The step-90 tutorial program

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

Table of contents

This program was contributed by Vladimir Yushutin and Timo Heister, Clemson University, 2023.

This material is based upon work partly supported by the National Science Foundation Award DMS-

Introduction

In this tutorial, we implement the trace finite element method (TraceFEM) in deal.II. TraceFEM solves PDEs posed on a possibly evolving $(dim-1)$-dimensional surface Γ employing a fixed uniform background mesh of a dim-dimensional domain in which the surface is embedded. Such surface PDEs arise in problems involving material films with complex properties and in other situations in which a non-trivial condition is imposed on either a stationary or a moving interface. Here we consider a steady, complex, non-trivial surface and the prototypical Laplace-Beltrami equation which is a counterpart of the Poisson problem on flat domains.

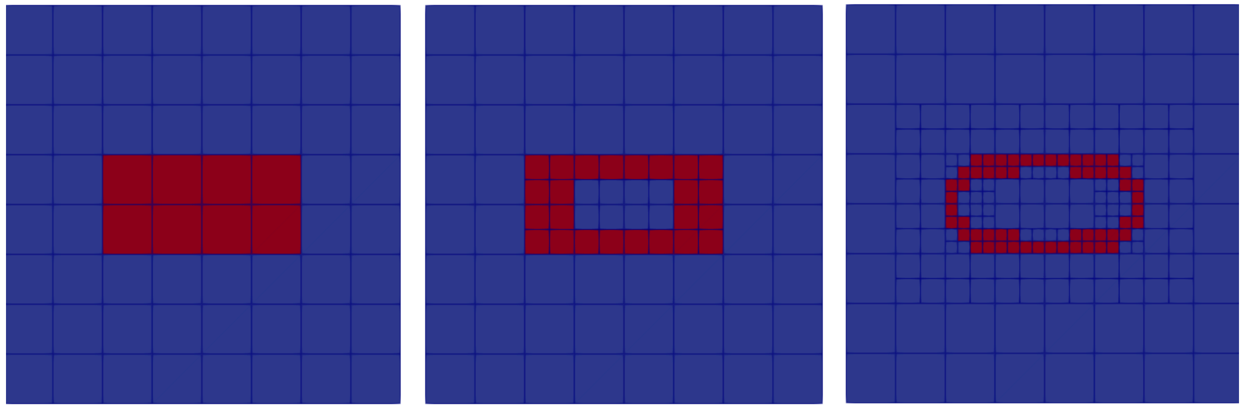
Being an unfitted method, TraceFEM allows to circumvent the need of remeshing of an evolving surface if it is implicitly given by the zero contour of a level-set function. At the same time, it easily provides with an extension of the discrete solution to a neighborhood of the surface which turns out to be very handy in case of non-stationary interfaces and films. Certainly, this flexibility comes with a price: one needs to design the nodes and weights for a quadrature customized for each implicit intersection of the zero level-set and the background mesh. Moreover, these intersections may be of arbitrary shape and size manifesting in the so-called ``small cut" problem and requiring addition of a stabilization form that restores well-conditioning of the problem.

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 on the same bulk triangulation. Second, we make sure that the 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 natural alternative to TraceFEM in solving surface PDEs is the parametric surface finite element method. The latter method relies on an explicit parametrization of the surface which may be not feasible especially for evolving interfaces with an unknown in advance shape - in this sense, TraceFEM is a technique inspired by the level-set description of interfaces. However, the parametric surface finite element method, when applicable, enjoys many well-known properties of fitted methods on flat domains provided the geometric errors - which a present for both methods - are taken under control.

A non-trivial surface

A fitted FEM on a flat two-dimensional domain, if discretized by piecewise linears with N degrees of freedom, typically results in $O(h)=O(N^{-1/2})$ convergence rate of the energy error; requires $O(N)$ storage for the degrees of freedom; and, more importantly, takes $O(N)$ of construction time to create them, i.e. to mesh the domain. TraceFEM, although solving a two-dimensional problem, relies on the inherently three-dimensional mesh on which the level-set function must be defined and, if implemented naively, suffers from the increased storage and the increased construction time in terms of the active degrees of freedom N_a that actually enters the scheme with, hopefully, $O(N_a^{-1/2})$ error. To combat these possible bottlenecks, we create iteratively a mesh which is localized near the zero contour line of the level set function, i.e near the surface, to restore the aforementioned two-dimensional performance typical for fitted FEM, see the first three typical iterations of this methodology below.



