The step-90 tutorial program

This tutorial depends on **step-85**.

Table of contents

This program was contributed by Vladimir Yushutin.

The material is based upon work partially supported by NSF.

Introduction

In this example, we implement the trace finite element method (TraceFEM) in deal.II. Two aspects are of our focus. First, the surface approximation is separated from the discretization of the surface PDE, e.g. a Q_2 discrete level-set and a Q_1 solution are possible, but both should be based on the same bulk triangulation. Second, we make sure that performance of TraceFEM in the parallel implementation corresponds to that of a classical fitted FEM for a two-dimensional problem. We demonstrate how to achieve both goals by using a combination of **MeshWorker** and **NonMatching** capabilities.

A non-trivial surface

One of the possible bottlenecks in achieving the two-dimensional complexity while solving a surface PDE with TraceFEM is the surface approximation. The level-set function must be defined in a three-dimensional domain and we will create a mesh which is localized near the zero contour line, i.e near the surface, to restore the two-dimensional complexity. To demonstrate this we choose a non-trivial surface Γ given by

$$\frac{x^2}{4}+y^2+\frac{4z^2}{(1+0.5\sin(\pi x))^2}=1$$

We note that the surface's curvature varies significantly.

Model problem

We would like to solve the simplest possible problem, Laplace–Beltrami equation,

$$-\Delta_\Gamma u+u=f \qquad \text{in } \Gamma,$$

Here we added the term u to the left-hand side so the problem becomes well-posed. The force f is calculated analytically for a manufactured solution,

$$u=xy\,,$$

using the exact normal $\mathbf{n}$ and the exact Hessian $\nabla^2\mathbf{n}$ of the surface computed analytically.

The Trace Finite Element Method

TraceFEM is an unfitted method meaning the surface Γ is immersed into a regular, uniform background mesh that stays fixed even if the surface would be evolving. To solve Laplace–Beltrami equation, we first construct an approximation, denoted by Γ_h, of Γ in a form of quadrature points for the surface integration combined with the approximate surface normals. Next we distribute degrees of freedom over a thin subdomain Ω_h that completely covers the Γ_h and that consists of the intersected cells $\mathcal{T}_\Omega^h$,

$$\mathcal{T}_\Omega^h=\{T\in\mathcal{T}^h : T\cap\Gamma_h\neq\emptyset\}.$$

The finite element space where we want to find our numerical solution, u_h, is now

$$V_h=\{v\in C(\Omega_h) : v\in Q_p(T),\, T\in\mathcal{T}_\Omega^h\},$$

where Ω_h is the union of all intersected cells from $\bigcup_{T\in\mathcal{T}_\Omega^h}\overline{T}$.

To create V_h, we first add an **FE_Q** and an **FE_Nothing** element to an **hp::FECollection**. We then iterate over each cell, T, and depending on whether T belongs to $\mathcal{T}_\Omega^h$ or not, we set the active_fe_index to either 0 or 1. For this purpose, we will use the class **NonMatching::MeshClassifier**.

A natural candidate for a weak formulation involves the following bilinear forms

$$a_h(u_h,v_h)=(\nabla_{\Gamma_h}u_h,\nabla_{\Gamma_h}v_h)_{\Gamma_h}+(u_h,v_h)_{\Gamma_h}\,,\qquad L_h(v_h)=(f^e,v_h)_{\Gamma_h}.$$

where f^e is an extension of f from Γ to Ω_h.

However, the so-called "small-cut" problem may arise and one should introduce the stabilized version of TraceFEM: Find $u_h\in V_h$ such that

$$a_h(u_h,v_h)+s_h(u_h,v_h)=L_h(v_h),\quad \forall v_h\in V_\Omega^h,$$

Here the normal-gradient stabilization s_h involves the three-dimensional integration over whole (but intersected) cells and is given by

$$s_h(u_h,v_h)=h^{-1}(\mathbf{n}_h\cdot\nabla u_h,\mathbf{n}_h\cdot\nabla v_h)_{\Omega_h},$$

Note that the h^{-1} scaling may be relaxed for sufficiently smooth solutions such as the manufactured one, but we choose the strong scaling to demonstrate the worst case.

Discrete Level Set Function

In TraceFEM we construct the approximation Γ_h using the interpolant ψ_h of the exact level-set function on the bulk triangulation:

$$\Gamma_h = \{x \in \mathbb{R}^3 : \psi_h(x) = 0\}.$$

The exact normal vector $\mathbf{n}$ is approximated by $\mathbf{n}_h = \nabla\psi_h/\|\nabla\psi_h\|$. This and the approximate quadrature rules for the integration over Γ_h are often referred to as the "geometrical error". Luckily, one can show that the method converges optimally for the model problem if the same element space V_h is employed for the discrete functions and for the interpolation of the level set function as if the exact domain would have been used. Furthermore, deal.II allows us to independently choose a more accurate geometry representation with a higher-order level set function, compared to the function space for solving the Laplace–Beltrami equation.

The commented program

Include files

The first include files have all been treated in previous examples.

```
#include <deal.II/base/function.h>
#include <deal.II/base/convergence_table.h>
#include <deal.II/base/numbers.h>
#include <deal.II/base/point.h>
#include <deal.II/base/quadrature.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/tensor.h>
#include <deal.II/base/timer.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_interface_values.h>
#include <deal.II/fe/fe_nothing.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_update_flags.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/grid_tools_cache.h>
#include <deal.II/grid/tria.h>
#include <deal.II/hp/fe_collection.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/solver_control.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/sparsity_pattern.h>
#include <deal.II/lac/vector.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/lac/sparse_direct.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>
```

#include <fstream> #include <vector>

```
#include <deal.II/base/function_signed_distance.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/meshworker/scratch_data.h>
#include <deal.II/non_matching/fe_immersed_values.h>
#include <deal.II/non_matching/fe_values.h>
#include <deal.II/non_matching/mesh_classifier.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/distributed/grid_refinement.h>

#include <deal.II/lac/sparsity_tools.h>
#include <deal.II/lac/trilinos_vector.h>
#include <deal.II/lac/trilinos_precondition.h>
#include <deal.II/lac/trilinos_solver.h>
#include <deal.II/lac/trilinos_sparse_matrix.h>

using namespace dealii;
using VectorType = TrilinosWrappers::MPI::Vector;
using MatrixType = TrilinosWrappers::SparseMatrix;
```

We will grant to some cells empty finite element spaces FE::Nothing as done in **step-85**, but this time active DOFS will be only assigned to cell which are intersected by the surface approximation.

```
enum ActiveFEIndex
{
  lagrange = 0,
  nothing  = 1
};

namespace Step90
{
```

Exact surface

This class defines the surface using the implicit level-set representation.

```
  template <int dim>
  class TamarindShape : public Function<dim>
  {
  public:
```

TamarindShape(){};

```
    double value(const Point<dim> &point,
                 const unsigned int /*component*/ = 0) const override
    {
      return 0.25 * std::pow(point[0], 2) + std::pow(point[1], 2) +
             4.0 * std::pow(point[2], 2) *
```

```
               std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -2) -
             1.0;
    }
```

The exact surface normal uses the Cartesian gradient of the level-set function.

```
    Tensor<1, dim> gradient(const Point<dim>  &point,
                            const unsigned int component = 0) const override
    {
      AssertIndexRange(component, this->n_components);
      (void)component;

      Tensor<1, dim> grad;
      grad[0] = 0.5 * point[0] +
                (-2.0) * 4.0 * std::pow(point[2], 2) *
                  std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
                  (0.5 * numbers::PI * std::cos(numbers::PI * point[0]));
      grad[1] = 2.0 * point[1];
      grad[2] = (2.0) * 4.0 * point[2] *
                std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -2);
      return grad;
    }
```

The exact hessian is needed for the construction of the test case only.

```
    SymmetricTensor<2, dim>
    hessian(const Point<dim>  &point,
            const unsigned int component = 0) const override
    {
      AssertIndexRange(component, this->n_components);
      (void)component;
      SymmetricTensor<2, dim> hess;

      hess[0][0] =
        0.5 +
        (-2.0) * 4.0 * std::pow(point[2], 2) *
          ((-3.0) * std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -4) *
             std::pow(0.5 * numbers::PI * std::cos(numbers::PI * point[0]), 2) +
           std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) * (-1.0) *
             (0.5 * numbers::PI * numbers::PI *
              std::sin(numbers::PI * point[0])));
      hess[0][1] = 0.0;
      hess[0][2] = (-2.0) * (2.0) * 4.0 * point[2] *
                   std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
                   (0.5 * numbers::PI * std::cos(numbers::PI * point[0]));
      hess[1][0] = 0.0;
      hess[1][1] = 2.0;
      hess[1][2] = 0.0;

      hess[2][0] = (-2.0) * (2.0) * 4.0 * point[2] *
                   std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
                   0.5 * numbers::PI * std::cos(numbers::PI * point[0]);
      hess[2][1] = 0.0;
      hess[2][2] = (2.0) * 4.0 *
                   std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -2);
      return hess;
    }
  };

  template <int dim>
  class AnalyticalSolution : public Function<dim>
  {
    TamarindShape<dim> tamarind;

  public:
    double value(const Point<dim>  &point,
                 const unsigned int component = 0) const override;
```

Note that the next method returns the surface gradient $\nabla_\Gamma u$ of the exact solution.

```
    Tensor<1, dim> gradient(const Point<dim>  &point,
                            const unsigned int component = 0) const override;
  };
```

The exact solution $u = xy$. Note that it may be evaluated away from Γ as any other function of Cartesian points.

```
  template <int dim>
  double AnalyticalSolution<dim>::value(const Point<dim>  &point,
                                        const unsigned int component) const
  {
    AssertIndexRange(component, this->n_components);
    (void)component;
    return point[0] * point[1];
  }

  template <int dim>
  Tensor<1, dim>
  AnalyticalSolution<dim>::gradient(const Point<dim>  &point,
                                    const unsigned int component) const
  {
    AssertIndexRange(component, this->n_components);
    (void)component;
    const Tensor<1, dim> grad   = tamarind.gradient(point, component);
    const Tensor<1, dim> normal = (1.0 / grad.norm()) * grad;
    Tensor<1, dim>       projector_first_column = (-normal[0]) * normal;
    projector_first_column[0] += 1.0;
    Tensor<1, dim> projector_second_column = (-normal[1]) * normal;
    projector_second_column[1] += 1.0;
    Tensor<1, dim> surface_gradient =
      point[1] * projector_first_column + point[0] * projector_second_column;
    return surface_gradient;
  }
```

Exact forcing

```
  template <int dim>
  class RightHandSide : public Function<dim>
  {
    TamarindShape<dim> tamarind;

  public:
    virtual double value(const Point<dim>  &p,
                             const unsigned int component = 0) const override;
  };
```

A calculation of the surface Laplacian for a manufactured solution u provides with the exact forcing $f = \nabla_\Gamma u + u$.

```
  template <int dim>
  double RightHandSide<dim>::value(const Point<dim>  &point,
                                   const unsigned int component) const
  {
    AssertIndexRange(component, this->n_components);
    (void)component;
    const Tensor<1, dim>          grad   = tamarind.gradient(point, component);
    const Tensor<1, dim>          normal = (1.0 / grad.norm()) * grad;
    const SymmetricTensor<2, dim> hess   = tamarind.hessian(point, component);

    double mean_curv = 0.0;
    for (int j = 0; j < 3; j++)
      for (int k = 0; k < 3; k++)
        mean_curv += (1.0) * (1.0 / grad.norm()) *
                     ((j == k ? 1 : 0) - normal[j] * normal[k]) * hess[j][k];
    return point[0] * point[1] + 2.0 * normal[0] * normal[1] +
           mean_curv * (point[1] * normal[0] + point[0] * normal[1]);
  }
```

