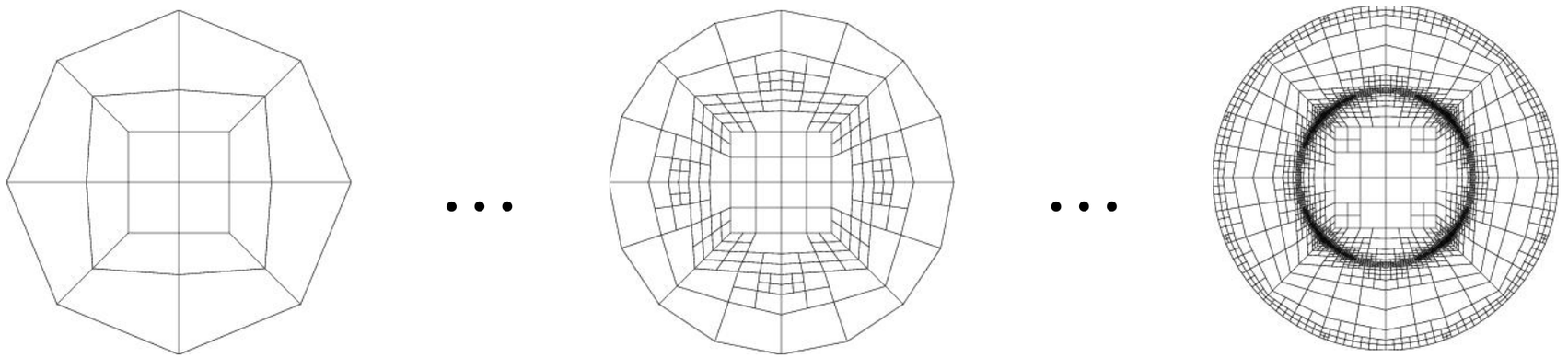
Approach: Residual based error-indicator for each eigenpair:

$$\eta_{K,\alpha}^2 := h_K^2 \int_K \left[\left(-\frac{1}{2}\nabla^2 + V_{\text{eff}}(\boldsymbol{x}) \right) \psi_\alpha^h - \lambda_\alpha^h \psi_\alpha^h \right]^2 \mathrm{d}x + h_K \sum_{e \in \partial K} \int_e \left[\!\left[\frac{1}{2} \boldsymbol{\nabla} \psi_\alpha^h \cdot \boldsymbol{n} \right]\!\right]^2 \mathrm{d}a$$

$$\eta_K^2 := \sum_\alpha \eta_{K,\alpha}^2$$

A posteriori mesh refinement

SOLVE - ESTIMATE - MARK - REFINE - SOLVE



(courtesy of deal.II authors)

Approach: Residual based error-indicator for each eigenpair:

$$\eta_{K,\alpha}^2 := h_K^2 \int_K \left[\left(-\frac{1}{2}\nabla^2 + V_{\text{eff}}(\boldsymbol{x}) \right) \psi_\alpha^h - \lambda_\alpha^h \psi_\alpha^h \right]^2 \mathrm{d}x + h_K \sum_{e \in \partial K} \int_e \left[\!\left[\frac{1}{2} \boldsymbol{\nabla} \psi_\alpha^h \cdot \boldsymbol{n} \right]\!\right]^2 \mathrm{d}a$$

$$\eta_K^2 := \sum_\alpha \eta_{K,\alpha}^2$$

Marking strategy: marks all cells which have error indicators larger than the maximum value among all cells

$$\{K' : \eta_{K'} \geq \theta \max_{K \in \mathcal{P}^h} \eta_K\}, \quad \theta \in (0, 1]$$

Main class

For optimal performance, a lot of variables should be known at compile time.

```
template <int dim, int fe_degree=1,int n_q_points=fe_degree+1, typename NumberType = double>
class EigenvalueProblem
{
public:
  ~EigenvalueProblem();
  EigenvalueProblem(const EigenvalueParameters &parameters);
  void run();

private:
  typedef LinearAlgebra::distributed::Vector<NumberType> VectorType;

  void make_mesh();
  void setup_system();
  void solve(const unsigned int cycle);
  void adjust_ghost_range(std::vector<LinearAlgebra::distributed::Vector<NumberType>> &eigenfunctions)
  void estimate_error(Vector<float> &error) const;
  void refine();
  void output(const unsigned int iteration) const;

  const EigenvalueParameters &parameters;

  MPI_Comm mpi_communicator;
  parallel::distributed::Triangulation<dim> triangulation;
  DoFHandler<dim> dof_handler;
  FE_Q<dim> fe;
  MappingQ<dim>  mapping;
  QGauss<1> quadrature_formula;

  ConstraintMatrix constraints;
  IndexSet locally_relevant_dofs;

  std::vector<LinearAlgebra::distributed::Vector<NumberType>> eigenfunctions;
  std::vector<NumberType> eigenvalues;

  std::shared_ptr<MatrixFree<dim,NumberType>> fine_level_data;

  HamiltonianOperator<dim,fe_degree,n_q_points,1,VectorType> hamiltonian_operator;
  MassOperator       <dim,fe_degree,n_q_points,1,VectorType> mass_operator;

  FunctionParser<dim> potential;
  std::shared_ptr<Table<2, VectorizedArray<NumberType>>> coefficient;

  Vector<float> estimated_error_per_cell;
};
```

Main class

For optimal performance, a lot of variables should be known at compile time.

Standard operations to initialise and solve problem

```
template <int dim, int fe_degree=1,int n_q_points=fe_degree+1, typename NumberType = double>
class EigenvalueProblem
{
public:
  ~EigenvalueProblem();
  EigenvalueProblem(const EigenvalueParameters &parameters);
  void run();

private:
  typedef LinearAlgebra::distributed::Vector<NumberType> VectorType;

  void make_mesh();
  void setup_system();
  void solve(const unsigned int cycle);
  void adjust_ghost_range(std::vector<LinearAlgebra::distributed::Vector<NumberType>> &eigenfunctions)
  void estimate_error(Vector<float> &error) const;
  void refine();
  void output(const unsigned int iteration) const;

  const EigenvalueParameters &parameters;

  MPI_Comm mpi_communicator;
  parallel::distributed::Triangulation<dim> triangulation;
  DoFHandler<dim> dof_handler;
  FE_Q<dim> fe;
  MappingQ<dim>  mapping;
  QGauss<1> quadrature_formula;

  ConstraintMatrix constraints;
  IndexSet locally_relevant_dofs;

  std::vector<LinearAlgebra::distributed::Vector<NumberType>> eigenfunctions;
  std::vector<NumberType> eigenvalues;

  std::shared_ptr<MatrixFree<dim,NumberType>> fine_level_data;

  HamiltonianOperator<dim,fe_degree,n_q_points,1,VectorType> hamiltonian_operator;
  MassOperator       <dim,fe_degree,n_q_points,1,VectorType> mass_operator;

  FunctionParser<dim> potential;
  std::shared_ptr<Table<2, VectorizedArray<NumberType>>> coefficient;

  Vector<float> estimated_error_per_cell;
};
```

Main class

For optimal performance, a lot of variables should be known at compile time.