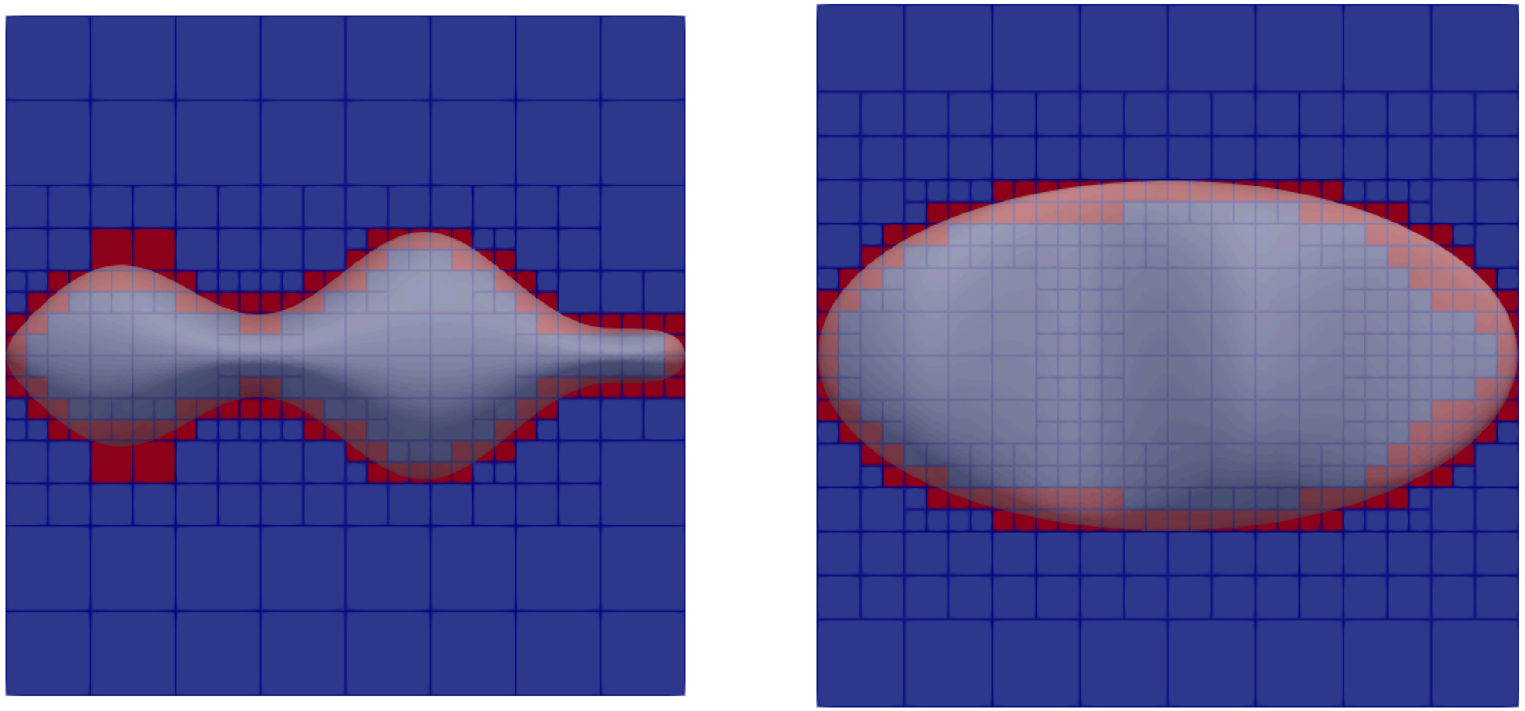
Iterative localization of the zero contour of a typical level set

The cells colored by red cary the active degrees of freedom (total number N_a) as the level set is not sign-definite at support points. Notice also that the mesh is graded: any cell has at most 4 neighbors adjacent to each of 6 faces.

Once a desired geometry approximation Γ_h is achieved using the iterative approach above, we can start forming the linear system using the constructed normals and quadratures. For the purposes of the tutorial we choose a non-trivial surface Γ given by

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

The OY and OX views of this tamarind-shaped, exact surface Γ are shown below along with the mesh after three iterations (the approximation Γ_h is not shown).



OY(left) and OZ(right) cross-sections of the background mesh along with the exact surface

Model problem

We would like to solve the simplest possible problem defined on a surface, namely the Laplace–Beltrami equation,

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

where we take $c = 1$ for concreteness. We added the term cu to the left-hand side so the problem becomes well-posed in the absence of any boundary; an alternative could be to take $c = 0$ but impose the zero mean condition.