Scratch object for TraceFEM

Since the assembly procedure will be performed via **MeshWorker**, we need a Scratch object that handles the Non-Matching **FEValues** effectively.

```
  template <int dim>
  struct TraceFEM_ScratchData
  {
```

The meaning of the input arguments of this constructor is given in the solver class below.

```
    TraceFEM_ScratchData(
      const Mapping<dim>                     &mapping,
      const hp::FECollection<dim>            &fe_collection,
      const NonMatching::MeshClassifier<dim> &mesh_classifier,
      const DoFHandler<dim>                  &level_set_dof_handler,
      const VectorType                       &level_set,
      const NonMatching::RegionUpdateFlags    nonmatching_update_flags,
      const Quadrature<dim>                  &quadrature,
      const Quadrature<dim - 1>              &quadrature_face,
      const Quadrature<1>                    &quadrature_edge,
      const UpdateFlags face_update_flags = update_values | update_gradients |
                                            update_quadrature_points |
                                            update_JxW_values |
                                            update_normal_vectors,
      const UpdateFlags cell_update_flags = update_values | update_gradients |
                                            update_quadrature_points |
                                            update_JxW_values)
      : fe_values(mapping,
                  fe_collection[ActiveFEIndex::lagrange],
                  quadrature,
                  cell_update_flags)
      , fe_interface_values(mapping,
                            fe_collection[ActiveFEIndex::lagrange],
                            quadrature_face,
                            face_update_flags)
      , region_update_flags(nonmatching_update_flags)
      , quadrature_1D(quadrature_edge)
      , fe_collection(fe_collection)
      , mesh_classifier(mesh_classifier)
      , level_set_dof_handler(level_set_dof_handler)
      , level_set(level_set)
      , level_set_fe_values(mapping,
                            level_set_dof_handler.get_fe(),
                            quadrature,
                            cell_update_flags)
      , non_matching_fe_values(fe_collection,
                               quadrature_edge,
                               nonmatching_update_flags,
                               mesh_classifier,
                               level_set_dof_handler,
                               level_set)
    {}

    TraceFEM_ScratchData(const TraceFEM_ScratchData<dim> &scratch_data)
      : fe_values(scratch_data.fe_values.get_mapping(),
                  scratch_data.fe_values.get_fe(),
                  scratch_data.fe_values.get_quadrature(),
                  scratch_data.fe_values.get_update_flags())
      , fe_interface_values(scratch_data.fe_interface_values.get_mapping(),
                            scratch_data.fe_interface_values.get_fe(),
                            scratch_data.fe_interface_values.get_quadrature(),
                            scratch_data.fe_interface_values.get_update_flags())
      , region_update_flags(scratch_data.region_update_flags)
      , quadrature_1D(scratch_data.quadrature_1D)
      , fe_collection(scratch_data.fe_collection)
      , mesh_classifier(scratch_data.mesh_classifier)
      , level_set_dof_handler(scratch_data.level_set_dof_handler)
      , level_set(scratch_data.level_set)
      , level_set_fe_values(scratch_data.level_set_fe_values.get_mapping(),
                            scratch_data.level_set_fe_values.get_fe(),
                            scratch_data.level_set_fe_values.get_quadrature(),
                            scratch_data.level_set_fe_values.get_update_flags())
      , non_matching_fe_values(fe_collection,
                               quadrature_1D,
                               region_update_flags,
                               mesh_classifier,
                               level_set_dof_handler,
                               level_set)
    {}
```

The following objects are standard for computations on cells and their faces. In TraceFEM, we need these fitted quadratures in addition to **NonMatching FEValues**, which takes care of unfitted ones, mainly due to the stabilization terms.

```
    FEValues<dim>          fe_values;
    FEInterfaceValues<dim> fe_interface_values;

    const NonMatching::RegionUpdateFlags    region_update_flags;
```

```
      const Quadrature<1>                    &quadrature_1D;
      const hp::FECollection<dim>            &fe_collection;
      const NonMatching::MeshClassifier<dim> &mesh_classifier;
      const DoFHandler<dim>                  &level_set_dof_handler;
      const VectorType                       &level_set;
```

In addition to **FEValues** an unfitted quadrature for the main solution, we need a cell quadrature for the FE space of the level-set.

```
      FEValues<dim>              level_set_fe_values;
      NonMatching::FEValues<dim> non_matching_fe_values;
    };
```

Copy objects for TraceFEM

This CopyData object is customized for TraceFEM. In particular, the implementation of the normal-gradient volume stabilization relies on it.

```
    template <int dim>
    struct TraceFEM_CopyData
    {
      FullMatrix<double>                   cell_matrix;
      Vector<double>                       cell_rhs;
      std::vector<types::global_dof_index> local_dof_indices;
      unsigned int                         cell_index;
      double                               cell_L2_error_sqr;
      double                               cell_H1_error_sqr;
      double                               cell_stab_error_sqr;

      template <class Iterator>
      void reinit(const Iterator &cell, const unsigned int dofs_per_cell)
      {
        cell_matrix.reinit(dofs_per_cell, dofs_per_cell);
        cell_rhs.reinit(dofs_per_cell);
        local_dof_indices.resize(dofs_per_cell);
        cell->get_dof_indices(local_dof_indices);
      }

      template <class Iterator>
      void reinit(const Iterator &cell)
      {
        cell_index          = cell->active_cell_index();
        cell_L2_error_sqr   = 0;
        cell_H1_error_sqr   = 0;
        cell_stab_error_sqr = 0;
      }
    };
```

Normal-gradient stabilization form of TraceFEM

The following class corresponds to the stabilization form, its contribution to the global matrix and to the error.

```
    template <int dim>
    class NormalGradientVolumeStabilization
    {
    public:
      NormalGradientVolumeStabilization(VectorType &solution,
                                        VectorType &level_set)
        : solution(solution)
        , level_set(level_set)
        , stabilization_parameter(1.0)
        , stabilization_exponent(-1.0)
      {}
```

Since the stabilization form contributes to the error computation, we pass the discrete solution and the discrete level-set to the class which refer to different dof-handlers.

```
      VectorType &solution;
      VectorType &level_set;
```

Clearly, the next method indicates that the assembly and the evaluation of the form does require a bulk cell quadrature. This methodology may be utilized in the **MeshWorker**.

```
      bool needs_cell_worker()
      {
        return true;
      }
```

We define the stabilization form here assuming that ScratchData and CopyData arguments are initialized properly. The local contribution of the stabilization on this cell is defined.

```
      template <class Iterator>
      void cell_worker_assemble(const Iterator            &cell,
                                TraceFEM_ScratchData<dim> &scratch_data,
                                TraceFEM_CopyData<dim>    &copy_data)
      {
        const FEValues<dim> &fe_values = scratch_data.fe_values;
        const FEValues<dim> &level_set_fe_values =
          scratch_data.level_set_fe_values;
```

These points and weights are for the bulk cell quadrature that is suitable for the discrete level-set.

```
        const std::vector<double> &cellJxW = fe_values.get_JxW_values();
```

Similarly, the gradients of the discrete level-set are computed in the bulk cell quadrature, which, upon normalization, give the discrete normal vector in a bulk cell.

```
        std::vector<Tensor<1, dim>> grad_level_set(
          level_set_fe_values.get_quadrature().size());
        level_set_fe_values.get_function_gradients(level_set, grad_level_set);

        double factor =
          stabilization_parameter *
          std::pow(cell->minimum_vertex_distance(), stabilization_exponent);
        for (const unsigned int q : fe_values.quadrature_point_indices())
          {
```

```
          const Tensor<1, dim> &normal =
            (1.0 / grad_level_set[q].norm()) * grad_level_set[q];
```

Finally, the CopyData matrix computes the cell contribution

```
          for (const unsigned int i : fe_values.dof_indices())
            for (const unsigned int j : fe_values.dof_indices())
              copy_data.cell_matrix(i, j) +=
                factor * (normal * fe_values.shape_grad(i, q)) *
                (normal * fe_values.shape_grad(j, q)) * cellJxW[q];
        }
    }
```

Similarly, we evaluate the stabilization form on this cell.

```
    template <class Iterator>
    void cell_worker_evaluate(const Iterator             &cell,
                              TraceFEM_ScratchData<dim> &scratch_data,
                              TraceFEM_CopyData<dim>    &copy_data)
    {
      double                     cell_stab_error_sqr = 0.0;
      const FEValues<dim>       &fe_values           = scratch_data.fe_values;
      const std::vector<double> &cellJxW = fe_values.get_JxW_values();
      const unsigned int n_q_points = fe_values.get_quadrature_points().size();
      const FEValues<dim> &level_set_fe_values =
        scratch_data.level_set_fe_values;

      std::vector<Tensor<1, dim>> level_set_grad(n_q_points);
      level_set_fe_values.get_function_gradients(level_set, level_set_grad);

      std::vector<Tensor<1, dim>> sol_grad(n_q_points);
      fe_values.get_function_gradients(solution, sol_grad);

      double factor =
        stabilization_parameter *
        std::pow(cell->minimum_vertex_distance(), stabilization_exponent);

      for (const unsigned int q : fe_values.quadrature_point_indices())
        {
          const Tensor<1, dim> normal =
            (1.0 / level_set_grad[q].norm()) * level_set_grad[q];

          const double error_at_point = normal * sol_grad[q];
          cell_stab_error_sqr +=
            factor * std::pow(error_at_point, 2.0) * cellJxW[q];
        }
      copy_data.cell_stab_error_sqr = cell_stab_error_sqr;
    }

  private:
```

These parameters define how the stabilization is scaled, Ch^{α}. For elliptic problems, $C \simeq 1$ and $-1 \leq \alpha \leq 1$.

```
    double stabilization_parameter;
    double stabilization_exponent;
  };
```

Laplace–Beltrami solver

The main class whose method **run()** performs the computation. The following methods are discussed and their definitions are given below.

```
  template <int dim>
  class LaplaceBeltramiSolver
  {
  public:
    LaplaceBeltramiSolver();
    void run();

  private:
    void make_grid();

    void localize_surface();

    void postprocess_level_set();

    void setup_discrete_level_set();

    void distribute_dofs();

    void initialize_matrices();

    void assemble();

    void solve();

    void mark_intersected();

    void refine();

    void evaluate_errors();

    void output_level_set(unsigned int);

    void output_solution();

    void display_results();

    MPI_Comm mpi_communicator;
```

The surface of interest corresponds to the zero contour of the following exact level-set function Function<dim> *exact_level_set;

```
    TamarindShape<dim> Tamarind;
```

The manufacture solution to the Laplace–Beltrami problem and the corresponding right-hand side.

```
    const AnalyticalSolution<dim> analytical_solution;
    const RightHandSide<dim>      right_hand_side;
```

There is single triangulation which is shared by the discretizations of the solution and of the level-set.

```
    parallel::distributed::Triangulation<dim, dim> triangulation;

    ConditionalOStream pcout;
    TimerOutput        computing_timer;
```

However, the degrees of their FE spaces may be different

```
    const unsigned int fe_degree;
    const unsigned int level_set_fe_degree;
```

The first two quadratures are for the bulk and face integration, typical for TraceFEM stabilizations, while the integration over implicit surface is based on the last, one-dimensional rule.

```
    const QGauss<dim>     cell_quadrature;
    const QGauss<dim - 1> face_quadrature;
    const QGauss<1>       quadrature_1D;
```

We need three separate FE spaces. The first manages the TraceFEM space which is active on intersected element. The second manages the discrete level set function that describes the geometry of the surface. The third handles a piecewise constant space which is used solely for the accumulation of refinement indicators and of errors.

```
    hp::FECollection<dim> fe_collection;
    const FE_Q<dim>       level_set_fe;
```