Distributed triangulation, Lagrange FE and Gaussian quadrature rule.

```
template <int dim, int fe_degree=1,int n_q_points=fe_degree+1, typename NumberType = double>
class EigenvalueProblem
{
public:
  ~EigenvalueProblem();
  EigenvalueProblem(const EigenvalueParameters &parameters);
  void run();

private:
  typedef LinearAlgebra::distributed::Vector<NumberType> VectorType;

  void make_mesh();
  void setup_system();
  void solve(const unsigned int cycle);
  void adjust_ghost_range(std::vector<LinearAlgebra::distributed::Vector<NumberType>> &eigenfunctions)
  void estimate_error(Vector<float> &error) const;
  void refine();
  void output(const unsigned int iteration) const;

  const EigenvalueParameters &parameters;

  MPI_Comm mpi_communicator;
  parallel::distributed::Triangulation<dim> triangulation;
  DoFHandler<dim> dof_handler;
  FE_Q<dim> fe;
  MappingQ<dim>  mapping;
  QGauss<1> quadrature_formula;

  ConstraintMatrix constraints;
  IndexSet locally_relevant_dofs;

  std::vector<LinearAlgebra::distributed::Vector<NumberType>> eigenfunctions;
  std::vector<NumberType> eigenvalues;

  std::shared_ptr<MatrixFree<dim,NumberType>> fine_level_data;

  HamiltonianOperator<dim,fe_degree,n_q_points,1,VectorType> hamiltonian_operator;
  MassOperator       <dim,fe_degree,n_q_points,1,VectorType> mass_operator;

  FunctionParser<dim> potential;
  std::shared_ptr<Table<2, VectorizedArray<NumberType>>> coefficient;

  Vector<float> estimated_error_per_cell;
};
```

Main class

For optimal performance, a lot of variables should be known at compile time.

```
template <int dim, int fe_degree=1,int n_q_points=fe_degree+1, typename NumberType = double>
class EigenvalueProblem
{
public:
  ~EigenvalueProblem();
  EigenvalueProblem(const EigenvalueParameters &parameters);
  void run();

private:
  typedef LinearAlgebra::distributed::Vector<NumberType> VectorType;

  void make_mesh();
  void setup_system();
  void solve(const unsigned int cycle);
  void adjust_ghost_range(std::vector<LinearAlgebra::distributed::Vector<NumberType>> &eigenfunctions)
  void estimate_error(Vector<float> &error) const;
  void refine();
  void output(const unsigned int iteration) const;

  const EigenvalueParameters &parameters;

  MPI_Comm mpi_communicator;
  parallel::distributed::Triangulation<dim> triangulation;
  DoFHandler<dim> dof_handler;
  FE_Q<dim> fe;
  MappingQ<dim>  mapping;
  QGauss<1> quadrature_formula;

  ConstraintMatrix constraints;
  IndexSet locally_relevant_dofs;

  std::vector<LinearAlgebra::distributed::Vector<NumberType>> eigenfunctions;
  std::vector<NumberType> eigenvalues;

  std::shared_ptr<MatrixFree<dim,NumberType>> fine_level_data;

  HamiltonianOperator<dim,fe_degree,n_q_points,1,VectorType> hamiltonian_operator;
  MassOperator       <dim,fe_degree,n_q_points,1,VectorType> mass_operator;

  FunctionParser<dim> potential;
  std::shared_ptr<Table<2, VectorizedArray<NumberType>>> coefficient;

  Vector<float> estimated_error_per_cell;
};
```

Matrix-free related objects: operators, and coefficient to be used.

Constructor and mesh generation

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::EigenvalueProblem(const Eigenvalu
  :
  parameters(parameters),
  mpi_communicator(MPI_COMM_WORLD),
  n_mpi_processes(Utilities::MPI::n_mpi_processes(mpi_communicator)),
  this_mpi_process(Utilities::MPI::this_mpi_process(mpi_communicator)),
  pcout(std::cout, this_mpi_process==0),
  plog(output_fstream, this_mpi_process==0),
  computing_timer(mpi_communicator,
                  pcout,
                  TimerOutput::summary,
                  TimerOutput::wall_times),
  triangulation(mpi_communicator,
                // guarantee that the mesh also does not change by more than refinement l
                Triangulation<dim>::limit_level_difference_at_vertices,
                parallel::distributed::Triangulation<dim>::construct_multigrid_hierarchy)
  dof_handler(triangulation),
  fe(fe_degree),
  mapping(fe_degree+1),
  quadrature_formula(n_q_points),
  eigenfunctions(parameters.number_of_eigenvalues),
  eigenvalues(parameters.number_of_eigenvalues)
{
  potential.initialize (FunctionParser<dim>::default_variable_names (),
                        parameters.potential,
                        typename FunctionParser<dim>::ConstMap());
}
```

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::make_mesh()
{
  TimerOutput::Scope t (computing_timer, "Make mesh");
  GridGenerator::hyper_cube (triangulation, -parameters.size/2., parameters.size/2.);
  triangulation.refine_global (parameters.global_mesh_refinement_steps);
}
```

Constructor and mesh generation

Constructor for parallel Triangulation

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::EigenvalueProblem(const Eigenvalu
  :
  parameters(parameters),
  mpi_communicator(MPI_COMM_WORLD),
  n_mpi_processes(Utilities::MPI::n_mpi_processes(mpi_communicator)),
  this_mpi_process(Utilities::MPI::this_mpi_process(mpi_communicator)),
  pcout(std::cout, this_mpi_process==0),
  plog(output_fstream, this_mpi_process==0),
  computing_timer(mpi_communicator,
                  pcout,
                  TimerOutput::summary,
                  TimerOutput::wall_times),
  triangulation(mpi_communicator,
                // guarantee that the mesh also does not change by more than refinement l
                Triangulation<dim>::limit_level_difference_at_vertices,
                parallel::distributed::Triangulation<dim>::construct_multigrid_hierarchy)
  dof_handler(triangulation),
  fe(fe_degree),
  mapping(fe_degree+1),
  quadrature_formula(n_q_points),
  eigenfunctions(parameters.number_of_eigenvalues),
  eigenvalues(parameters.number_of_eigenvalues)
{
  potential.initialize (FunctionParser<dim>::default_variable_names (),
                        parameters.potential,
                        typename FunctionParser<dim>::ConstMap());
}
```

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::make_mesh()
{
  TimerOutput::Scope t (computing_timer, "Make mesh");
  GridGenerator::hyper_cube (triangulation, -parameters.size/2., parameters.size/2.);
  triangulation.refine_global (parameters.global_mesh_refinement_steps);
}
```

Constructor and mesh generation

Associate the DoFHandler with a triangulation object, setup FE object with a given degree, quadrature rule and mapping.

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::EigenvalueProblem(const Eigenvalu
  :
  parameters(parameters),
  mpi_communicator(MPI_COMM_WORLD),
  n_mpi_processes(Utilities::MPI::n_mpi_processes(mpi_communicator)),
  this_mpi_process(Utilities::MPI::this_mpi_process(mpi_communicator)),
  pcout(std::cout, this_mpi_process==0),
  plog(output_fstream, this_mpi_process==0),
  computing_timer(mpi_communicator,
                  pcout,
                  TimerOutput::summary,
                  TimerOutput::wall_times),
  triangulation(mpi_communicator,
                // guarantee that the mesh also does not change by more than refinement l
                Triangulation<dim>::limit_level_difference_at_vertices,
                parallel::distributed::Triangulation<dim>::construct_multigrid_hierarchy)
  dof_handler(triangulation),
  fe(fe_degree),
  mapping(fe_degree+1),
  quadrature_formula(n_q_points),
  eigenfunctions(parameters.number_of_eigenvalues),
  eigenvalues(parameters.number_of_eigenvalues)
{
  potential.initialize (FunctionParser<dim>::default_variable_names (),
                        parameters.potential,
                        typename FunctionParser<dim>::ConstMap());
}
```

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::make_mesh()
{
  TimerOutput::Scope t (computing_timer, "Make mesh");
  GridGenerator::hyper_cube (triangulation, -parameters.size/2., parameters.size/2.);
  triangulation.refine_global (parameters.global_mesh_refinement_steps);
}
```