Exact solution

We choose the test solution and the right-hand side forcing as the restriction to Γ of

$$u(x,y,z) = xy\,, \quad f(x,y,z) = xy + 2.0\,\mathbf{n}_x\mathbf{n}_y + \kappa(y\mathbf{n}_x + x\mathbf{n}_y),$$

where the latter is manufactured using the exact normal $\mathbf{n}$, the exact Hessian $\nabla^2\mathbf{n}$ and the mean curvature, $\kappa = \mathrm{div} n$ of the surface. Note that we do not need to impose any boundary conditions as the surface Γ is closed.

The Trace Finite Element Method

TraceFEM is an unfitted method: 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 a surface approximation Γ_h by intersecting implicitly the cells of the background mesh with the iso surface of an approximation of the level-set field. We note that we never actually create any two-dimensional meshes for the surface but only compute approximate quadrature points and surface normals. Next we distribute degrees of freedom over a thin subdomain Ω_h that completely covers Γ_h and that consists of the intersected cells $\mathcal{T}_\Gamma^h$,

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

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

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

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

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 $\mathcal{T}_\Gamma^h$ or not, we set the active_fe_index to either 0 or 1. To determine whether a cell is intersected or not, we use the class **NonMatching::MeshClassifier**.

A natural candidate for a weak formulation involves the following (bi)linear 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 (non-necessarily the the so-called normal extension) of f from Γ to Ω_h. Note that the right-hand side f of the Laplace-Beltrami problem is defined on the exact surface Γ only and we need to specify how to evaluate its action on the perturbed approximate geometry Γ_h which is immersed in Ω_h. For the purposes of this test, the forcing f is manufactured using $u = xy$ and the level-set function and, therefore, is a function of Cartesian coordinates x, y, z. The latter is identified with f^e on Γ_h and it is not the normal extension of the function f.

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 extreme case [160].

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 $\mathbf{n}_h = \nabla\psi_h/\|\nabla\psi_h\|$ which, together with approximate quadrature for the integration over Γ_h, leads to the so-called "geometrical error". Luckily, one can show [160] 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 to choose independently the discrete space for the solution and a higher-order discrete space for the level set function for a more accurate geometric approximation.

The commented program

```
#include <deal.II/base/convergence_table.h>
#include <deal.II/base/function.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/distributed/grid_refinement.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/dofs/dof_tools.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/fe/mapping_q1.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/sparse_direct.h>
#include <deal.II/lac/sparsity_pattern.h>
#include <deal.II/lac/sparsity_tools.h>
#include <deal.II/lac/trilinos_precondition.h>
#include <deal.II/lac/trilinos_solver.h>
#include <deal.II/lac/trilinos_sparse_matrix.h>
#include <deal.II/lac/trilinos_vector.h>
#include <deal.II/lac/vector.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/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
```

The parallization in this tutorial relies on the Trilinos library. 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.

```
using namespace dealii;
using VectorType = TrilinosWrappers::MPI::Vector;
using MatrixType = TrilinosWrappers::SparseMatrix;
namespace Step90
{
  enum class ActiveFEIndex : types::fe_index
  {
    lagrange = 0,
    nothing  = 1
  };
```

Exact surface

The following class defines the surface using the implicit level set representation. The exact surface normal uses the Cartesian gradient of the level set function. The exact Hessian is needed for the construction of the test case only.

```
  template <int dim>
  class TamarindShape : public Function<dim>
  {
  public:
    double value(const Point<dim> &point,
                 const unsigned int /*component*/ = 0) const override
    {
      Assert(dim == 3, ExcNotImplemented());
      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;
      Assert(dim == 3, ExcNotImplemented());

      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;
      Assert(dim == 3, ExcNotImplemented());
      SymmetricTensor<2, dim> hessian;

      hessian[0][0] =
        0.5 +
        8.0 * 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) * 0.5 *
             numbers::PI * numbers::PI * std::sin(numbers::PI * point[0]));
      hessian[0][1] = 0.0;
      hessian[0][2] =
        (-8.0) * point[2] *
        std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
        numbers::PI * std::cos(numbers::PI * point[0]);

      hessian[1][1] = 2.0;
      hessian[1][2] = 0.0;

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

      return hessian;
    }
  };
```

Exact solution

The following class defines the chosen exact solution and its surface gradient. The exact solution $u = xy$ and it may be evaluated away from Γ as any other function of Cartesian points. Also note that the gradient() method returns the surface gradient $\nabla_\Gamma u$ of the exact solution.

```
template <int dim>
class AnalyticalSolution : public Function<dim>
{
private:
  const 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;
}
```