The corresponding DOF-handlers are given by

```
    DoFHandler<dim> dof_handler;
    DoFHandler<dim> level_set_dof_handler;
```

Since we will adaptively refine the bulk triangulation, two constraints are needed: one for the solution space and another for the level-set space.

```
    AffineConstraints<double> constraints;
    AffineConstraints<double> level_set_constraints;
```

Discrete vectors initialized with dof_handler and level_set_dof_handler.

```
    VectorType    completely_distributed_solution;
    VectorType    locally_relevant_solution;
    VectorType    locally_relevant_exact;
    VectorType    level_set;
    Vector<float> activeFE_indicator;
```

Mesh_classifier separates intersected elements and non-intersected ones in the fe_collection

```
    NonMatching::MeshClassifier<dim> mesh_classifier;
    const MappingQ1<dim>             mapping;
```

Any TraceFEM need a stabilization, and we choose the normal-gradient, volume stabilization.

```
    NormalGradientVolumeStabilization<dim> stabilization_scheme;
```

Discrete right-hand side and the final matrix correspondent to dof_handler.

```
    VectorType global_rhs;
    MatrixType global_matrix;

    SparsityPattern sparsity_pattern;

    IndexSet locally_owned_dofs;
    IndexSet locally_relevant_dofs;
```

Depending on the type of the quadrature, surface, face or volume, we need to define different update flags.

```
    NonMatching::RegionUpdateFlags surface_update_flags;
    UpdateFlags                    face_update_flags;
    UpdateFlags                    cell_update_flags;
```

The following displays the errors of the convergence test.

```
    ConvergenceTable convergence_table;
    unsigned int     number_of_iterations;
    double           average;
    double           area;
    double           error_L2_sqr;
    double           error_H1_sqr;
    double           error_stab_sqr;

    int number_of_proc;
  };
```

The constructor of the solver

One may adjust the parameters of TraceFEM here.

```
  template <int dim>
  LaplaceBeltramiSolver<dim>::LaplaceBeltramiSolver()
    : mpi_communicator(MPI_COMM_WORLD)
    , triangulation(
        mpi_communicator,
        typename Triangulation<dim>::MeshSmoothing(
          Triangulation<dim>::
            none //smoothing_on_refinement |  Triangulation<dim>::smoothing_on_coarsening
          ),
        parallel::distributed::Triangulation<dim, dim>::
          mesh_reconstruction_after_repartitioning)
    , pcout(std::cout,
            (Utilities::MPI::this_mpi_process(mpi_communicator) == 0))
    , computing_timer(mpi_communicator,
                      pcout,
                      TimerOutput::never,
                      TimerOutput::wall_times)
    , fe_degree(1)
    , level_set_fe_degree(1)
    , cell_quadrature(fe_degree + 1)
    , face_quadrature(fe_degree + 1)
    , quadrature_1D(fe_degree + 1)
    , level_set_fe(level_set_fe_degree)
    , dof_handler(triangulation)
    , level_set_dof_handler(triangulation)
    , mesh_classifier(level_set_dof_handler, level_set)
    , mapping()
    , stabilization_scheme(locally_relevant_solution, level_set)
  {
    fe_collection.push_back(FE_Q<dim>(fe_degree));
    fe_collection.push_back(FE_Nothing<dim>());
```

The update flags for the embedded surface integration, bulk and face integration on cells.

```
    surface_update_flags.surface =
      update_values | update_gradients | update_JxW_values |
      update_quadrature_points | update_normal_vectors;

    face_update_flags = update_default;
    cell_update_flags = update_default;
    MPI_Comm_size(mpi_communicator, &number_of_proc);
  }
```

Geometric approximation

The initial refinement helps the level set to approximate the surface meaningfully.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::make_grid()
  {
    pcout << std::endl
          << "Creating background mesh with MPI_Size=" << number_of_proc
          << std::endl;
    double cube_side = 2.0;
    GridGenerator::hyper_cube(triangulation, -cube_side, cube_side);
    triangulation.refine_global(3);
  }
```

The following procedure rules out the situation when the intersection of a cell and of the surface has a zero two-dimensional measure. In this situation, we slightly perturb the surface approximation to guarantee that all cuts have positive area. This is done in a postprocessing manner of the next method.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::postprocess_level_set()
  {
    const hp::FECollection<dim, dim> &level_set_fe(
      level_set_dof_handler.get_fe_collection());
    std::vector<types::global_dof_index> level_set_dofs_on_cell(
      level_set_fe.max_dofs_per_cell());
    for (const auto &level_set_cell :
         level_set_dof_handler.active_cell_iterators())
      {
        if (!level_set_cell->is_locally_owned())
          continue;
        const double cell_side_length =
          level_set_cell->minimum_vertex_distance();
        const auto n_dofs =
          level_set_fe[level_set_cell->active_fe_index()].n_dofs_per_cell();
        level_set_dofs_on_cell.resize(n_dofs);
        level_set_cell->get_dof_indices(level_set_dofs_on_cell);
        for (unsigned int i = 0; i < n_dofs; ++i)
          {
            if (std::abs(level_set[level_set_dofs_on_cell[i]]) <
                cell_side_length * 1e-12)
              level_set[level_set_dofs_on_cell[i]] = cell_side_length * 1e-12;
          }
      }
  }
```

Now we construct the discrete level-set and determine which cells are intersected.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::setup_discrete_level_set()
  {
    pcout
      << "Setting up discrete level set function and reclassifying cells on MPI_rank=0... "
      << std::flush;
    TimerOutput::Scope t(computing_timer, "setup_level_set");
    Timer              timer;
```

Note that the piecewise constant FE space is independent of the mesh classification, and that all cells, intersected and non-intersected, have a correspondent value in the activeFE_indicator

```
    activeFE_indicator.reinit(triangulation.n_active_cells());
```

Similarly, the exact level-set function is approximated on the whole triangulation and postprocessed afterward resulting in a surface approximation with no gaps

```
    level_set_dof_handler.distribute_dofs(level_set_fe);
```

Filling in the constaints for the level_set

```
    level_set_constraints.clear();
```

```
      IndexSet level_set_locally_relevant_dofs;
      DoFTools::extract_locally_relevant_dofs(level_set_dof_handler,
                                              level_set_locally_relevant_dofs);
      level_set_constraints.reinit(level_set_locally_relevant_dofs);
      DoFTools::make_hanging_node_constraints(level_set_dof_handler,
                                              level_set_constraints);
      level_set_constraints.close();

      VectorType tmp_sol;
      tmp_sol.reinit(level_set_dof_handler.locally_owned_dofs(),
                     mpi_communicator);
```

The only place where the geometric information enters the code

```
      VectorTools::interpolate(level_set_dof_handler, Tamarind, tmp_sol);

      level_set.reinit(level_set_locally_relevant_dofs,
                       level_set_dof_handler.locally_owned_dofs(),
                       mpi_communicator);
      level_set_constraints.distribute(tmp_sol);
      level_set = tmp_sol;

      postprocess_level_set();
```

Using the discrete level-set, mark the cell which are intersected by it.

```
      mesh_classifier.reclassify();
```

Once the triangulation's cells are classified, we can determine which cells are active.

```
      for (const auto &cell : dof_handler.active_cell_iterators())
        {
          if (!cell->is_locally_owned())
            continue;
          if (mesh_classifier.location_to_level_set(cell) ==
              NonMatching::LocationToLevelSet::intersected)
            cell->set_active_fe_index(ActiveFEIndex::lagrange);
          else
            cell->set_active_fe_index(ActiveFEIndex::nothing);
        }
      timer.stop();
      pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    }
```

The method fills in the indicator telling which cells are intersected. It is used in the adaptive refinement near the surface.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::mark_intersected()
    {
      pcout << "Determining cells with active FE index on MPI_rank=0..."
            << std::flush;
      Timer timer;
      for (const auto &cell : dof_handler.active_cell_iterators())
        {
          if (!cell->is_locally_owned())
            continue;
          if (mesh_classifier.location_to_level_set(cell) ==
              NonMatching::LocationToLevelSet::intersected)
            activeFE_indicator[cell->active_cell_index()] = 1.0;
        }
      timer.stop();
      pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    }
```

We refine only intersected cells with activeFE_indicator=1.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::refine()
    {
      Timer              timer;
      TimerOutput::Scope t(computing_timer, "refine");
      pcout << "Refining near surface on MPI_rank=0"
            << "... " << std::flush;
```

Note that refining by number would not be useful here because the number of non-intersected cells also grows interfering with the number of active, intersected cells.

```
      parallel::distributed::GridRefinement::refine_and_coarsen_fixed_fraction(
        triangulation, activeFE_indicator, 1.0, 0.0);

      triangulation.execute_coarsening_and_refinement();
      timer.stop();
      pcout << "took (" << timer.wall_time() << "s)" << std::endl << std::endl;
    }
```

As the surface is properly approximated by several adaptive steps, we may now distribute the degrees of freedom to cells which are intersected by the discrete approximation.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::distribute_dofs()
    {
      pcout << "Distributing degrees of freedom on MPI_rank=0... " << std::flush;
      Timer timer;
      dof_handler.distribute_dofs(fe_collection);
      locally_owned_dofs = dof_handler.locally_owned_dofs();
      locally_relevant_dofs =
        DoFTools::extract_locally_relevant_dofs(dof_handler);
      completely_distributed_solution.reinit(dof_handler.locally_owned_dofs(),
                                             mpi_communicator);
      locally_relevant_solution.reinit(locally_owned_dofs,
                                       locally_relevant_dofs,
                                       mpi_communicator);
      global_rhs.reinit(locally_owned_dofs, mpi_communicator);

      timer.stop();
      pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    }
```

Next, we initialize matrices for active DOFS and apply the constraints for the solution.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::initialize_matrices()
  {
    pcout << "Initializing the matricex on MPI_rank=0... " << std::flush;
    Timer                  timer;
    DynamicSparsityPattern dsp(dof_handler.n_dofs(), dof_handler.n_dofs());
    constraints.clear();
    constraints.reinit(locally_relevant_dofs);

    DoFTools::make_hanging_node_constraints(dof_handler, constraints);
    constraints.close();
    DoFTools::make_sparsity_pattern(dof_handler, dsp, constraints);

    SparsityTools::distribute_sparsity_pattern(dsp,
                                               locally_owned_dofs,
                                               mpi_communicator,
                                               locally_relevant_dofs);
    global_matrix.reinit(locally_owned_dofs,
                         locally_owned_dofs,
                         dsp,
                         mpi_communicator);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }
```

Assembly and surface accumulation

We use a **MeshWorker** to assemble the linear problem efficiently.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::assemble()
  {
    pcout << "Assembling on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "assembly");
    Timer              timer;
    using Iterator         = typename DoFHandler<dim>::active_cell_iterator;
    const auto cell_worker = [&](const Iterator            &cell,
                                 TraceFEM_ScratchData<dim> &scratch_data,
                                 TraceFEM_CopyData<dim>    &copy_data) {
```

This cell worker does not do anything for non-intersected cells

```
      if (mesh_classifier.location_to_level_set(cell) ==
            NonMatching::LocationToLevelSet::intersected &&
          cell->is_locally_owned())
        {
```

Once we now that the cell is intersected, we construct the unfitted quadratures for the solutions FE space on the cell.

```
          scratch_data.non_matching_fe_values.reinit(cell);
          copy_data.reinit(cell,
                           scratch_data.fe_values.get_fe().n_dofs_per_cell());
          copy_data.cell_matrix = 0;
          copy_data.cell_rhs    = 0;
          const std::optional<NonMatching::FEImmersedSurfaceValues<dim>>
            &surface_fe_values =
              scratch_data.non_matching_fe_values.get_surface_fe_values();
          const std::vector<double> &surfJxW =
            surface_fe_values->get_JxW_values();
```