Constructor and mesh generation

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::EigenvalueProblem(const Eigenvalu
  :
  parameters(parameters),
  mpi_communicator(MPI_COMM_WORLD),
  n_mpi_processes(Utilities::MPI::n_mpi_processes(mpi_communicator)),
  this_mpi_process(Utilities::MPI::this_mpi_process(mpi_communicator)),
  pcout(std::cout, this_mpi_process==0),
  plog(output_fstream, this_mpi_process==0),
  computing_timer(mpi_communicator,
                  pcout,
                  TimerOutput::summary,
                  TimerOutput::wall_times),
  triangulation(mpi_communicator,
                // guarantee that the mesh also does not change by more than refinement l
                Triangulation<dim>::limit_level_difference_at_vertices,
                parallel::distributed::Triangulation<dim>::construct_multigrid_hierarchy)
  dof_handler(triangulation),
  fe(fe_degree),
  mapping(fe_degree+1),
  quadrature_formula(n_q_points),
  eigenfunctions(parameters.number_of_eigenvalues),
  eigenvalues(parameters.number_of_eigenvalues)
{
  potential.initialize (FunctionParser<dim>::default_variable_names (),
                        parameters.potential,
                        typename FunctionParser<dim>::ConstMap());
}
```

Hypercube with specified size and number of global refinements.

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::make_mesh()
{
  TimerOutput::Scope t (computing_timer, "Make mesh");
  GridGenerator::hyper_cube (triangulation, -parameters.size/2., parameters.size/2.);
  triangulation.refine_global (parameters.global_mesh_refinement_steps);
}
```

setup_system: Data structure initialization

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::setup_system()
{
  <something-not-so-interesting>
  // matrix-free data
  fine_level_data.reset();
  fine_level_data = std::make_shared<MatrixFree<dim,NumberType>>();
  typename MatrixFree<dim,NumberType>::AdditionalData data;
  data.tasks_parallel_scheme = MatrixFree<dim,NumberType>::AdditionalData::partition_colo
  data.mapping_update_flags = update_values | update_gradients | update_JxW_values | upda
  fine_level_data->reinit (mapping, dof_handler, constraints, quadrature_formula, data);

  // initialize matrix-free operators:
  mass_operator.initialize(fine_level_data);
  hamiltonian_operator.initialize(fine_level_data);

  <something-not-so-interesting>
  // evaluate potential
  coefficient.reset();
  coefficient = std::make_shared<Table<2, VectorizedArray<NumberType>>>();
{
  FEEvaluation<dim,fe_degree,n_q_points,1,NumberType> fe_eval(*fine_level_data);
  const unsigned int n_cells = fine_level_data->n_macro_cells();
  const unsigned int nqp = fe_eval.n_q_points;
  coefficient->reinit(n_cells, nqp);
  for (unsigned int cell=0; cell<n_cells; ++cell)
    {
      fe_eval.reinit(cell);
      for (unsigned int q=0; q<nqp; ++q)
        {
          VectorizedArray<NumberType> val = make_vectorized_array<NumberType> (0.);
          Point<dim> p;
          for (unsigned int v = 0; v < VectorizedArray<NumberType>::n_array_elements;
            {
              for (unsigned int d = 0; d < dim; ++d)
                p[d] = fe_eval.quadrature_point(q)[d][v];
              val[v] = potential.value(p) - parameters.shift;
            }
          (*coefficient)(cell,q) = val;
        }
    }
}

  hamiltonian_operator.set_coefficient(coefficient);
}
```

setup_system: Data structure initialization

Main matrix-free object: we specify TBB partitioning, mapping, DoFHandler, constraints, quadrature and what we will need in our weak form / operators.

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::setup_system()
{
  <something-not-so-interesting>
  // matrix-free data
  fine_level_data.reset();
  fine_level_data = std::make_shared<MatrixFree<dim,NumberType>>();
  typename MatrixFree<dim,NumberType>::AdditionalData data;
  data.tasks_parallel_scheme = MatrixFree<dim,NumberType>::AdditionalData::partition_col
  data.mapping_update_flags = update_values | update_gradients | update_JxW_values | upd
  fine_level_data->reinit (mapping, dof_handler, constraints, quadrature_formula, data);

  // initialize matrix-free operators:
  mass_operator.initialize(fine_level_data);
  hamiltonian_operator.initialize(fine_level_data);

  <something-not-so-interesting>
  // evaluate potential
  coefficient.reset();
  coefficient = std::make_shared<Table<2, VectorizedArray<NumberType>>>();
{
  FEEvaluation<dim,fe_degree,n_q_points,1,NumberType> fe_eval(*fine_level_data);
  const unsigned int n_cells = fine_level_data->n_macro_cells();
  const unsigned int nqp = fe_eval.n_q_points;
  coefficient->reinit(n_cells, nqp);
  for (unsigned int cell=0; cell<n_cells; ++cell)
    {
      fe_eval.reinit(cell);
      for (unsigned int q=0; q<nqp; ++q)
        {
          VectorizedArray<NumberType> val = make_vectorized_array<NumberType> (0.);
          Point<dim> p;
          for (unsigned int v = 0; v < VectorizedArray<NumberType>::n_array_elements;
            {
              for (unsigned int d = 0; d < dim; ++d)
                p[d] = fe_eval.quadrature_point(q)[d][v];
              val[v] = potential.value(p) - parameters.shift;
            }
          (*coefficient)(cell,q) = val;
        }
    }
}

  hamiltonian_operator.set_coefficient(coefficient);
}
```

setup_system: Data structure initialization

Initialize matrix-free operators to use the data object.

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::setup_system()
{
  <something-not-so-interesting>
  // matrix-free data
  fine_level_data.reset();
  fine_level_data = std::make_shared<MatrixFree<dim,NumberType>>();
  typename MatrixFree<dim,NumberType>::AdditionalData data;
  data.tasks_parallel_scheme = MatrixFree<dim,NumberType>::AdditionalData::partition_colo
  data.mapping_update_flags = update_values | update_gradients | update_JxW_values | upda
  fine_level_data->reinit (mapping, dof_handler, constraints, quadrature_formula, data);

  // initialize matrix-free operators:
  mass_operator.initialize(fine_level_data);
  hamiltonian_operator.initialize(fine_level_data);

  <something-not-so-interesting>
  // evaluate potential
  coefficient.reset();
  coefficient = std::make_shared<Table<2, VectorizedArray<NumberType>>>();
  {
    FEEvaluation<dim,fe_degree,n_q_points,1,NumberType> fe_eval(*fine_level_data);
    const unsigned int n_cells = fine_level_data->n_macro_cells();
    const unsigned int nqp = fe_eval.n_q_points;
    coefficient->reinit(n_cells, nqp);
    for (unsigned int cell=0; cell<n_cells; ++cell)
      {
        fe_eval.reinit(cell);
        for (unsigned int q=0; q<nqp; ++q)
          {
            VectorizedArray<NumberType> val = make_vectorized_array<NumberType> (0.);
            Point<dim> p;
            for (unsigned int v = 0; v < VectorizedArray<NumberType>::n_array_elements;
              {
                for (unsigned int d = 0; d < dim; ++d)
                  p[d] = fe_eval.quadrature_point(q)[d][v];
                val[v] = potential.value(p) - parameters.shift;
              }
            (*coefficient)(cell,q) = val;
          }
      }
  }