Exact forcing

An evaluation of the surface Laplacian for a manufactured solution u that corresponds to the exact forcing $f = -\Delta_\Gamma u + u$.

```
    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;
      Assert(dim == 3, ExcNotImplemented());
      const Tensor<1, dim>           grad    = tamarind.gradient(point, component);
      const Tensor<1, dim>           normal  = (1.0 / grad.norm()) * grad;
      const SymmetricTensor<2, dim> hessian = 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]) * hessian[j][k];

      return point[0] * point[1] + 2.0 * normal[0] * normal[1] +
             mean_curv * (point[1] * normal[0] + point[0] * normal[1]);
    }
```

Scratch and Copy objects for TraceFEM

Since the assembly procedure will be performed via **MeshWorker**, we need a Scratch object that handles the Non-Matching **FEValues** effectively. The input arguments of its constructor are discussed in the solver class below.

```
    template <int dim>
    struct ScratchData
    {
      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<1>                  &quadrature_edge,
              const UpdateFlags cell_update_flags = update_values |
                                                    update_gradients |
                                                    update_quadrature_points |
                                                    update_JxW_values)
        : fe_values(
            mapping,
            fe_collection[static_cast<types::fe_index>(ActiveFEIndex::lagrange)],
            quadrature,
            cell_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)
      {}

      ScratchData(const 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())
        , 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 **FEValues** is for the standard quadrature on cells involving the FE space of the solution. In TraceFEM, we need this quadrature due to the stabilization term. In addition, a cell quadrature for the FE space of the level set is defined. At the end, the CopyData object custimized for TraceFEM is presented. In particular, the implementation of the normal-gradient volume stabilization relies on it.

```
      FEValues<dim>                          fe_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 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. More specifically, the method needs_cell_worker() indicates whether the bilinear form of the stabilization, unlike the main bilinear form of Laplace-Beltrami operator, needs the bulk cell quadratures. The cell worker which is useful in an accumulation by MeshWorkers is provided by the assemble_cell_worker() method. The remaining method evaluate_cell_worker() computes the stabilization error for the solution u_h, i.e $s_h(u_h, u_h)$. Also note that the method needs_cell_worker() indicates that the assembly and the evaluation of the form does require a bulk cell quadrature. This methodology may be utilized in the **MeshWorker**. The stabilization scaling is specified by $\mathrm{stabilization\_parameter} \cdot h^{\mathrm{stabilization\_exponent}}$. For elliptic problems with smooth solutions we can choose any $-1 \leq \mathrm{stabilization\_exponent} \leq 1$ and a sufficiently large $\mathrm{stabilization\_parameter}$ that depends of Γ.

```
    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;
      }
```

We define the stabilization form here assuming that ScratchData and CopyData arguments are initialized properly. The local contribution of the stabilization from this cell to the global matrix is given in assemble_cell_worker() and, later in evaluate_cell_worker(), the local bilinear form of the stabilization is evaluated on the solution. Note the gradients of the discrete level set are computed in the bulk cell quadrature points, which, upon normalization, give the discrete normal vector in a bulk cell.

```
      void assemble_cell_worker(
        const typename DoFHandler<dim>::active_cell_iterator &cell,
        ScratchData<dim>                                     &scratch_data,
        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);

        const 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];
          }
      }

      void evaluate_cell_worker(
        const typename DoFHandler<dim>::active_cell_iterator &cell,
        ScratchData<dim>                                     &scratch_data,
        CopyData<dim>                                        &copy_data)
      {
        double                       cell_stab_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);

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

    private:
      const double stabilization_parameter;
      const double stabilization_exponent;
    };
```

Laplace–Beltrami solver

The main class whose method **run()** performs the computation. One may adjust main parameters of TraceFEM in the constructor. The other methods are discussed are below.

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

    private:
      void make_grid();

      void localize_surface();

      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

```
      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 bulk quadrature is required for the for TraceFEM stabilization, while the integration over implicit surface is based on the last, one-dimensional rule.

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

We need two 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.

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

The corresponding **DoFHandler** objects 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                    cell_update_flags;
```

The following variables are used to display the results 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;
    };

    template <int dim>
    LaplaceBeltramiSolver<dim>::LaplaceBeltramiSolver()
      : mpi_communicator(MPI_COMM_WORLD)
      , triangulation(mpi_communicator)
      , 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)
      , 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;
      cell_update_flags = update_default;
    }
```

Geometric approximation

The initial refinement helps the level set to approximate the surface meaningfully. The background cube size is chosen to avoid situations in which level set function vanishes at mesh vertices. In the next method we construct the discrete level set and determine which cells are intersected. Note that all cells, intersected and non-intersected, have a correspondent value in the activeFE_indicator. Similarly, the exact level set function is approximated on the whole triangulation and postprocessed afterward resulting in a surface approximation with no gaps.