The accumulation of the surface integrals, including the forcing, is performed here.

```
          for (unsigned int q : surface_fe_values->quadrature_point_indices())
            {
              const Tensor<1, dim> &normal =
                surface_fe_values->normal_vector(q);

              for (const unsigned int i : surface_fe_values->dof_indices())
                {
                  copy_data.cell_rhs(i) +=
                    right_hand_side.value(
                      surface_fe_values->quadrature_point(q)) *
                    surface_fe_values->shape_value(i, q) * surfJxW[q];

                  for (const unsigned int j : surface_fe_values->dof_indices())
                    {
```

This is the local contribution to the global linear system

```
                      copy_data.cell_matrix(i, j) +=
                        1.0 *
                        (surface_fe_values->shape_value(i, q) *
                         surface_fe_values->shape_value(j, q)) *
                        surfJxW[q];
                      copy_data.cell_matrix(i, j) +=
                        (1.0 *
                         (surface_fe_values->shape_grad(i, q) -
                          (normal * surface_fe_values->shape_grad(i, q)) *
                            normal) *
                         (surface_fe_values->shape_grad(j, q) -
                          (normal * surface_fe_values->shape_grad(j, q)) *
                            normal)) *
                        surfJxW[q];
                    }
                }
            }
```

The normal-gradient volume stabilization form needs a bulk cell integration while other types of stabilization may need face quadratures, for example. So we check it first.

```
          if (stabilization_scheme.needs_cell_worker())
            {
```

The cell was provided by the solution's DOF-handler, so we recast it as a level-set's DOF-handler cell. However, it is the same geometric entity of the common triangulation.

```
              typename DoFHandler<dim>::active_cell_iterator level_set_cell(
                &(triangulation),
                cell->level(),
                cell->index(),
                &level_set_dof_handler);
```

Once we know what type of quadratures are required for this stabilization scheme, we **reinit()** them in the TraceFEM ScratchData, before acquiring the local matrix contribution in the CopyData.

```
              scratch_data.fe_values.reinit(cell);
              scratch_data.level_set_fe_values.reinit(level_set_cell);
              stabilization_scheme.cell_worker_assemble(cell,
                                                        scratch_data,
                                                        copy_data);
            }
        }
    };
```

This copier worker distributes the local contributions from the CopyData taking into account the constraints.

```
    const auto copier = [&](const TraceFEM_CopyData<dim> &c) {
      constraints.distribute_local_to_global(c.cell_matrix,
                                             c.cell_rhs,
                                             c.local_dof_indices,
                                             global_matrix,
                                             global_rhs);
    };
```

Finally, we set up the TraceFEM ScratchData with the discrete level-set, classifier and proper quadrature rules.

```
    TraceFEM_ScratchData<dim> scratch_data(mapping,
                                           fe_collection,
                                           mesh_classifier,
                                           level_set_dof_handler,
                                           level_set,
                                           surface_update_flags,
                                           cell_quadrature,
                                           face_quadrature,
                                           quadrature_1D,
                                           face_update_flags);
    TraceFEM_CopyData<dim>    copy_data;
```

The **MeshWorker** goes over all cells provided by the solutions' DOF-handler. Note that this includes non-intersected cells as well, but the cell worker does nothing on them.

```
    MeshWorker::mesh_loop(dof_handler.begin_active(),
                          dof_handler.end(),
                          cell_worker,
                          copier,
                          scratch_data,
                          copy_data,
                          MeshWorker::assemble_own_cells);

    global_matrix.compress(VectorOperation::add);
    global_rhs.compress(VectorOperation::add);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }
```

The straighforward direct solver is called in the following method.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::solve()
  {
    TimerOutput::Scope t(computing_timer, "solve");
    Timer              timer;
    bool               apply_direct_solver = false;
    if (apply_direct_solver)
      {
        pcout << "Solving directly on MPI_rank=0... " << std::flush;
        SolverControl solver_control(2000, 1e-8);
        TrilinosWrappers::SolverDirect::AdditionalData data;
        TrilinosWrappers::SolverDirect trilinos(solver_control, data);
        trilinos.solve(global_matrix,
                       completely_distributed_solution,
                       global_rhs);
        number_of_iterations = -1;
      }
    else
      {
        Timer timer;
        pcout << "Solving with AMG on MPI_rank=0... " << std::flush;
        const unsigned int             max_iterations = dof_handler.n_dofs();
        SolverControl                  solver_control(max_iterations);
        std::vector<std::vector<bool>> constant_modes;
        DoFTools::extract_constant_modes(dof_handler,
                                         ComponentMask(),
                                         constant_modes);
        TrilinosWrappers::PreconditionAMG preconditioner_stiffness;
        TrilinosWrappers::PreconditionAMG::AdditionalData Amg_data;
        Amg_data.constant_modes        = constant_modes;
        Amg_data.elliptic              = true;
        Amg_data.higher_order_elements = false;
        Amg_data.smoother_sweeps       = 2;
        Amg_data.aggregation_threshold = 0.02;
        Amg_data.output_details        = true;
        preconditioner_stiffness.initialize(global_matrix);

        SolverCG<VectorType> cg(solver_control);
        cg.solve(global_matrix,
                 completely_distributed_solution,
                 global_rhs,
                 preconditioner_stiffness);
        pcout << "required " << solver_control.last_step() << " iterations and "
              << std::flush;
        number_of_iterations = solver_control.last_step();
      }
    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    constraints.distribute(completely_distributed_solution);
    locally_relevant_solution = completely_distributed_solution;
  }
```

Similarly to the assembly(), a **MeshWorker** is used to accumulate errors

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::evaluate_errors()
  {
    pcout << "Evaluating errors on the surface on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "eval_errors");
    Timer              timer;
    error_L2_sqr         = 0.0;
    error_H1_sqr         = 0.0;
    error_stab_sqr       = 0.0;
    const auto cell_worker = [&](const auto &cell,
                                 auto       &scratch_data,
                                 auto       &copy_data) {
      if (mesh_classifier.location_to_level_set(cell) ==
            NonMatching::LocationToLevelSet::intersected &&
          cell->is_locally_owned())
        {
          double cell_L2_error_sqr = 0.0;
          double cell_H1_error_sqr = 0.0;

          copy_data.reinit(cell);
          scratch_data.non_matching_fe_values.reinit(cell);

          const std::optional<NonMatching::FEImmersedSurfaceValues<dim>>
            &surface_fe_values =
              scratch_data.non_matching_fe_values.get_surface_fe_values();
          const std::vector<double> &surfJxW =
            surface_fe_values->get_JxW_values();
          int n_q_points = surface_fe_values->n_quadrature_points;

          std::vector<double> sol(n_q_points);
          surface_fe_values->get_function_values(locally_relevant_solution,
                                                 sol);

          std::vector<Tensor<1, dim>> sol_grad(n_q_points);
          surface_fe_values->get_function_gradients(locally_relevant_solution,
                                                    sol_grad);

          for (const unsigned int q :
               surface_fe_values->quadrature_point_indices())
            {
              const Point<dim> &point = surface_fe_values->quadrature_point(q);
              const Tensor<1, dim> &normal =
                surface_fe_values->normal_vector(q);
              const double error_at_point =
                sol.at(q) - analytical_solution.value(point);
              const Tensor<1, dim> vector_error_at_point =
                (sol_grad.at(q) - (normal * sol_grad.at(q)) * normal -
                 analytical_solution.gradient(point));

              cell_L2_error_sqr += std::pow(error_at_point, 2) * surfJxW[q];
              cell_H1_error_sqr +=
                vector_error_at_point * vector_error_at_point * surfJxW[q];
            }
          copy_data.cell_L2_error_sqr = cell_L2_error_sqr;
          copy_data.cell_H1_error_sqr = cell_H1_error_sqr;

          if (stabilization_scheme.needs_cell_worker())
            {
              typename DoFHandler<dim>::active_cell_iterator level_set_cell(
                &(triangulation),
                cell->level(),
                cell->index(),
                &level_set_dof_handler);
              scratch_data.fe_values.reinit(cell);
              scratch_data.level_set_fe_values.reinit(level_set_cell);
              stabilization_scheme.cell_worker_evaluate(cell,
                                                        scratch_data,
                                                        copy_data);
            }
        }
    };
```

The copier worker checks if the CopyData is initialized meaningfully in the cell worker for an intersected cell.

```
    const auto copier = [&](const auto &copy_data) {
      if (copy_data.cell_index < activeFE_indicator.size())
        {
          error_L2_sqr += copy_data.cell_L2_error_sqr;
          error_H1_sqr += copy_data.cell_H1_error_sqr;
          error_stab_sqr += copy_data.cell_stab_error_sqr;
        }
    };
```

The rest is equivalent to assembly().

```
    TraceFEM_ScratchData<dim> scratch_data(mapping,
                                           fe_collection,
                                           mesh_classifier,
                                           level_set_dof_handler,
                                           level_set,
                                           surface_update_flags,
                                           cell_quadrature,
                                           face_quadrature,
                                           quadrature_1D,
                                           face_update_flags);
    TraceFEM_CopyData<dim>    copy_data;

    MeshWorker::mesh_loop(dof_handler.begin_active(),
                          dof_handler.end(),
                          cell_worker,
                          copier,
                          scratch_data,
                          copy_data,
                          MeshWorker::assemble_own_cells);
    timer.stop();
```

```
      pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    }
```

The method collects the results, and prints it out.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::display_results()
    {
      const double error_L2 =
        std::sqrt(Utilities::MPI::sum(error_L2_sqr, mpi_communicator));
      const double error_semiH1 =
        std::sqrt(Utilities::MPI::sum(error_H1_sqr, mpi_communicator));
      const double error_stab =
        std::sqrt(Utilities::MPI::sum(error_stab_sqr, mpi_communicator));

      const int dof_handler_size           = dof_handler.n_dofs();
      const int level_set_dof_handler_size = level_set_dof_handler.n_dofs();
```

const double time = Utilities::MPI::max(solve_time, mpi_communicator);

```
      const double iterations =
        Utilities::MPI::max(number_of_iterations, mpi_communicator);

      convergence_table.add_value("LevelSet dofs", level_set_dof_handler_size);
      convergence_table.evaluate_convergence_rates(
        "LevelSet dofs", ConvergenceTable::reduction_rate_log2);

      convergence_table.add_value("Active dofs", dof_handler_size);
      convergence_table.evaluate_convergence_rates(
        "Active dofs", ConvergenceTable::reduction_rate_log2);
```

convergence_table.add_value("CPU time", time); convergence_table.evaluate_convergence_rates("CPU time", **ConvergenceTable::reduction_rate_log2**);

```
      convergence_table.add_value("Iterations", iterations);

      convergence_table.add_value("L2 Error", error_L2);
      convergence_table.evaluate_convergence_rates(
        "L2 Error", ConvergenceTable::reduction_rate_log2);
      convergence_table.set_scientific("L2 Error", true);

      convergence_table.add_value("H1 error", error_semiH1);
      convergence_table.evaluate_convergence_rates(
        "H1 error", ConvergenceTable::reduction_rate_log2);
      convergence_table.set_scientific("H1 error", true);

      convergence_table.add_value("Stab norm", error_stab);
      convergence_table.evaluate_convergence_rates(
        "Stab norm", ConvergenceTable::reduction_rate_log2);
      convergence_table.set_scientific("Stab norm", true);

      pcout << std::endl;
      if (Utilities::MPI::this_mpi_process(mpi_communicator) == 0)
        convergence_table.write_text(pcout.get_stream());
    }
```

The method performs VTK output of preliminary mesh refinements of geometry approximation.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::output_level_set(unsigned int n)
    {
      pcout << "Writing vtu file for surface on MPI_rank=0... " << std::flush;
      TimerOutput::Scope t(computing_timer, "output_level_set");
      Timer              timer;
      DataOut<dim>       data_out;
      data_out.add_data_vector(level_set_dof_handler, level_set, "level_set");
      data_out.add_data_vector(activeFE_indicator, "ref_indicator");
      data_out.build_patches();

      data_out.write_vtu_in_parallel("surface_" + std::to_string(n) + ".vtu",
                                     mpi_communicator);

      timer.stop();
      pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    }
```

The method performs VTK output of the TraceFEM solution.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::output_solution()
    {
      pcout << "Writing vtu file on MPI_rank=0... " << std::flush;
      TimerOutput::Scope t(computing_timer, "output_solution");
      Timer              timer;
      Vector<double>     exact(dof_handler.locally_owned_dofs().size());

      VectorTools::interpolate(dof_handler, analytical_solution, exact);
      DataOut<dim> data_out;
      data_out.add_data_vector(dof_handler,
                               locally_relevant_solution,
                               "solution");
      data_out.add_data_vector(dof_handler, exact, "exact");
      data_out.add_data_vector(level_set_dof_handler, level_set, "level_set");
```

Here we filter out the non-intersected cells from the output. This is not neccessary for the solution to be output but saves considerable amount of storage.