  hamiltonian_operator.set_coefficient(coefficient);
}
```

setup_system: Data structure initialization

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::setup_system()
{
  <something-not-so-interesting>
  // matrix-free data
  fine_level_data.reset();
  fine_level_data = std::make_shared<MatrixFree<dim,NumberType>>();
  typename MatrixFree<dim,NumberType>::AdditionalData data;
  data.tasks_parallel_scheme = MatrixFree<dim,NumberType>::AdditionalData::partition_colo
  data.mapping_update_flags = update_values | update_gradients | update_JxW_values | upda
  fine_level_data->reinit (mapping, dof_handler, constraints, quadrature_formula, data);

  // initialize matrix-free operators:
  mass_operator.initialize(fine_level_data);
  hamiltonian_operator.initialize(fine_level_data);

  <something-not-so-interesting>
  // evaluate potential
  coefficient.reset();
  coefficient = std::make_shared<Table<2, VectorizedArray<NumberType>>>();
  {
    FEEvaluation<dim,fe_degree,n_q_points,1,NumberType> fe_eval(*fine_level_data);
    const unsigned int n_cells = fine_level_data->n_macro_cells();
    const unsigned int nqp = fe_eval.n_q_points;
    coefficient->reinit(n_cells, nqp);
    for (unsigned int cell=0; cell<n_cells; ++cell)
      {
        fe_eval.reinit(cell);
        for (unsigned int q=0; q<nqp; ++q)
          {
            VectorizedArray<NumberType> val = make_vectorized_array<NumberType> (0.);
            Point<dim> p;
            for (unsigned int v = 0; v < VectorizedArray<NumberType>::n_array_elements;
              {
                for (unsigned int d = 0; d < dim; ++d)
                  p[d] = fe_eval.quadrature_point(q)[d][v];
                val[v] = potential.value(p) - parameters.shift;
              }
            (*coefficient)(cell,q) = val;
          }
      }
  }

  hamiltonian_operator.set_coefficient(coefficient);
}
```

Evaluate the potential at each quadrature point of each cell block (SIMD) including scalar shift value for spectral transformation.

setup_system: Data structure initialization

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::setup_system()
{
  <something-not-so-interesting>
  // matrix-free data
  fine_level_data.reset();
  fine_level_data = std::make_shared<MatrixFree<dim,NumberType>>();
  typename MatrixFree<dim,NumberType>::AdditionalData data;
  data.tasks_parallel_scheme = MatrixFree<dim,NumberType>::AdditionalData::partition_colo
  data.mapping_update_flags = update_values | update_gradients | update_JxW_values | upda
  fine_level_data->reinit (mapping, dof_handler, constraints, quadrature_formula, data);

  // initialize matrix-free operators:
  mass_operator.initialize(fine_level_data);
  hamiltonian_operator.initialize(fine_level_data);

  <something-not-so-interesting>
  // evaluate potential
  coefficient.reset();
  coefficient = std::make_shared<Table<2, VectorizedArray<NumberType>>>();
{
  FEEvaluation<dim,fe_degree,n_q_points,1,NumberType> fe_eval(*fine_level_data);
  const unsigned int n_cells = fine_level_data->n_macro_cells();
  const unsigned int nqp = fe_eval.n_q_points;
  coefficient->reinit(n_cells, nqp);
  for (unsigned int cell=0; cell<n_cells; ++cell)
    {
      fe_eval.reinit(cell);
      for (unsigned int q=0; q<nqp; ++q)
        {
          VectorizedArray<NumberType> val = make_vectorized_array<NumberType> (0.);
          Point<dim> p;
          for (unsigned int v = 0; v < VectorizedArray<NumberType>::n_array_elements;
            {
              for (unsigned int d = 0; d < dim; ++d)
                p[d] = fe_eval.quadrature_point(q)[d][v];
              val[v] = potential.value(p) - parameters.shift;
            }
          (*coefficient)(cell,q) = val;
        }
    }
}

hamiltonian_operator.set_coefficient(coefficient);
}
```

Pass potential to our custom matrix-free operator which is a mixture of Laplace and heterogeneous mass operators.

Solving the GHEP in pArpack

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::solve(const unsigned int cyc
{
  std::vector<std::complex<NumberType>> lambda(parameters.number_of_eigenvalues);

  // set up iterative inverse
  static ReductionControl inner_control_c(dof_handler.n_dofs(), 0.0, 1.e-13);
  SolverCG<VectorType> solver_c(inner_control_c);
  PreconditionIdentity preconditioner;
  const auto shift_and_invert =
    inverse_operator(linear_operator<VectorType>(hamiltonian_operator),
                     solver_c,
                     preconditioner);

  const unsigned int num_arnoldi_vectors = 2*eigenvalues.size() + 2;
  typename PArpackSolver<VectorType>::AdditionalData
  additional_data(num_arnoldi_vectors,
                  PArpackSolver<VectorType>::largest_magnitude,
                  true);

  SolverControl solver_control(dof_handler.n_dofs(), 1e-9, /*log_history*/ false, /*
  PArpackSolver<VectorType> eigensolver(solver_control, mpi_communicator, additional
  eigensolver.set_shift(parameters.shift);

  eigensolver.solve (hamiltonian_operator,
                     mass_operator,
                     shift_and_invert,
                     lambda,
                     eigenfunctions,
                     eigenvalues.size());

}
```

Solving the GHEP in pArpack

Setup application of [H-\sigma M]^{-1} using CG unpreconditioned solver and "linear operators" — syntactic sugar to define linear operators, e.g.
const auto op = (op_a + k * op_b) * op_c;

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::solve(const unsigned int cyc
{
  std::vector<std::complex<NumberType>> lambda(parameters.number_of_eigenvalues);

  // set up iterative inverse
  static ReductionControl inner_control_c(dof_handler.n_dofs(), 0.0, 1.e-13);
  SolverCG<VectorType> solver_c(inner_control_c);
  PreconditionIdentity preconditioner;
  const auto shift_and_invert =
    inverse_operator(linear_operator<VectorType>(hamiltonian_operator),
                     solver_c,
                     preconditioner);

  const unsigned int num_arnoldi_vectors = 2*eigenvalues.size() + 2;
  typename PArpackSolver<VectorType>::AdditionalData
  additional_data(num_arnoldi_vectors,
                  PArpackSolver<VectorType>::largest_magnitude,
                  true);

  SolverControl solver_control(dof_handler.n_dofs(), 1e-9, /*log_history*/ false, /*
  PArpackSolver<VectorType> eigensolver(solver_control, mpi_communicator, additional
  eigensolver.set_shift(parameters.shift);

  eigensolver.solve (hamiltonian_operator,
                     mass_operator,
                     shift_and_invert,
                     lambda,
                     eigenfunctions,
                     eigenvalues.size());

}
```

Solving the GHEP in pArpack

Setup eigensolver, tell which part of the spectrum we need, set the shift value for spectral transformation.

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::solve(const unsigned int cyc
{
  std::vector<std::complex<NumberType>> lambda(parameters.number_of_eigenvalues);

  // set up iterative inverse
  static ReductionControl inner_control_c(dof_handler.n_dofs(), 0.0, 1.e-13);
  SolverCG<VectorType> solver_c(inner_control_c);
  PreconditionIdentity preconditioner;
  const auto shift_and_invert =
    inverse_operator(linear_operator<VectorType>(hamiltonian_operator),
                     solver_c,
                     preconditioner);

  const unsigned int num_arnoldi_vectors = 2*eigenvalues.size() + 2;
  typename PArpackSolver<VectorType>::AdditionalData
  additional_data(num_arnoldi_vectors,
                  PArpackSolver<VectorType>::largest_magnitude,
                  true);

  SolverControl solver_control(dof_handler.n_dofs(), 1e-9, /*log_history*/ false, /*
  PArpackSolver<VectorType> eigensolver(solver_control, mpi_communicator, additional
  eigensolver.set_shift(parameters.shift);

  eigensolver.solve (hamiltonian_operator,
                     mass_operator,
                     shift_and_invert,
                     lambda,
                     eigenfunctions,
                     eigenvalues.size());

}
```