```
    template <int dim>
    void LaplaceBeltramiSolver<dim>::make_grid()
    {
      pcout << std::endl
            << "Creating background mesh with MPI_Size="
            << Utilities::MPI::this_mpi_process(mpi_communicator) << std::endl;
      const double cube_side = 2.008901281;
      GridGenerator::hyper_cube(triangulation, -cube_side, cube_side);
      triangulation.refine_global(3);
    }

    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();
```

Here is where the geometric information enters the code. Next, using the discrete level set, we mark the cell which are intersected by its zero contour. Finally, once the triangulation's cells are classified, we determine which cells are active.

```
      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;

      mesh_classifier.reclassify();

      for (const auto &cell :
           dof_handler.active_cell_iterators() |
             [this](const typename DoFHandler<dim>::active_cell_iterator &cell) {
               return cell->is_locally_owned();
             })
        {
          if (mesh_classifier.location_to_level_set(cell) ==
              NonMatching::LocationToLevelSet::intersected)
            cell->set_active_fe_index(
              static_cast<types::fe_index>(ActiveFEIndex::lagrange));
          else
            cell->set_active_fe_index(
              static_cast<types::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() |
           [this](const typename DoFHandler<dim>::active_cell_iterator &cell) {
             return cell->is_locally_owned();
           })
      {
        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. 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.

```
  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;
  }
```

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

```
  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 matrix on MPI_rank=0... " << std::flush;
    Timer                  timer;
    DynamicSparsityPattern dsp(locally_relevant_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. This cell worker does not do anything for non-intersected cells.

```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::assemble()
  {
    pcout << "Assembling ... " << std::flush;
    TimerOutput::Scope t(computing_timer, "assembly");
    Timer              timer;
    using IteratorType     = typename DoFHandler<dim>::active_cell_iterator;
    const auto cell_worker = [&](const IteratorType &cell,
                                 ScratchData<dim>   &scratch_data,
                                 CopyData<dim>      &copy_data) {
      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())
                      {
                        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. 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. Next, the copier worker distributes the local contributions from the CopyData taking into account the constraints. Finally, 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.

```
          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.assemble_cell_worker(cell,
                                                        scratch_data,
                                                        copy_data);
            }
        }
    };

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

    ScratchData<dim> scratch_data(mapping,
                                  fe_collection,
                                  mesh_classifier,
                                  level_set_dof_handler,
                                  level_set,
                                  surface_update_flags,
                                  cell_quadrature,
                                  quadrature_1D);

    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;
    }
```

In the following, we solve the resulting linear system of equations. We either use a direct solver or AMG.

```
    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 including the stabilization term. At the end, we collect the results, and print them out.

```
    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.evaluate_cell_worker(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;
          }
      };

      ScratchData<dim> scratch_data(mapping,
                                    fe_collection,
                                    mesh_classifier,
                                    level_set_dof_handler,
                                    level_set,
                                    surface_update_flags,
                                    cell_quadrature,
                                    quadrature_1D);

      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());
}
```

The following two methods performs VTK output of the preliminary mesh refinements for the geometry approximation and of the TraceFEM solution. The important difference between the two is that the non-intersected cells are excluded from the output saving considerable amount of time and storage.