```
      data_out.set_cell_selection(
        [this](const typename Triangulation<dim>::cell_iterator &cell) {
          return cell->is_active() && cell->is_locally_owned() &&
                 mesh_classifier.location_to_level_set(cell) ==
                   NonMatching::LocationToLevelSet::intersected;
        });
      data_out.build_patches();

      data_out.write_vtu_in_parallel("solution.vtu", mpi_communicator);

      timer.stop();
      pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    }
```

This method generates an efficient surface approximation by refining the cell intersected by the current surface approximation.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::localize_surface()
  {
    unsigned int preliminary_levels = 3;
    for (unsigned int localization_cycle = 0;
         localization_cycle < preliminary_levels;
         localization_cycle++)
      {
        pcout << std::endl
              << "Preliminary refinement #" << localization_cycle << std::endl;
        setup_discrete_level_set();
        mark_intersected();
        output_level_set(localization_cycle);
        refine();
      }
```

computing_timer.print_summary();

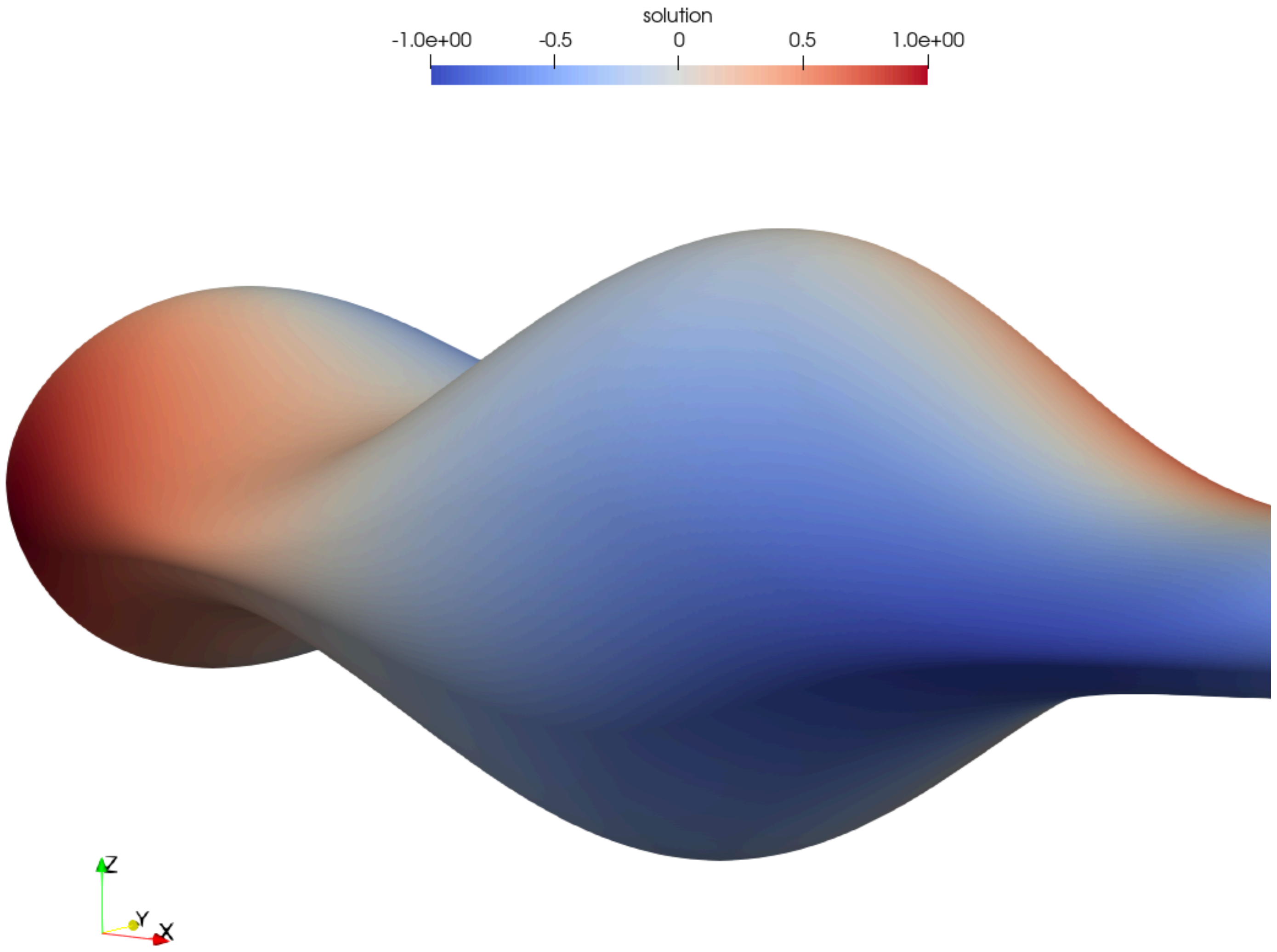
```
    computing_timer.reset();
  }
```

The main logic of the solver is presented here.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::run()
  {
    make_grid();
    localize_surface();
    const unsigned int convergence_levels = 3;
    for (unsigned int cycle = 0; cycle < convergence_levels; cycle++)
      {
        pcout << std::endl << "Convergence refinement #" << cycle << std::endl;
        setup_discrete_level_set();
        distribute_dofs();
        initialize_matrices();
        assemble();
        solve();
        evaluate_errors();
        display_results();
        if (cycle < convergence_levels - 1)
          {
            mark_intersected();
            refine();
          }
        else
          output_solution();
        computing_timer.print_summary();
        computing_timer.reset();
      }
  }
} // namespace Step90

int main(int argc, char *argv[])
{
  try
    {
      using namespace dealii;
      using namespace Step90;
      Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);
      LaplaceBeltramiSolver<3>         LB_solver;
      LB_solver.run();
    }
  catch (std::exception &exc)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Exception on processing: " << std::endl
                << exc.what() << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;

      return 1;
    }
  catch (...)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Unknown exception!" << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;
      return 1;
    }

  return 0;
}
```

Results

The numerical solution for a very finest mesh is shown below.



Convergence test

The results of the convergence study is shown in the table below.

Cycle	DOFS	EOC	Iterations	L^2-Error	EOC	H^1-Error	EOC	Stabilization	EOC
0	12370	-	15	7.6322e-02	-	3.6212e-01	-	2.2423e-01	-
1	49406	-2.00	18	1.1950e-02	2.68	1.4752e-01	1.30	1.1238e-01	1.00
2	196848	-1.99	19	1.7306e-03	2.79	7.4723e-02	0.98	6.1131e-02	0.88
3	785351	-2.00	22	3.6276e-04	2.25	3.9329e-02	0.93	3.0185e-02	1.02
4	3136501	-2.00	25	7.5910e-05	2.26	1.9694e-02	1.00	1.4875e-02	1.02
5	12536006	-2.00	26	1.7279e-05	2.14	9.8443e-03	1.00	7.4067e-03	1.01
6	50122218	-2.00	30	4.3891e-06	1.98	4.9219e-03	1.00	3.7042e-03	1.00

In this test we refine the mesh near the surface and, as a result, the number of degrees of freedom scales in the two-dimensional fashion. The optimal rates of error convergence in $L^2(\Gamma)$ and $H^1(\Gamma)$ norms are clearly observable. We also note the first order convergence of the stabilization term $s_h(u_h, u_h)$.

Parallel scalability

In progress...

The plain program

```
/* ---------------------------------------------------------------------
 *
 * Copyright (C) 2021 - 2024 by the deal.II authors
 *
 * This file is part of the deal.II library.
 *
 * The deal.II library is free software; you can use it, redistribute
 * it, and/or modify it under the terms of the GNU Lesser General
 * Public License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 * The full text of the license can be found in the file LICENSE.md at
 * the top level directory of deal.II.
 *
 * ---------------------------------------------------------------------
 * This program was contributed by Vladimir Yushutin and Timo Heister
 */



#include <deal.II/base/function.h>
#include <deal.II/base/convergence_table.h>
#include <deal.II/base/numbers.h>
#include <deal.II/base/point.h>
#include <deal.II/base/quadrature.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/tensor.h>
#include <deal.II/base/timer.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_interface_values.h>
#include <deal.II/fe/fe_nothing.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_update_flags.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/grid_tools_cache.h>
#include <deal.II/grid/tria.h>
#include <deal.II/hp/fe_collection.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/solver_control.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/sparsity_pattern.h>
#include <deal.II/lac/vector.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/lac/sparse_direct.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/base/function_signed_distance.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/meshworker/scratch_data.h>
#include <deal.II/non_matching/fe_immersed_values.h>
#include <deal.II/non_matching/fe_values.h>
#include <deal.II/non_matching/mesh_classifier.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/distributed/grid_refinement.h>

#include <deal.II/lac/sparsity_tools.h>
#include <deal.II/lac/trilinos_vector.h>
#include <deal.II/lac/trilinos_precondition.h>
#include <deal.II/lac/trilinos_solver.h>
#include <deal.II/lac/trilinos_sparse_matrix.h>

using namespace dealii;
using VectorType = TrilinosWrappers::MPI::Vector;
using MatrixType = TrilinosWrappers::SparseMatrix;

enum ActiveFEIndex
{
  lagrange = 0,
  nothing  = 1
};

namespace Step90
{
  template <int dim>
  class TamarindShape : public Function<dim>
  {
  public:
    double value(const Point<dim> &point,
                 const unsigned int /*component*/ = 0) const override
    {
      return 0.25 * std::pow(point[0], 2) + std::pow(point[1], 2) +
             4.0 * std::pow(point[2], 2) *
               std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -2) -
             1.0;
    }
    Tensor<1, dim> gradient(const Point<dim>  &point,
                            const unsigned int component = 0) const override
    {
      AssertIndexRange(component, this->n_components);
      (void)component;