Solving the GHEP in pArpack

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::solve(const unsigned int cyc
{
  std::vector<std::complex<NumberType>> lambda(parameters.number_of_eigenvalues);

  // set up iterative inverse
  static ReductionControl inner_control_c(dof_handler.n_dofs(), 0.0, 1.e-13);
  SolverCG<VectorType> solver_c(inner_control_c);
  PreconditionIdentity preconditioner;
  const auto shift_and_invert =
    inverse_operator(linear_operator<VectorType>(hamiltonian_operator),
                     solver_c,
                     preconditioner);

  const unsigned int num_arnoldi_vectors = 2*eigenvalues.size() + 2;
  typename PArpackSolver<VectorType>::AdditionalData
  additional_data(num_arnoldi_vectors,
                  PArpackSolver<VectorType>::largest_magnitude,
                  true);

  SolverControl solver_control(dof_handler.n_dofs(), 1e-9, /*log_history*/ false, /*
  PArpackSolver<VectorType> eigensolver(solver_control, mpi_communicator, additional
  eigensolver.set_shift(parameters.shift);

  eigensolver.solve (hamiltonian_operator,
                     mass_operator,
                     shift_and_invert,
                     lambda,
                     eigenfunctions,
                     eigenvalues.size());
}
```

Solve the GHEP. Note that only mass operator and shift-and-invert operator are used.

Error indicator

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::
estimate_error()
{
  <something-not-so-interesting>
  const double coefficient = -0.5;
  Functions::ConstantFunction<dim> one_half(coefficient);
  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(fe_degree+1),
                                      /*neumann_bc:*/typename FunctionMap<dim>::type(),
                                      sol,
                                      err,
                                      ComponentMask(),
                                      /*coeeficient*/&one_half,
                                      numbers::invalid_unsigned_int,
                                      numbers::invalid_subdomain_id,
                                      numbers::invalid_material_id,
                                      KellyErrorEstimator<dim>::cell_diameter);

  // Now do the volumetric (residual) part:
<something-not-so-interesting>

for (; cell!=endc; ++cell,++cell_no)
  if (cell->is_locally_owned())
    {
      fe_values.reinit (cell);
      const double factor = Utilities::fixed_power<2>(cell->diameter());
      potential.value_list(fe_values.get_quadrature_points(), potential_values);
      for (unsigned int i = 0; i < eigenfunctions.size(); i++)
        {
          fe_values.get_function_laplacians (eigenfunctions[i], laplacians);
          fe_values.get_function_values     (eigenfunctions[i], values);
          double integral = 0.;
          for (unsigned int q=0; q<nqp; ++q)
            integral +=  Utilities::fixed_power<2>(
                           coefficient*laplacians[q] +
                           (potential_values[q] - eigenvalues[i]) * values[q]
                         ) * fe_values.JxW (q);
          integral *= factor;
          estimated_error_per_cell[cell_no] += integral +
             Utilities::fixed_power<2>(errors_per_cell[i][cell_no]);
        }
      estimated_error_per_cell[cell_no] = std::sqrt(estimated_error_per_cell[cell_no]);
    }
}
```

Error indicator

Standard Kelly estimator with consistent with PDE coefficient and scaling based cell diameter

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::
estimate_error()
{
  <something-not-so-interesting>
  const double coefficient = -0.5;
  Functions::ConstantFunction<dim> one_half(coefficient);
  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(fe_degree+1),
                                      /*neumann_bc:*/typename FunctionMap<dim>::type(),
                                      sol,
                                      err,
                                      ComponentMask(),
                                      /*coeeficient*/&one_half,
                                      numbers::invalid_unsigned_int,
                                      numbers::invalid_subdomain_id,
                                      numbers::invalid_material_id,
                                      KellyErrorEstimator<dim>::cell_diameter);

  // Now do the volumetric (residual) part:
  <something-not-so-interesting>

  for (; cell!=endc; ++cell,++cell_no)
    if (cell->is_locally_owned())
      {
        fe_values.reinit (cell);
        const double factor = Utilities::fixed_power<2>(cell->diameter());
        potential.value_list(fe_values.get_quadrature_points(), potential_values);
        for (unsigned int i = 0; i < eigenfunctions.size(); i++)
          {
            fe_values.get_function_laplacians (eigenfunctions[i], laplacians);
            fe_values.get_function_values     (eigenfunctions[i], values);
            double integral = 0.;
            for (unsigned int q=0; q<nqp; ++q)
              integral +=  Utilities::fixed_power<2>(
                             coefficient*laplacians[q] +
                             (potential_values[q] - eigenvalues[i]) * values[q]
                           ) * fe_values.JxW (q);
            integral *= factor;
            estimated_error_per_cell[cell_no] += integral +
               Utilities::fixed_power<2>(errors_per_cell[i][cell_no]);
          }
        estimated_error_per_cell[cell_no] = std::sqrt(estimated_error_per_cell[cell_no]);
      }
}
```

Error indicator

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::
estimate_error()
{
  <something-not-so-interesting>
  const double coefficient = -0.5;
  Functions::ConstantFunction<dim> one_half(coefficient);
  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(fe_degree+1),
                                      /*neumann_bc:*/typename FunctionMap<dim>::type(),
                                      sol,
                                      err,
                                      ComponentMask(),
                                      /*coeeficient*/&one_half,
                                      numbers::invalid_unsigned_int,
                                      numbers::invalid_subdomain_id,
                                      numbers::invalid_material_id,
                                      KellyErrorEstimator<dim>::cell_diameter);

  // Now do the volumetric (residual) part:
  <something-not-so-interesting>

  for (; cell!=endc; ++cell,++cell_no)
    if (cell->is_locally_owned())
      {
        fe_values.reinit (cell);
        const double factor = Utilities::fixed_power<2>(cell->diameter());
        potential.value_list(fe_values.get_quadrature_points(), potential_values);
        for (unsigned int i = 0; i < eigenfunctions.size(); i++)
          {
            fe_values.get_function_laplacians (eigenfunctions[i], laplacians);
            fe_values.get_function_values     (eigenfunctions[i], values);
            double integral = 0.;
            for (unsigned int q=0; q<nqp; ++q)
              integral +=  Utilities::fixed_power<2>(
                             coefficient*laplacians[q] +
                             (potential_values[q] - eigenvalues[i]) * values[q]
                           ) * fe_values.JxW (q);
            integral *= factor;
            estimated_error_per_cell[cell_no] += integral +
               Utilities::fixed_power<2>(errors_per_cell[i][cell_no]);
          }
        estimated_error_per_cell[cell_no] = std::sqrt(estimated_error_per_cell[cell_no]);
      }
}
```