```
template <int dim>
void LaplaceBeltramiSolver<dim>::output_level_set(unsigned int n)
```

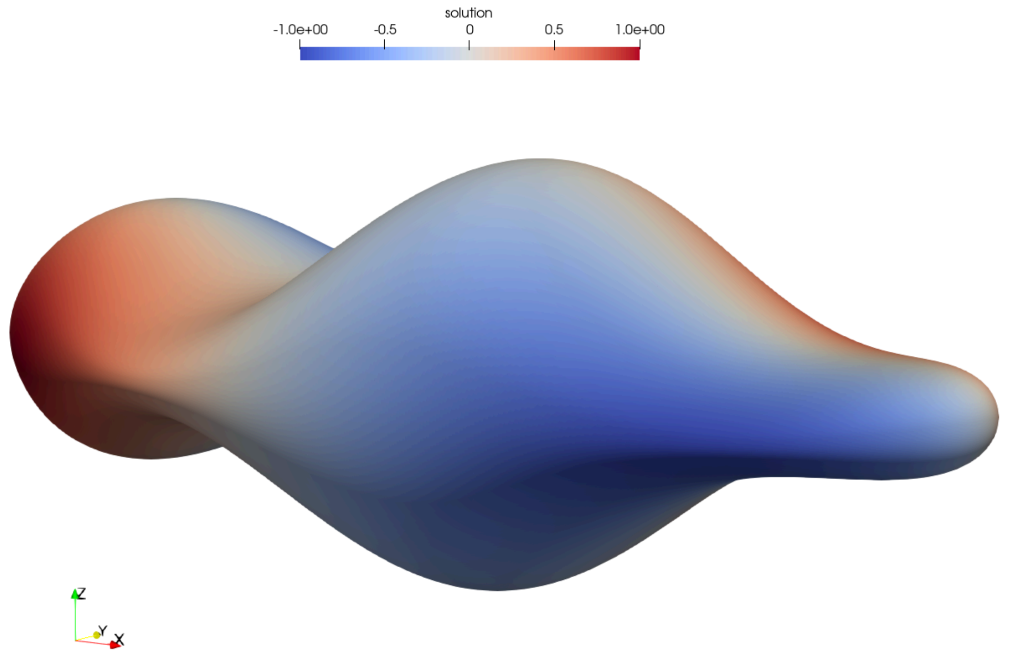
```
  {
    pcout << "Writing vtu file for surface on MPI_rank=0... " << std::flush;
    TimerOutput::Scope t(computing_timer, "output_level_set");
    Timer              timer;
    DataOut<dim>       data_out;
    data_out.add_data_vector(level_set_dof_handler, level_set, "level_set");
    data_out.add_data_vector(activeFE_indicator,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

The method localize_surface() generates iteratively a surface approximation as described above. Once the surface approximation is constructed, the main logic of the solver is executed as presented in the method **run()**.