      Tensor<1, dim> grad;
      grad[0] = 0.5 * point[0] +
                (-2.0) * 4.0 * std::pow(point[2], 2) *
                  std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
                  (0.5 * numbers::PI * std::cos(numbers::PI * point[0]));
      grad[1] = 2.0 * point[1];
      grad[2] = (2.0) * 4.0 * point[2] *
                std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -2);
      return grad;
    }
    SymmetricTensor<2, dim>
    hessian(const Point<dim>  &point,
            const unsigned int component = 0) const override
    {
      AssertIndexRange(component, this->n_components);
      (void)component;
      SymmetricTensor<2, dim> hess;

      hess[0][0] =
        0.5 +
        (-2.0) * 4.0 * std::pow(point[2], 2) *
          ((-3.0) * std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -4) *
             std::pow(0.5 * numbers::PI * std::cos(numbers::PI * point[0]), 2) +
           std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) * (-1.0) *
             (0.5 * numbers::PI * numbers::PI *
              std::sin(numbers::PI * point[0])));
      hess[0][1] = 0.0;
      hess[0][2] = (-2.0) * (2.0) * 4.0 * point[2] *
                   std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
                   (0.5 * numbers::PI * std::cos(numbers::PI * point[0]));
      hess[1][0] = 0.0;
      hess[1][1] = 2.0;
      hess[1][2] = 0.0;
      hess[2][0] = (-2.0) * (2.0) * 4.0 * point[2] *
                   std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
                   0.5 * numbers::PI * std::cos(numbers::PI * point[0]);
      hess[2][1] = 0.0;
      hess[2][2] = (2.0) * 4.0 *
                   std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -2);
      return hess;
    }
  };

  template <int dim>
  class AnalyticalSolution : public Function<dim>
  {
    TamarindShape<dim> tamarind;

  public:
    double value(const Point<dim>  &point,
                 const unsigned int component = 0) const override;
    Tensor<1, dim> gradient(const Point<dim>  &point,
                            const unsigned int component = 0) const override;
  };

  template <int dim>
  double AnalyticalSolution<dim>::value(const Point<dim>  &point,
                                        const unsigned int component) const
  {
    AssertIndexRange(component, this->n_components);
    (void)component;
    return point[0] * point[1];
  }

  template <int dim>
  Tensor<1, dim>
  AnalyticalSolution<dim>::gradient(const Point<dim>  &point,
                                    const unsigned int component) const
  {
    AssertIndexRange(component, this->n_components);
    (void)component;
    const Tensor<1, dim> grad   = tamarind.gradient(point, component);
    const Tensor<1, dim> normal = (1.0 / grad.norm()) * grad;
    Tensor<1, dim>       projector_first_column = (-normal[0]) * normal;
    projector_first_column[0] += 1.0;
    Tensor<1, dim> projector_second_column = (-normal[1]) * normal;
    projector_second_column[1] += 1.0;
    Tensor<1, dim> surface_gradient =
      point[1] * projector_first_column + point[0] * projector_second_column;
    return surface_gradient;
  }

  template <int dim>
  class RightHandSide : public Function<dim>
  {
    TamarindShape<dim> tamarind;

  public:
    virtual double value(const Point<dim>  &p,
                         const unsigned int component = 0) const override;
  };

  template <int dim>
  double RightHandSide<dim>::value(const Point<dim>  &point,
                                   const unsigned int component) const
  {
    AssertIndexRange(component, this->n_components);
    (void)component;
    const Tensor<1, dim>          grad   = tamarind.gradient(point, component);
    const Tensor<1, dim>          normal = (1.0 / grad.norm()) * grad;
    const SymmetricTensor<2, dim> hess   = tamarind.hessian(point, component);

    double mean_curv = 0.0;
    for (int j = 0; j < 3; j++)
      for (int k = 0; k < 3; k++)
        mean_curv += (1.0) * (1.0 / grad.norm()) *
                     ((j == k ? 1 : 0) - normal[j] * normal[k]) * hess[j][k];
    return point[0] * point[1] + 2.0 * normal[0] * normal[1] +
           mean_curv * (point[1] * normal[0] + point[0] * normal[1]);
  }

  template <int dim>
  struct TraceFEM_ScratchData
  {
    TraceFEM_ScratchData(
      const Mapping<dim>                      &mapping,
      const hp::FECollection<dim>             &fe_collection,
      const NonMatching::MeshClassifier<dim> &mesh_classifier,
      const DoFHandler<dim>                   &level_set_dof_handler,
      const VectorType                        &level_set,
      const NonMatching::RegionUpdateFlags     nonmatching_update_flags,
      const Quadrature<dim>                   &quadrature,
      const Quadrature<dim - 1>               &quadrature_face,
      const Quadrature<1>                     &quadrature_edge,
      const UpdateFlags face_update_flags = update_values | update_gradients |
                                            update_quadrature_points |
                                            update_JxW_values |
                                            update_normal_vectors,
      const UpdateFlags cell_update_flags = update_values | update_gradients |
                                            update_quadrature_points |
                                            update_JxW_values)
      : fe_values(mapping,
                  fe_collection[ActiveFEIndex::lagrange],
                  quadrature,
                  cell_update_flags)
      , fe_interface_values(mapping,
                            fe_collection[ActiveFEIndex::lagrange],
                            quadrature_face,
                            face_update_flags)
      , region_update_flags(nonmatching_update_flags)
      , quadrature_1D(quadrature_edge)
      , fe_collection(fe_collection)
      , mesh_classifier(mesh_classifier)
      , level_set_dof_handler(level_set_dof_handler)
      , level_set(level_set)
      , level_set_fe_values(mapping,
                            level_set_dof_handler.get_fe(),
                            quadrature,
                            cell_update_flags)
      , non_matching_fe_values(fe_collection,
                               quadrature_edge,
                               nonmatching_update_flags,
                               mesh_classifier,
                               level_set_dof_handler,
                               level_set)
    {}

    TraceFEM_ScratchData(const TraceFEM_ScratchData<dim> &scratch_data)
      : fe_values(scratch_data.fe_values.get_mapping(),
                  scratch_data.fe_values.get_fe(),
                  scratch_data.fe_values.get_quadrature(),
                  scratch_data.fe_values.get_update_flags())
      , fe_interface_values(scratch_data.fe_interface_values.get_mapping(),
                            scratch_data.fe_interface_values.get_fe(),
                            scratch_data.fe_interface_values.get_quadrature(),
                            scratch_data.fe_interface_values.get_update_flags())
      , region_update_flags(scratch_data.region_update_flags)
      , quadrature_1D(scratch_data.quadrature_1D)
      , fe_collection(scratch_data.fe_collection)
      , mesh_classifier(scratch_data.mesh_classifier)
      , level_set_dof_handler(scratch_data.level_set_dof_handler)
      , level_set(scratch_data.level_set)
      , level_set_fe_values(scratch_data.level_set_fe_values.get_mapping(),
                            scratch_data.level_set_fe_values.get_fe(),
                            scratch_data.level_set_fe_values.get_quadrature(),
                            scratch_data.level_set_fe_values.get_update_flags())
      , non_matching_fe_values(fe_collection,
                               quadrature_1D,
                               region_update_flags,
                               mesh_classifier,
                               level_set_dof_handler,
                               level_set)
    {}

    FEValues<dim>          fe_values;
    FEInterfaceValues<dim> fe_interface_values;

    const NonMatching::RegionUpdateFlags    region_update_flags;
    const Quadrature<1>                    &quadrature_1D;
    const hp::FECollection<dim>            &fe_collection;
    const NonMatching::MeshClassifier<dim> &mesh_classifier;
    const DoFHandler<dim>                  &level_set_dof_handler;
    const VectorType                       &level_set;

    FEValues<dim>              level_set_fe_values;
    NonMatching::FEValues<dim> non_matching_fe_values;
  };



  template <int dim>
  struct TraceFEM_CopyData
  {
    FullMatrix<double>                   cell_matrix;
    Vector<double>                       cell_rhs;
    std::vector<types::global_dof_index> local_dof_indices;
    unsigned int                         cell_index;
    double                               cell_L2_error_sqr;
    double                               cell_H1_error_sqr;
    double                               cell_stab_error_sqr;

    template <class Iterator>
    void reinit(const Iterator &cell, const unsigned int dofs_per_cell)
    {
      cell_matrix.reinit(dofs_per_cell, dofs_per_cell);
      cell_rhs.reinit(dofs_per_cell);
      local_dof_indices.resize(dofs_per_cell);
      cell->get_dof_indices(local_dof_indices);
    }

    template <class Iterator>
    void reinit(const Iterator &cell)
    {
      cell_index          = cell->active_cell_index();
      cell_L2_error_sqr   = 0;
      cell_H1_error_sqr   = 0;
      cell_stab_error_sqr = 0;
    }
  };

  template <int dim>
  class NormalGradientVolumeStabilization
  {
  public:
    NormalGradientVolumeStabilization(VectorType &solution,
                                      VectorType &level_set)
      : solution(solution)
      , level_set(level_set)
      , stabilization_parameter(1.0)
      , stabilization_exponent(-1.0)
    {}

    VectorType &solution;
    VectorType &level_set;
    bool needs_cell_worker()
    {
      return true;
    }
    template <class Iterator>
    void cell_worker_assemble(const Iterator            &cell,
                              TraceFEM_ScratchData<dim> &scratch_data,
                              TraceFEM_CopyData<dim>    &copy_data)
    {
      const FEValues<dim> &fe_values = scratch_data.fe_values;
      const FEValues<dim> &level_set_fe_values =
        scratch_data.level_set_fe_values;
      const std::vector<double> &cellJxW = fe_values.get_JxW_values();

      std::vector<Tensor<1, dim>> grad_level_set(
        level_set_fe_values.get_quadrature().size());
      level_set_fe_values.get_function_gradients(level_set, grad_level_set);

      double factor =
        stabilization_parameter *
        std::pow(cell->minimum_vertex_distance(), stabilization_exponent);
      for (const unsigned int q : fe_values.quadrature_point_indices())
        {
          const Tensor<1, dim> &normal =
            (1.0 / grad_level_set[q].norm()) * grad_level_set[q];
          for (const unsigned int i : fe_values.dof_indices())
            for (const unsigned int j : fe_values.dof_indices())
              copy_data.cell_matrix(i, j) +=
                factor * (normal * fe_values.shape_grad(i, q)) *
                (normal * fe_values.shape_grad(j, q)) * cellJxW[q];
        }
    }

    template <class Iterator>
    void cell_worker_evaluate(const Iterator            &cell,
                              TraceFEM_ScratchData<dim> &scratch_data,
                              TraceFEM_CopyData<dim>    &copy_data)
    {
      double                     cell_stab_error_sqr = 0.0;
      const FEValues<dim>       &fe_values           = scratch_data.fe_values;
      const std::vector<double> &cellJxW = fe_values.get_JxW_values();
      const unsigned int n_q_points = fe_values.get_quadrature_points().size();
      const FEValues<dim> &level_set_fe_values =
        scratch_data.level_set_fe_values;

      std::vector<Tensor<1, dim>> level_set_grad(n_q_points);
      level_set_fe_values.get_function_gradients(level_set, level_set_grad);

      std::vector<Tensor<1, dim>> sol_grad(n_q_points);
      fe_values.get_function_gradients(solution, sol_grad);

      double factor =
        stabilization_parameter *
        std::pow(cell->minimum_vertex_distance(), stabilization_exponent);

      for (const unsigned int q : fe_values.quadrature_point_indices())
        {
          const Tensor<1, dim> normal =
            (1.0 / level_set_grad[q].norm()) * level_set_grad[q];

          const double error_at_point = normal * sol_grad[q];
          cell_stab_error_sqr +=
            factor * std::pow(error_at_point, 2.0) * cellJxW[q];
        }
      copy_data.cell_stab_error_sqr = cell_stab_error_sqr;
    }

  private:
    double stabilization_parameter;
    double stabilization_exponent;
  };

  template <int dim>
  class LaplaceBeltramiSolver
  {
  public:
    LaplaceBeltramiSolver();
    void run();

  private:
    void make_grid();

    void localize_surface();

    void postprocess_level_set();

    void setup_discrete_level_set();

    void distribute_dofs();

    void initialize_matrices();

    void assemble();

    void solve();

    void mark_intersected();

    void refine();

    void evaluate_errors();

    void output_level_set(unsigned int);

    void output_solution();

    void display_results();