Calculate volumetric part of the indicator

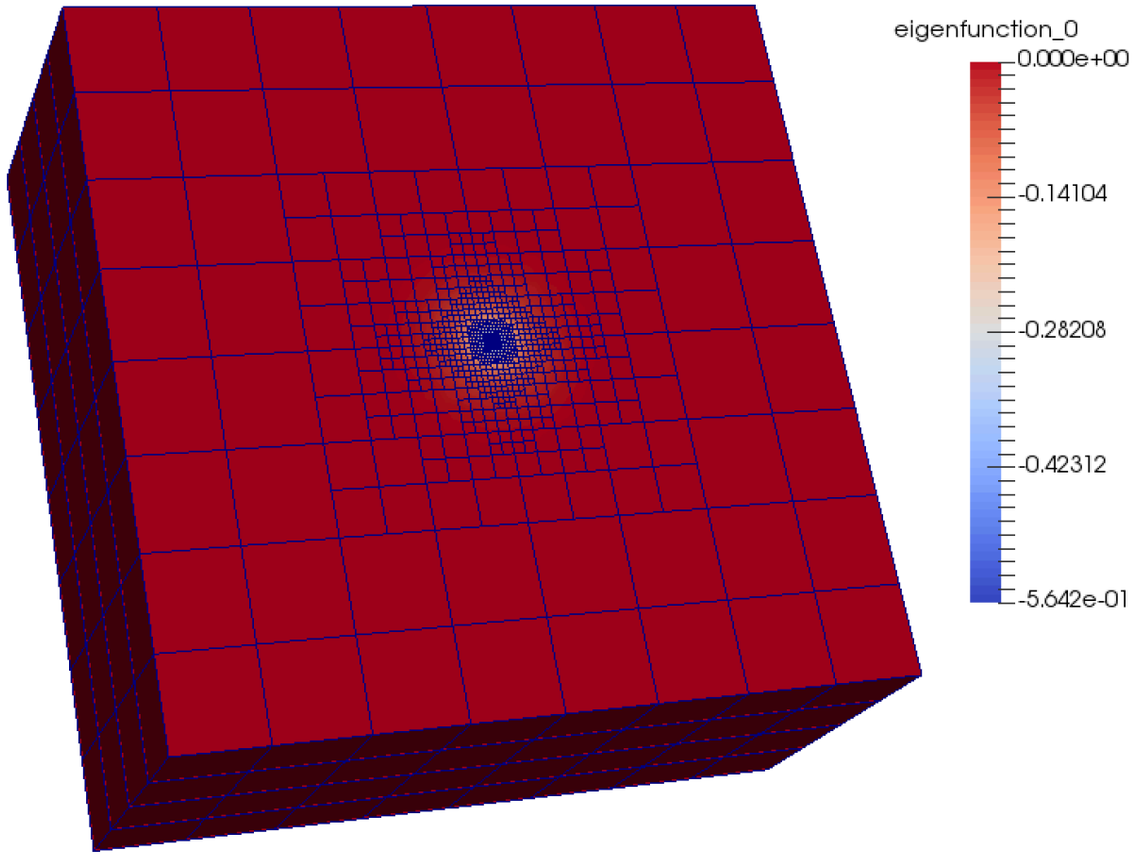
Error indicator

```
template <int dim, int fe_degree, int n_q_points,typename NumberType>
void
EigenvalueProblem<dim,fe_degree,n_q_points,NumberType>::
estimate_error()
{
  <something-not-so-interesting>
  const double coefficient = -0.5;
  Functions::ConstantFunction<dim> one_half(coefficient);
  KellyErrorEstimator<dim>::estimate (dof_handler,
                                      QGauss<dim-1>(fe_degree+1),
                                      /*neumann_bc:*/typename FunctionMap<dim>::type(),
                                      sol,
                                      err,
                                      ComponentMask(),
                                      /*coeeficient*/&one_half,
                                      numbers::invalid_unsigned_int,
                                      numbers::invalid_subdomain_id,
                                      numbers::invalid_material_id,
                                      KellyErrorEstimator<dim>::cell_diameter);

  // Now do the volumetric (residual) part:
<something-not-so-interesting>

for (; cell!=endc; ++cell,++cell_no)
  if (cell->is_locally_owned())
    {
      fe_values.reinit (cell);
      const double factor = Utilities::fixed_power<2>(cell->diameter());
      potential.value_list(fe_values.get_quadrature_points(), potential_values);
      for (unsigned int i = 0; i < eigenfunctions.size(); i++)
        {
          fe_values.get_function_laplacians (eigenfunctions[i], laplacians);
          fe_values.get_function_values     (eigenfunctions[i], values);
          double integral = 0.;
          for (unsigned int q=0; q<nqp; ++q)
            integral +=  Utilities::fixed_power<2>(
                           coefficient*laplacians[q] +
                           (potential_values[q] - eigenvalues[i]) * values[q]
                         ) * fe_values.JxW (q);
          integral *= factor;
          estimated_error_per_cell[cell_no] += integral +
             Utilities::fixed_power<2>(errors_per_cell[i][cell_no]);
        }
      estimated_error_per_cell[cell_no] = std::sqrt(estimated_error_per_cell[cell_no]);
    }
}
```