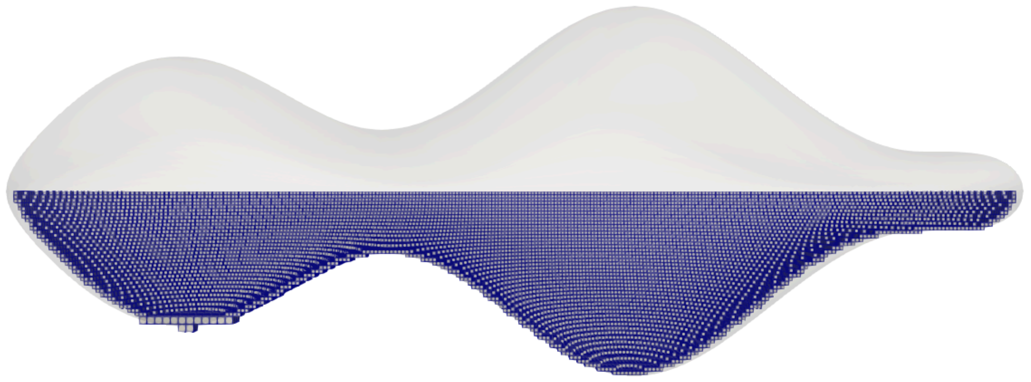
```
  template <int dim>
  void LaplaceBeltramiSolver<dim>::localize_surface()
  {
    unsigned int preliminary_levels = 3;
    for (unsigned int localization_cycle = 0;
         localization_cycle < preliminary_levels;
         ++localization_cycle)
      {
        pcout << std::endl
              << "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();
        computing_timer.print_summary();
        computing_timer.reset();
        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 u_h for a very fine mesh Γ_h is shown below by plotting in Paraview the zero contour of the approximate level set ψ_h and restricting the discrete solution u_h to the resulting surface approximation Γ_h.



Next, we demonstrate the corresponding set of intersected cells with active degrees of freedom. Note that not all cells are of the same refinement level which is attributed to the insufficiently fine initial uniform grid.



Convergence test

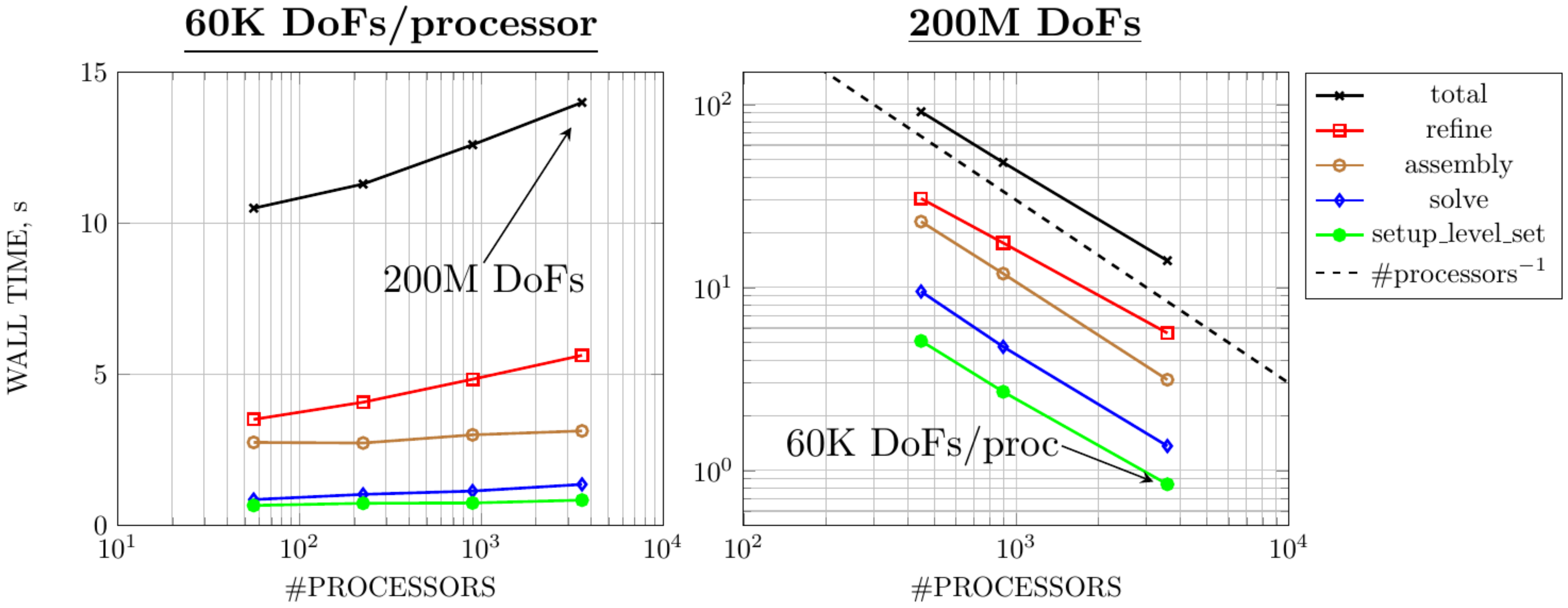
The results of the convergence study are shown in the following table.

Cycle	DOFS	Rate	Iterations	L^2-Error	EOC	H^1-Error	EOC	$s_h^{1/2}(u_h)$	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 $s_h^{1/2}(u_h) = \sqrt{s_h(u_h, u_h)}$ evaluated at the solution u_h.

Parallel scalability

The weak and strong scalability test results are shown in the following figure. Clearly, the **refine()** method is responsible for the certain lack of parallel scalability.



The plain program

```
/* ---------------------------------------------------------------------
 *
 * Copyright (C) 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, Clemson
 * University, 2023.
 */

#include <deal.II/base/convergence_table.h>
#include <deal.II/base/function.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/distributed/grid_refinement.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/dofs/dof_tools.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/fe/mapping_q1.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/sparse_direct.h>
#include <deal.II/lac/sparsity_pattern.h>
#include <deal.II/lac/sparsity_tools.h>
#include <deal.II/lac/trilinos_precondition.h>
#include <deal.II/lac/trilinos_solver.h>
#include <deal.II/lac/trilinos_sparse_matrix.h>
#include <deal.II/lac/trilinos_vector.h>
#include <deal.II/lac/vector.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/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>

using namespace dealii;
using VectorType = TrilinosWrappers::MPI::Vector;
using MatrixType = TrilinosWrappers::SparseMatrix;
namespace Step90
{
  enum class ActiveFEIndex : types::fe_index
  {
    lagrange = 0,
    nothing  = 1
  };

  template <int dim>
```

```
class TamarindShape : public Function<dim>
{
public:
  double value(const Point<dim> &point,
               const unsigned int /*component*/ = 0) const override
  {
    Assert(dim == 3, ExcNotImplemented());
    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;
    Assert(dim == 3, ExcNotImplemented());