    MPI_Comm mpi_communicator;

    TamarindShape<dim> Tamarind;

    const AnalyticalSolution<dim> analytical_solution;
    const RightHandSide<dim>      right_hand_side;

    parallel::distributed::Triangulation<dim, dim> triangulation;

    ConditionalOStream pcout;
    TimerOutput        computing_timer;

    const unsigned int fe_degree;
    const unsigned int level_set_fe_degree;

    const QGauss<dim>     cell_quadrature;
    const QGauss<dim - 1> face_quadrature;
    const QGauss<1>       quadrature_1D;

    hp::FECollection<dim> fe_collection;
    const FE_Q<dim>       level_set_fe;

    DoFHandler<dim> dof_handler;
    DoFHandler<dim> level_set_dof_handler;

    AffineConstraints<double> constraints;
    AffineConstraints<double> level_set_constraints;

    VectorType    completely_distributed_solution;
    VectorType    locally_relevant_solution;
    VectorType    locally_relevant_exact;
    VectorType    level_set;
    Vector<float> activeFE_indicator;

    NonMatching::MeshClassifier<dim> mesh_classifier;
    const MappingQ1<dim>             mapping;

    NormalGradientVolumeStabilization<dim> stabilization_scheme;

    VectorType global_rhs;
    MatrixType global_matrix;
```

```
    SparsityPattern sparsity_pattern;

    IndexSet locally_owned_dofs;
    IndexSet locally_relevant_dofs;

    NonMatching::RegionUpdateFlags surface_update_flags;
    UpdateFlags                    face_update_flags;
    UpdateFlags                    cell_update_flags;

    ConvergenceTable convergence_table;
    unsigned int     number_of_iterations;
    double           average;
    double           area;
    double           error_L2_sqr;
    double           error_H1_sqr;
    double           error_stab_sqr;

    int number_of_proc;
  };

  template <int dim>
  LaplaceBeltramiSolver<dim>::LaplaceBeltramiSolver()
    : mpi_communicator(MPI_COMM_WORLD)
    , triangulation(
        mpi_communicator,
        typename Triangulation<dim>::MeshSmoothing(
          Triangulation<dim>::
            none //smoothing_on_refinement |  Triangulation<dim>::smoothing_on_coarsening
          ),
        parallel::distributed::Triangulation<dim, dim>::
          mesh_reconstruction_after_repartitioning)
    , pcout(std::cout,
            (Utilities::MPI::this_mpi_process(mpi_communicator) == 0))
    , computing_timer(mpi_communicator,
                      pcout,
                      TimerOutput::never,
                      TimerOutput::wall_times)
    , fe_degree(1)
    , level_set_fe_degree(1)
    , cell_quadrature(fe_degree + 1)
    , face_quadrature(fe_degree + 1)
    , quadrature_1D(fe_degree + 1)
    , level_set_fe(level_set_fe_degree)
    , dof_handler(triangulation)
    , level_set_dof_handler(triangulation)
    , mesh_classifier(level_set_dof_handler, level_set)
    , mapping()
    , stabilization_scheme(locally_relevant_solution, level_set)
  {
    fe_collection.push_back(FE_Q<dim>(fe_degree));
    fe_collection.push_back(FE_Nothing<dim>());

    surface_update_flags.surface =
      update_values | update_gradients | update_JxW_values |
      update_quadrature_points | update_normal_vectors;

    face_update_flags = update_default;
    cell_update_flags = update_default;
    MPI_Comm_size(mpi_communicator, &number_of_proc);
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::make_grid()
  {
    pcout << std::endl
          << "Creating background mesh with MPI_Size=" << number_of_proc
          << std::endl;
    double cube_side = 2.0;
    GridGenerator::hyper_cube(triangulation, -cube_side, cube_side);
    triangulation.refine_global(3);
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::postprocess_level_set()
  {
    const hp::FECollection<dim, dim> &level_set_fe(
      level_set_dof_handler.get_fe_collection());
    std::vector<types::global_dof_index> level_set_dofs_on_cell(
      level_set_fe.max_dofs_per_cell());
    for (const auto &level_set_cell :
         level_set_dof_handler.active_cell_iterators())
      {
        if (!level_set_cell->is_locally_owned())
          continue;
        const double cell_side_length =
          level_set_cell->minimum_vertex_distance();
        const auto n_dofs =
          level_set_fe[level_set_cell->active_fe_index()].n_dofs_per_cell();
        level_set_dofs_on_cell.resize(n_dofs);
        level_set_cell->get_dof_indices(level_set_dofs_on_cell);
        for (unsigned int i = 0; i < n_dofs; ++i)
          {
            if (std::abs(level_set[level_set_dofs_on_cell[i]]) <
                cell_side_length * 1e-12)
              level_set[level_set_dofs_on_cell[i]] = cell_side_length * 1e-12;
          }
      }
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::setup_discrete_level_set()
  {
    pcout
      << "Setting up discrete level set function and reclassifying cells on MPI_rank=0... "
      << std::flush;
    TimerOutput::Scope t(computing_timer, "setup_level_set");
    Timer              timer;
    activeFE_indicator.reinit(triangulation.n_active_cells());

    level_set_dof_handler.distribute_dofs(level_set_fe);

    level_set_constraints.clear();
    IndexSet level_set_locally_relevant_dofs;
    DoFTools::extract_locally_relevant_dofs(level_set_dof_handler,
                                            level_set_locally_relevant_dofs);
    level_set_constraints.reinit(level_set_locally_relevant_dofs);
    DoFTools::make_hanging_node_constraints(level_set_dof_handler,
                                            level_set_constraints);
    level_set_constraints.close();
    VectorType tmp_sol;
    tmp_sol.reinit(level_set_dof_handler.locally_owned_dofs(),
                   mpi_communicator);
    VectorTools::interpolate(level_set_dof_handler, Tamarind, tmp_sol);

    level_set.reinit(level_set_locally_relevant_dofs,
                     level_set_dof_handler.locally_owned_dofs(),
                     mpi_communicator);
    level_set_constraints.distribute(tmp_sol);
    level_set = tmp_sol;

    postprocess_level_set();

    mesh_classifier.reclassify();

    for (const auto &cell : dof_handler.active_cell_iterators())
      {
        if (!cell->is_locally_owned())
          continue;
        if (mesh_classifier.location_to_level_set(cell) ==
            NonMatching::LocationToLevelSet::intersected)
          cell->set_active_fe_index(ActiveFEIndex::lagrange);
        else
          cell->set_active_fe_index(ActiveFEIndex::nothing);
      }
    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::mark_intersected()
  {
    pcout << "Determining cells with active FE index on MPI_rank=0..."
          << std::flush;
    Timer timer;
    for (const auto &cell : dof_handler.active_cell_iterators())
      {
        if (!cell->is_locally_owned())
          continue;
        if (mesh_classifier.location_to_level_set(cell) ==
            NonMatching::LocationToLevelSet::intersected)
          activeFE_indicator[cell->active_cell_index()] = 1.0;
      }
    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }


  template <int dim>
  void LaplaceBeltramiSolver<dim>::refine()
  {
    Timer              timer;
    TimerOutput::Scope t(computing_timer, "refine");
    pcout << "Refining near surface on MPI_rank=0"
          << "... " << std::flush;
    parallel::distributed::GridRefinement::refine_and_coarsen_fixed_fraction(
      triangulation, activeFE_indicator, 1.0, 0.0);

    triangulation.execute_coarsening_and_refinement();
    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::distribute_dofs()
  {
    pcout << "Distributing degrees of freedom on MPI_rank=0... " << std::flush;
    Timer timer;
    dof_handler.distribute_dofs(fe_collection);
    locally_owned_dofs = dof_handler.locally_owned_dofs();
    locally_relevant_dofs =
      DoFTools::extract_locally_relevant_dofs(dof_handler);
    completely_distributed_solution.reinit(dof_handler.locally_owned_dofs(),
                                           mpi_communicator);
    locally_relevant_solution.reinit(locally_owned_dofs,
                                     locally_relevant_dofs,
                                     mpi_communicator);
    global_rhs.reinit(locally_owned_dofs, mpi_communicator);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::initialize_matrices()
  {
    pcout << "Initializing the matricex on MPI_rank=0... " << std::flush;
    Timer                  timer;
    DynamicSparsityPattern dsp(dof_handler.n_dofs(), dof_handler.n_dofs());
    constraints.clear();
    constraints.reinit(locally_relevant_dofs);

    DoFTools::make_hanging_node_constraints(dof_handler, constraints);
    constraints.close();
    DoFTools::make_sparsity_pattern(dof_handler, dsp, constraints);

    SparsityTools::distribute_sparsity_pattern(dsp,
                                               locally_owned_dofs,
                                               mpi_communicator,
                                               locally_relevant_dofs);
    global_matrix.reinit(locally_owned_dofs,
                         locally_owned_dofs,
                         dsp,
                         mpi_communicator);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::assemble()
  {
    pcout << "Assembling on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "assembly");
    Timer              timer;
    using Iterator        = typename DoFHandler<dim>::active_cell_iterator;
    const auto cell_worker = [&](const Iterator            &cell,
                                 TraceFEM_ScratchData<dim> &scratch_data,
                                 TraceFEM_CopyData<dim>    &copy_data) {
```

```
      if (mesh_classifier.location_to_level_set(cell) ==
            NonMatching::LocationToLevelSet::intersected &&
          cell->is_locally_owned())
        {
          scratch_data.non_matching_fe_values.reinit(cell);
          copy_data.reinit(cell,
                           scratch_data.fe_values.get_fe().n_dofs_per_cell());
          copy_data.cell_matrix = 0;
          copy_data.cell_rhs    = 0;
          const std::optional<NonMatching::FEImmersedSurfaceValues<dim>>
            &surface_fe_values =
              scratch_data.non_matching_fe_values.get_surface_fe_values();
          const std::vector<double> &surfJxW =
            surface_fe_values->get_JxW_values();

          for (unsigned int q : surface_fe_values->quadrature_point_indices())
            {
              const Tensor<1, dim> &normal =
                surface_fe_values->normal_vector(q);

              for (const unsigned int i : surface_fe_values->dof_indices())
                {
                  copy_data.cell_rhs(i) +=
                    right_hand_side.value(
                      surface_fe_values->quadrature_point(q)) *
                    surface_fe_values->shape_value(i, q) * surfJxW[q];

                  for (const unsigned int j : surface_fe_values->dof_indices())
                    {
                      copy_data.cell_matrix(i, j) +=
                        1.0 *
                        (surface_fe_values->shape_value(i, q) *
                         surface_fe_values->shape_value(j, q)) *
                        surfJxW[q];
                      copy_data.cell_matrix(i, j) +=
                        (1.0 *
                         (surface_fe_values->shape_grad(i, q) -
                          (normal * surface_fe_values->shape_grad(i, q)) *
                            normal) *
                         (surface_fe_values->shape_grad(j, q) -
                          (normal * surface_fe_values->shape_grad(j, q)) *
                            normal)) *
                        surfJxW[q];
                    }
                }
            }

          if (stabilization_scheme.needs_cell_worker())
            {
              typename DoFHandler<dim>::active_cell_iterator level_set_cell(
                &(triangulation),
                cell->level(),
                cell->index(),
                &level_set_dof_handler);
              scratch_data.fe_values.reinit(cell);
              scratch_data.level_set_fe_values.reinit(level_set_cell);
              stabilization_scheme.cell_worker_assemble(cell,
                                                        scratch_data,
                                                        copy_data);
            }
        }
    };
    const auto copier = [&](const TraceFEM_CopyData<dim> &c) {
      constraints.distribute_local_to_global(c.cell_matrix,
                                             c.cell_rhs,
                                             c.local_dof_indices,
                                             global_matrix,
                                             global_rhs);
    };