Sum volumetric and jump terms for a single eigenpair

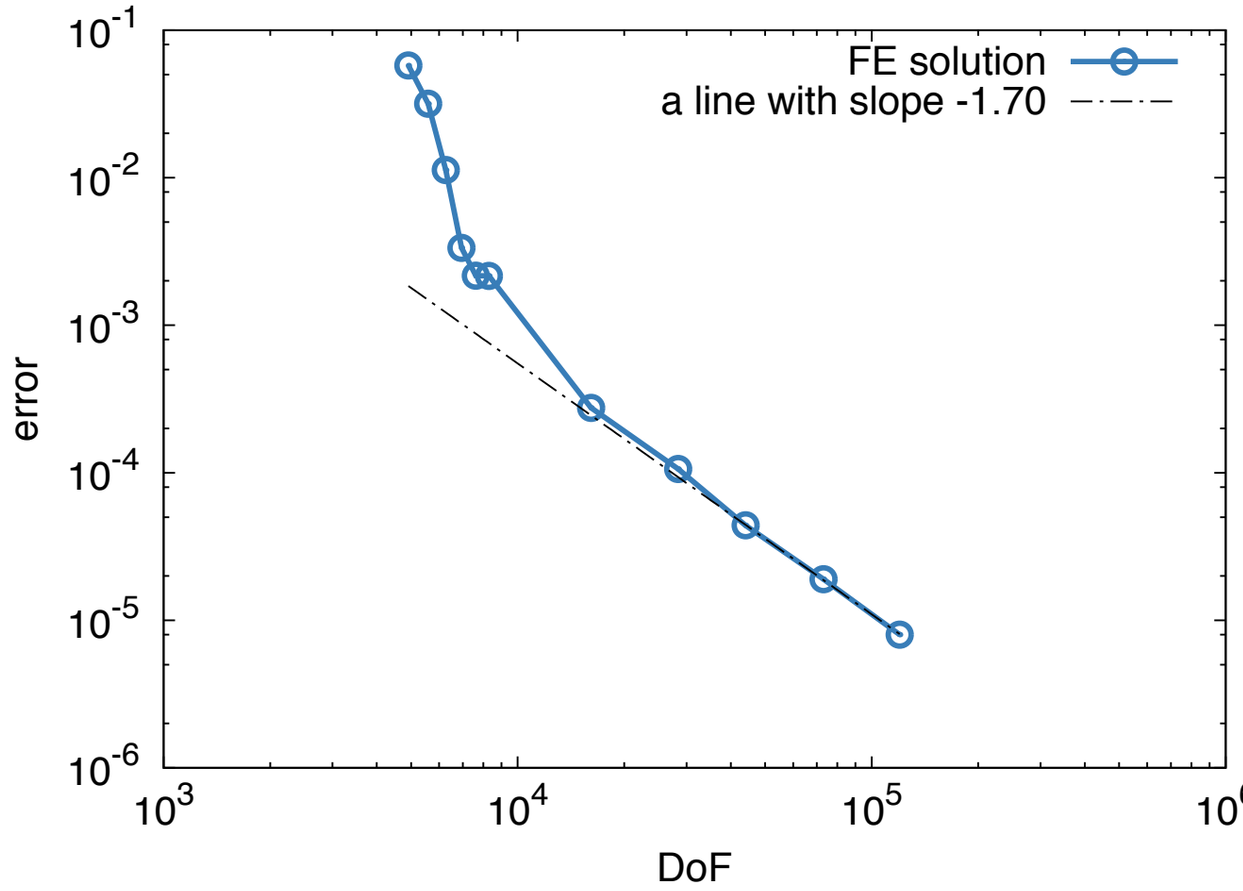
Results

Hydrogen atom: $V = -1/r$



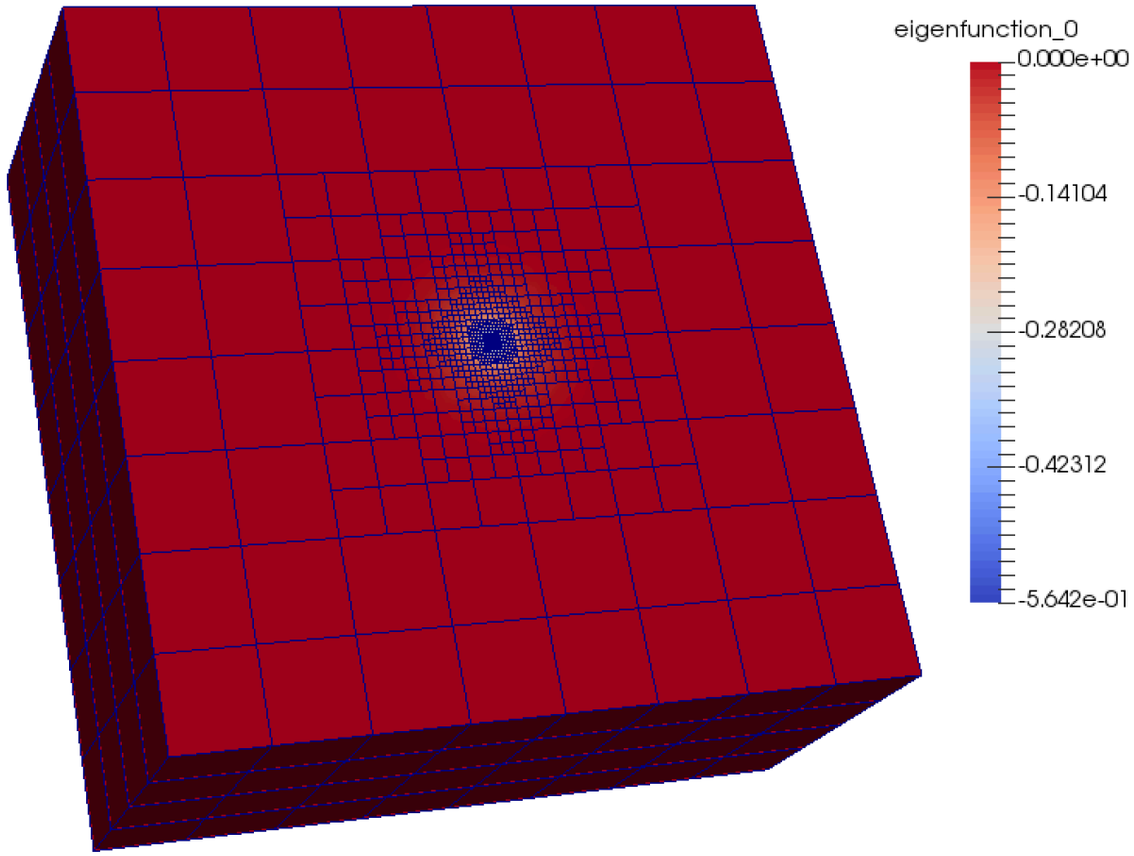
Convergence is variational (thanks to Galerkin projection property of the FEM)