    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;
    Assert(dim == 3, ExcNotImplemented());
    SymmetricTensor<2, dim> hessian;

    hessian[0][0] =
      0.5 +
      8.0 * 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) * 0.5 *
           numbers::PI * numbers::PI * std::sin(numbers::PI * point[0]));
    hessian[0][1] = 0.0;
    hessian[0][2] =
      (-8.0) * point[2] *
      std::pow(1.0 + 0.5 * std::sin(numbers::PI * point[0]), -3) *
      numbers::PI * std::cos(numbers::PI * point[0]);

    hessian[1][1] = 2.0;
    hessian[1][2] = 0.0;
```

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

      return hessian;
    }
  };

  template <int dim>
  class AnalyticalSolution : public Function<dim>
  {
  private:
    const 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;
  Assert(dim == 3, ExcNotImplemented());
  const Tensor<1, dim>          grad    = tamarind.gradient(point, component);
  const Tensor<1, dim>          normal  = (1.0 / grad.norm()) * grad;
  const SymmetricTensor<2, dim> hessian = 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]) * hessian[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 ScratchData
{
  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<1>                  &quadrature_edge,
              const UpdateFlags cell_update_flags = update_values |
                                                    update_gradients |
                                                    update_quadrature_points |
                                                    update_JxW_values)
    : fe_values(
        mapping,
        fe_collection[static_cast<types::fe_index>(ActiveFEIndex::lagrange)],
        quadrature,
        cell_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)
    {}

    ScratchData(const 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())
      , 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;
    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 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;
    }

    void assemble_cell_worker(
      const typename DoFHandler<dim>::active_cell_iterator &cell,
      ScratchData<dim>                                     &scratch_data,
      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);

      const 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];
        }
    }

    void evaluate_cell_worker(
      const typename DoFHandler<dim>::active_cell_iterator &cell,
      ScratchData<dim>                                     &scratch_data,
      CopyData<dim>                                        &copy_data)
    {
      double                     cell_stab_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);

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

  private:
    const double stabilization_parameter;
    const double stabilization_exponent;
  };
```

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

private:
  void make_grid();

  void localize_surface();

  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<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                    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;
  };

  template <int dim>
  LaplaceBeltramiSolver<dim>::LaplaceBeltramiSolver()
    : mpi_communicator(MPI_COMM_WORLD)
    , triangulation(mpi_communicator)
    , 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)
    , 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;
  cell_update_flags = update_default;
}

template <int dim>
void LaplaceBeltramiSolver<dim>::make_grid()
{
  pcout << std::endl
        << "Creating background mesh with MPI_Size="
        << Utilities::MPI::this_mpi_process(mpi_communicator) << std::endl;
  const double cube_side = 2.008901281;
  GridGenerator::hyper_cube(triangulation, -cube_side, cube_side);
  triangulation.refine_global(3);
}

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;

  mesh_classifier.reclassify();

  for (const auto &cell :
       dof_handler.active_cell_iterators() |
         [this](const typename DoFHandler<dim>::active_cell_iterator &cell) {
           return cell->is_locally_owned();
         })
    {
      if (mesh_classifier.location_to_level_set(cell) ==
          NonMatching::LocationToLevelSet::intersected)
        cell->set_active_fe_index(
          static_cast<types::fe_index>(ActiveFEIndex::lagrange));
      else
        cell->set_active_fe_index(
          static_cast<types::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() |
         [this](const typename DoFHandler<dim>::active_cell_iterator &cell) {
           return cell->is_locally_owned();
         })
    {
      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 matrix on MPI_rank=0... " << std::flush;
  Timer                  timer;
  DynamicSparsityPattern dsp(locally_relevant_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 ... " << std::flush;
  TimerOutput::Scope t(computing_timer, "assembly");
  Timer              timer;
  using IteratorType     = typename DoFHandler<dim>::active_cell_iterator;
  const auto cell_worker = [&](const IteratorType &cell,
                               ScratchData<dim>   &scratch_data,
                               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.assemble_cell_worker(cell,
                                                      scratch_data,
                                                      copy_data);
          }
      }
  };

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

  ScratchData<dim> scratch_data(mapping,
                                fe_collection,
                                mesh_classifier,
                                level_set_dof_handler,
                                level_set,
                                surface_update_flags,
                                cell_quadrature,
                                quadrature_1D);

  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.evaluate_cell_worker(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;
        }
    };

    ScratchData<dim> scratch_data(mapping,
                                  fe_collection,
                                  mesh_classifier,
                                  level_set_dof_handler,
                                  level_set,
                                  surface_update_flags,
                                  cell_quadrature,
                                  quadrature_1D);

    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();
        computing_timer.print_summary();
        computing_timer.reset();
        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