    TraceFEM_ScratchData<dim> scratch_data(mapping,
                                           fe_collection,
                                           mesh_classifier,
                                           level_set_dof_handler,
                                           level_set,
                                           surface_update_flags,
                                           cell_quadrature,
                                           face_quadrature,
                                           quadrature_1D,
                                           face_update_flags);
    TraceFEM_CopyData<dim>    copy_data;
    MeshWorker::mesh_loop(dof_handler.begin_active(),
                          dof_handler.end(),
                          cell_worker,
                          copier,
                          scratch_data,
                          copy_data,
                          MeshWorker::assemble_own_cells);

    global_matrix.compress(VectorOperation::add);
    global_rhs.compress(VectorOperation::add);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::solve()
  {
    TimerOutput::Scope t(computing_timer, "solve");
    Timer              timer;
    bool               apply_direct_solver = false;
    if (apply_direct_solver)
      {
        pcout << "Solving directly on MPI_rank=0... " << std::flush;
        SolverControl solver_control(2000, 1e-8);
        TrilinosWrappers::SolverDirect::AdditionalData data;
        TrilinosWrappers::SolverDirect trilinos(solver_control, data);
        trilinos.solve(global_matrix,
                       completely_distributed_solution,
                       global_rhs);
        number_of_iterations = -1;
      }
    else
      {
        Timer timer;
        pcout << "Solving with AMG on MPI_rank=0... " << std::flush;
        const unsigned int             max_iterations = dof_handler.n_dofs();
        SolverControl                  solver_control(max_iterations);
        std::vector<std::vector<bool>> constant_modes;
        DoFTools::extract_constant_modes(dof_handler,
                                         ComponentMask(),
                                         constant_modes);
        TrilinosWrappers::PreconditionAMG preconditioner_stiffness;
        TrilinosWrappers::PreconditionAMG::AdditionalData Amg_data;
        Amg_data.constant_modes        = constant_modes;
        Amg_data.elliptic              = true;
        Amg_data.higher_order_elements = false;
        Amg_data.smoother_sweeps       = 2;
        Amg_data.aggregation_threshold = 0.02;
        Amg_data.output_details        = true;
        preconditioner_stiffness.initialize(global_matrix);

        SolverCG<VectorType> cg(solver_control);
        cg.solve(global_matrix,
                 completely_distributed_solution,
                 global_rhs,
                 preconditioner_stiffness);
        pcout << "required " << solver_control.last_step() << " iterations and "
              << std::flush;
        number_of_iterations = solver_control.last_step();
      }
    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
    constraints.distribute(completely_distributed_solution);
    locally_relevant_solution = completely_distributed_solution;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::evaluate_errors()
  {
    pcout << "Evaluating errors on the surface on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "eval_errors");
    Timer              timer;
    error_L2_sqr           = 0.0;
    error_H1_sqr           = 0.0;
    error_stab_sqr         = 0.0;
    const auto cell_worker = [&](const auto &cell,
                                 auto       &scratch_data,
                                 auto       &copy_data) {
      if (mesh_classifier.location_to_level_set(cell) ==
            NonMatching::LocationToLevelSet::intersected &&
          cell->is_locally_owned())
        {
          double cell_L2_error_sqr = 0.0;
          double cell_H1_error_sqr = 0.0;

          copy_data.reinit(cell);
          scratch_data.non_matching_fe_values.reinit(cell);

          const std::optional<NonMatching::FEImmersedSurfaceValues<dim>>
            &surface_fe_values =
              scratch_data.non_matching_fe_values.get_surface_fe_values();
          const std::vector<double> &surfJxW =
            surface_fe_values->get_JxW_values();
          int n_q_points = surface_fe_values->n_quadrature_points;

          std::vector<double> sol(n_q_points);
          surface_fe_values->get_function_values(locally_relevant_solution,
                                                 sol);

          std::vector<Tensor<1, dim>> sol_grad(n_q_points);
          surface_fe_values->get_function_gradients(locally_relevant_solution,
                                                    sol_grad);

          for (const unsigned int q :
               surface_fe_values->quadrature_point_indices())
            {
              const Point<dim> &point = surface_fe_values->quadrature_point(q);
              const Tensor<1, dim> &normal =
                surface_fe_values->normal_vector(q);
              const double error_at_point =
                sol.at(q) - analytical_solution.value(point);
              const Tensor<1, dim> vector_error_at_point =
                (sol_grad.at(q) - (normal * sol_grad.at(q)) * normal -
                 analytical_solution.gradient(point));

              cell_L2_error_sqr += std::pow(error_at_point, 2) * surfJxW[q];
              cell_H1_error_sqr +=
                vector_error_at_point * vector_error_at_point * surfJxW[q];
            }
          copy_data.cell_L2_error_sqr = cell_L2_error_sqr;
          copy_data.cell_H1_error_sqr = cell_H1_error_sqr;

          if (stabilization_scheme.needs_cell_worker())
            {
              typename DoFHandler<dim>::active_cell_iterator level_set_cell(
                &(triangulation),
                cell->level(),
                cell->index(),
                &level_set_dof_handler);
              scratch_data.fe_values.reinit(cell);
              scratch_data.level_set_fe_values.reinit(level_set_cell);
              stabilization_scheme.cell_worker_evaluate(cell,
                                                        scratch_data,
                                                        copy_data);
            }
        }
    };

    const auto copier = [&](const auto &copy_data) {
      if (copy_data.cell_index < activeFE_indicator.size())
        {
          error_L2_sqr += copy_data.cell_L2_error_sqr;
          error_H1_sqr += copy_data.cell_H1_error_sqr;
          error_stab_sqr += copy_data.cell_stab_error_sqr;
        }
    };

    TraceFEM_ScratchData<dim> scratch_data(mapping,
                                           fe_collection,
                                           mesh_classifier,
                                           level_set_dof_handler,
                                           level_set,
                                           surface_update_flags,
                                           cell_quadrature,
                                           face_quadrature,
                                           quadrature_1D,
                                           face_update_flags);
    TraceFEM_CopyData<dim>    copy_data;
```

```
    MeshWorker::mesh_loop(dof_handler.begin_active(),
                          dof_handler.end(),
                          cell_worker,
                          copier,
                          scratch_data,
                          copy_data,
                          MeshWorker::assemble_own_cells);
    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::display_results()
  {
    const double error_L2 =
      std::sqrt(Utilities::MPI::sum(error_L2_sqr, mpi_communicator));
    const double error_semiH1 =
      std::sqrt(Utilities::MPI::sum(error_H1_sqr, mpi_communicator));
    const double error_stab =
      std::sqrt(Utilities::MPI::sum(error_stab_sqr, mpi_communicator));

    const int dof_handler_size           = dof_handler.n_dofs();
    const int level_set_dof_handler_size = level_set_dof_handler.n_dofs();

    const double iterations =
      Utilities::MPI::max(number_of_iterations, mpi_communicator);

    convergence_table.add_value("LevelSet dofs", level_set_dof_handler_size);
    convergence_table.evaluate_convergence_rates(
      "LevelSet dofs", ConvergenceTable::reduction_rate_log2);

    convergence_table.add_value("Active dofs", dof_handler_size);
    convergence_table.evaluate_convergence_rates(
      "Active dofs", ConvergenceTable::reduction_rate_log2);


    convergence_table.add_value("Iterations", iterations);

    convergence_table.add_value("L2 Error", error_L2);
    convergence_table.evaluate_convergence_rates(
      "L2 Error", ConvergenceTable::reduction_rate_log2);
    convergence_table.set_scientific("L2 Error", true);

    convergence_table.add_value("H1 error", error_semiH1);
    convergence_table.evaluate_convergence_rates(
      "H1 error", ConvergenceTable::reduction_rate_log2);
    convergence_table.set_scientific("H1 error", true);

    convergence_table.add_value("Stab norm", error_stab);
    convergence_table.evaluate_convergence_rates(
      "Stab norm", ConvergenceTable::reduction_rate_log2);
    convergence_table.set_scientific("Stab norm", true);

    pcout << std::endl;
    if (Utilities::MPI::this_mpi_process(mpi_communicator) == 0)
      convergence_table.write_text(pcout.get_stream());
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::output_level_set(unsigned int n)
  {
    pcout << "Writing vtu file for surface on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "output_level_set");
    Timer              timer;
    DataOut<dim>       data_out;
    data_out.add_data_vector(level_set_dof_handler, level_set, "level_set");
    data_out.add_data_vector(activeFE_indicator, "ref_indicator");
    data_out.build_patches();

    data_out.write_vtu_in_parallel("surface_" + std::to_string(n) + ".vtu",
                                   mpi_communicator);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::output_solution()
  {
    pcout << "Writing vtu file on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "output_solution");
    Timer              timer;
    Vector<double>     exact(dof_handler.locally_owned_dofs().size());

    VectorTools::interpolate(dof_handler, analytical_solution, exact);
    DataOut<dim> data_out;
    data_out.add_data_vector(dof_handler,
                             locally_relevant_solution,
                             "solution");
    data_out.add_data_vector(dof_handler, exact, "exact");
    data_out.add_data_vector(level_set_dof_handler, level_set, "level_set");

    data_out.set_cell_selection(
      [this](const typename Triangulation<dim>::cell_iterator &cell) {
        return cell->is_active() && cell->is_locally_owned() &&
               mesh_classifier.location_to_level_set(cell) ==
                 NonMatching::LocationToLevelSet::intersected;
      });
    data_out.build_patches();

    data_out.write_vtu_in_parallel("solution.vtu", mpi_communicator);

    timer.stop();
    pcout << "took (" << timer.wall_time() << "s)" << std::endl;
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::localize_surface()
  {
    unsigned int preliminary_levels = 3;
    for (unsigned int localization_cycle = 0;
         localization_cycle < preliminary_levels;
         localization_cycle++)
      {
        pcout << std::endl
              << "Preliminary refinement #" << localization_cycle << std::endl;
        setup_discrete_level_set();
        mark_intersected();
        output_level_set(localization_cycle);
        refine();
      }
    computing_timer.reset();
  }

  template <int dim>
  void LaplaceBeltramiSolver<dim>::run()
  {
    make_grid();
    localize_surface();
    const unsigned int convergence_levels = 3;
    for (unsigned int cycle = 0; cycle < convergence_levels; cycle++)
      {
        pcout << std::endl << "Convergence refinement #" << cycle << std::endl;
        setup_discrete_level_set();
        distribute_dofs();
        initialize_matrices();
        assemble();
        solve();
        evaluate_errors();
        display_results();
        if (cycle < convergence_levels - 1)
          {
            mark_intersected();
            refine();
          }
        else
          output_solution();
        computing_timer.print_summary();
        computing_timer.reset();
      }
  }
} // namespace Step90

int main(int argc, char *argv[])
{
  try
    {
      using namespace dealii;
      using namespace Step90;
      Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);
      LaplaceBeltramiSolver<3>         LB_solver;
      LB_solver.run();
    }
  catch (std::exception &exc)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Exception on processing: " << std::endl
                << exc.what() << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;

      return 1;
    }
  catch (...)
    {
      std::cerr << std::endl
                << std::endl
                << "----------------------------------------------------"
                << std::endl;
      std::cerr << "Unknown exception!" << std::endl
                << "Aborting!" << std::endl
                << "----------------------------------------------------"
                << std::endl;
      return 1;
    }

  return 0;
}
```

Generated by doxygen 1.9.1