$$\mathcal{O}(h^{2p}) \sim \mathcal{O}(\mathrm{DoFs}^{-2p/3})$$



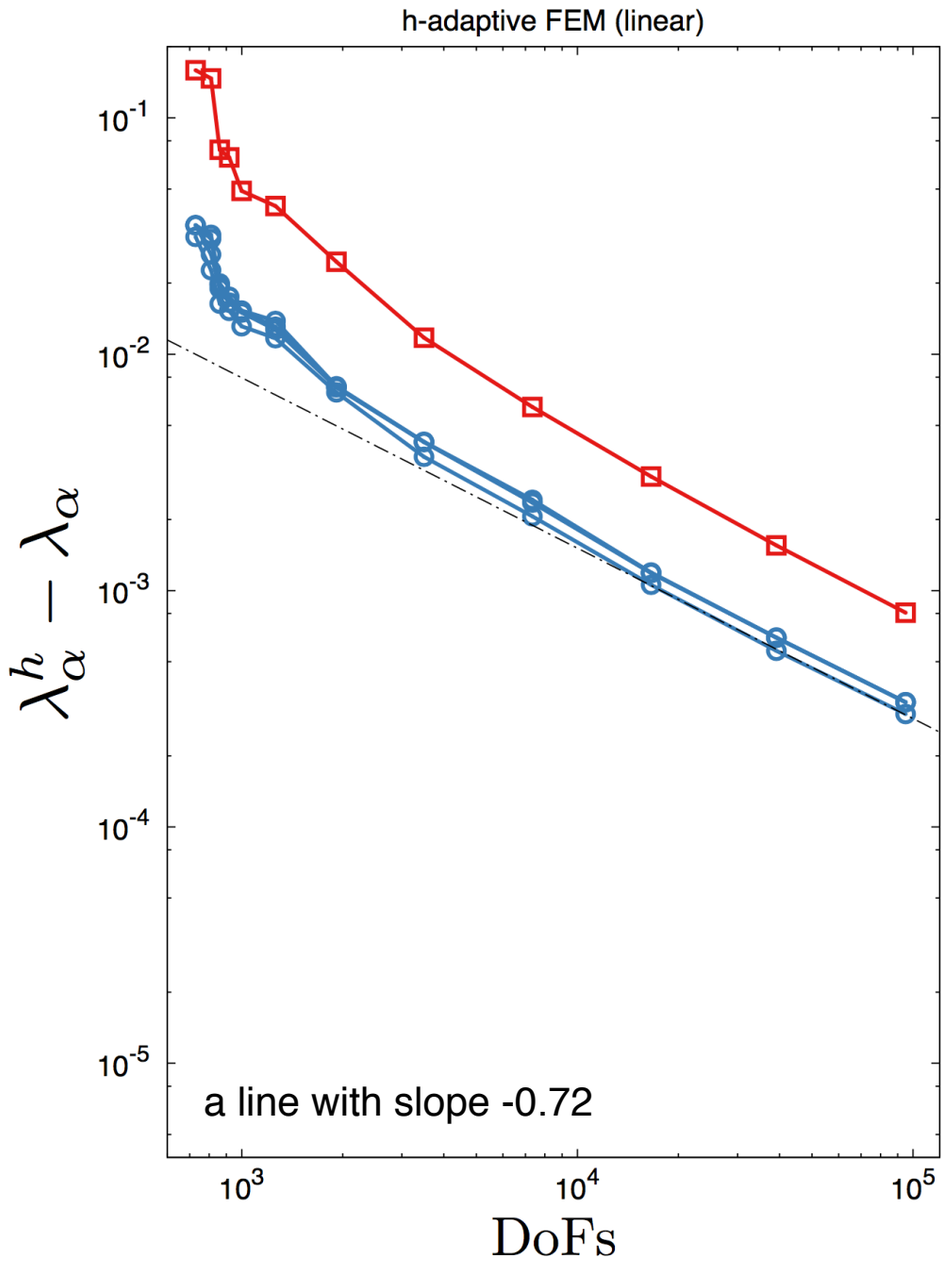
Results

Hydrogen atom: $V = -1/r$



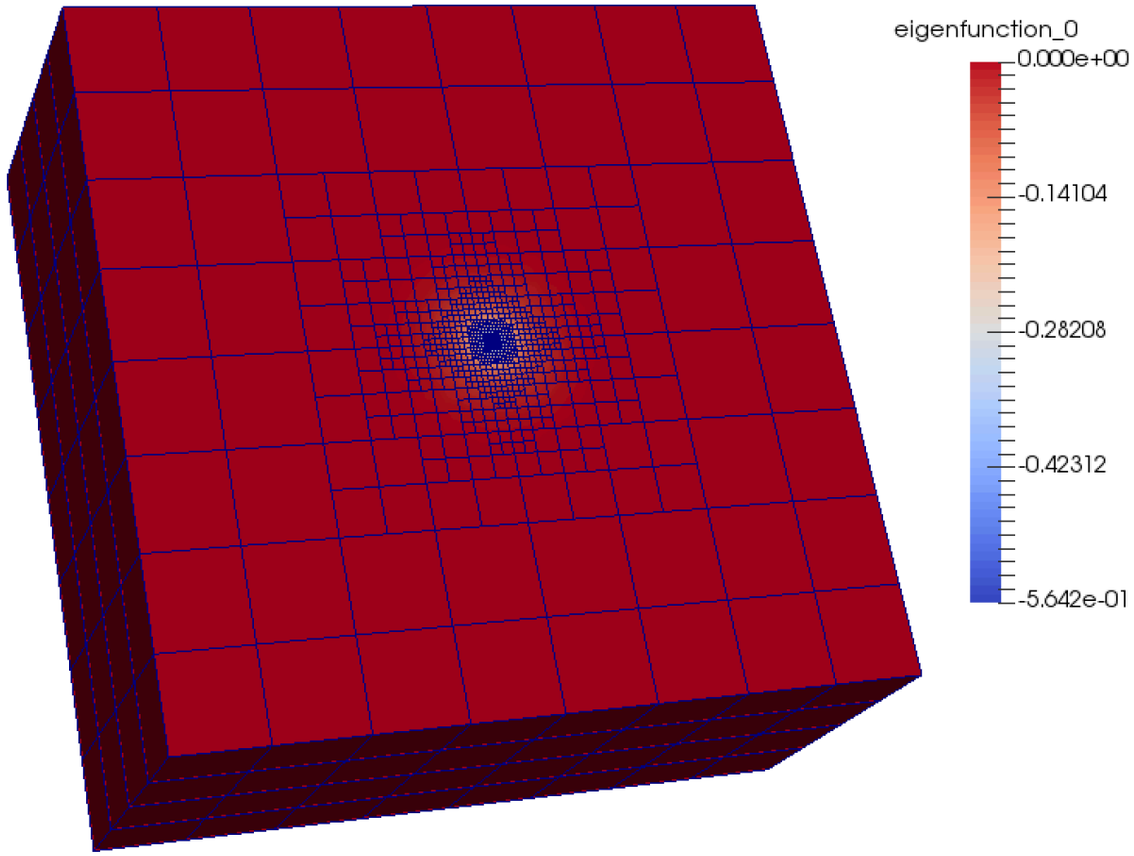
Convergence is variational (thanks to Galerkin projection property of the FEM)

$$\mathcal{O}(h^{2p}) \sim \mathcal{O}(\mathrm{DoFs}^{-2p/3})$$



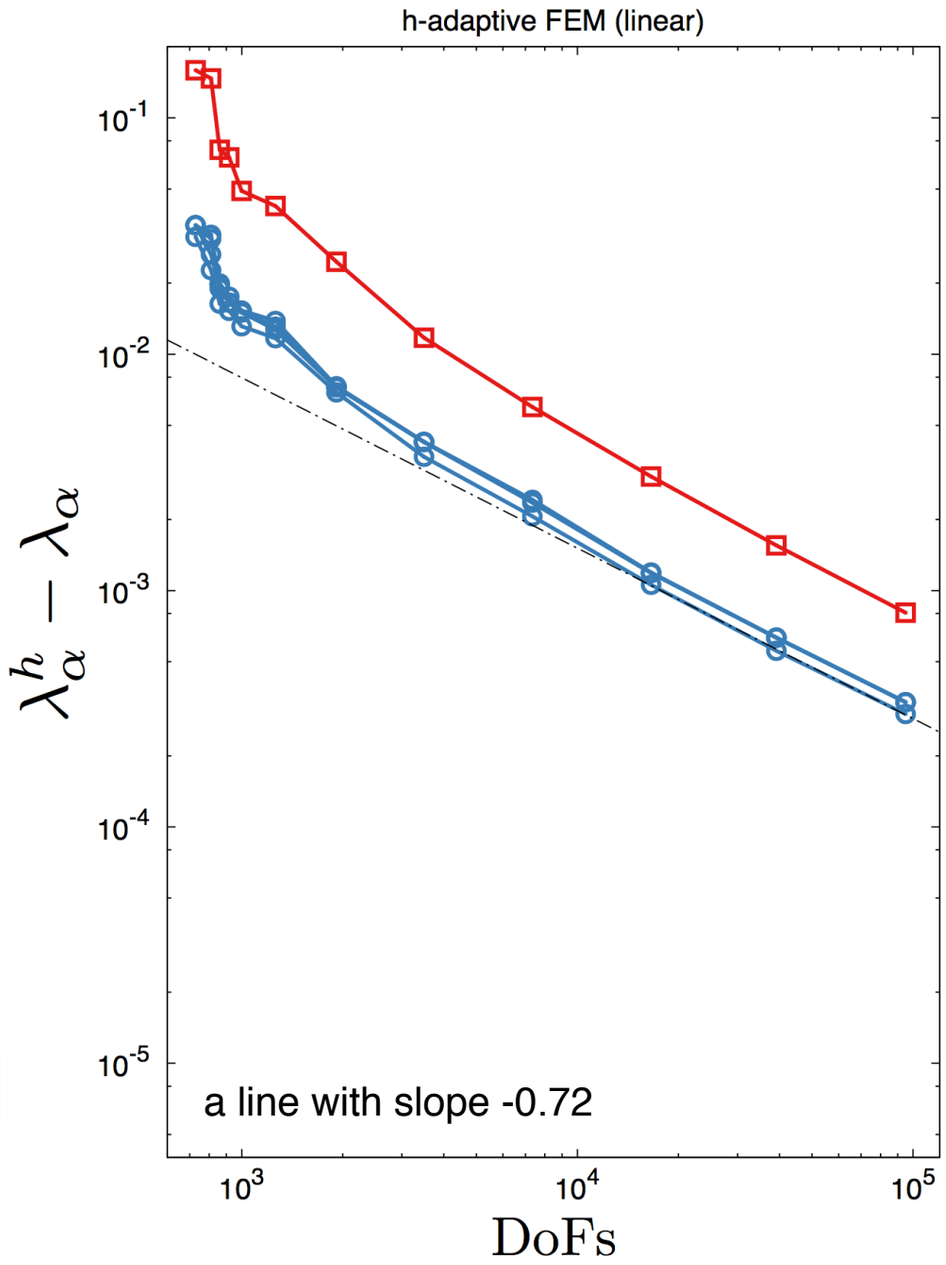
Results

Hydrogen atom: $V = -1/r$



Convergence is variational (thanks to Galerkin projection property of the FEM)

$$\mathcal{O}(h^{2p}) \sim \mathcal{O}(\mathrm{DoFs}^{-2p/3})$$



Experiment with:

0.5 Laplace (2D): $V = 0$ $\lambda_{mn} = \frac{\pi^2}{8}\left[m^2 + n^2\right]$

Harmonic potential (3D): $V = r^2/2$ $\lambda_n = n + 1/2$

Coulomb potential (3D): $V = -1/r$ $\lambda_n = \lambda_1/n^2 \quad \lambda_1 = -1/2$