The step-87 tutorial program

This tutorial depends on **step-40**.

Table of contents

This program was contributed by Magdalena Schreter-Fleischhacker and Peter Munch. Many ideas presented here are the result of common code development with Maximilian Bergbauer, Marco Feder, Niklas Fehn, Johannes Heinz, Luca Heltai, Martin Kronbichler, Nils Much, and Judith Pauen.

This tutorial is loosely based on chapter 3.4 of the submitted PhD thesis "Matrix-free finite element computations at extreme scale and for challenging applications" by Peter Munch. Magdalena Schreter-Fleischhacker is funded by the Austrian Science Fund (FWF) Schrödinger Fellowship (J4577).

Introduction

This tutorial presents the advanced point-evaluation functionalities of deal.II, specifically useful for evaluating finite element solutions at arbitrary points. The underlying finite element mesh can be distributed among processes, which makes the operations more involved due to communication. In the examples discussed in this tutorial, we focus on the particular use case of point evaluation for distributed meshes, like **parallel::distributed::Triangulation**. Nevertheless, the application to non-distributed meshes is also possible.

Point evaluation

In the context of the finite element method (FEM), it is a common task to query the solution u at an arbitrary point $\boldsymbol{x}_q$ in the domain of interest Ω

$$u(\boldsymbol{x}_q) = \sum_i N_i(\boldsymbol{x}_q) u_i \quad \text{with} \quad i \in [0, n_{\text{dofs}}),$$

by evaluating the shape functions N_i at this point together with the corresponding solution coefficients u_i. After identification of the cell K where the arbitrary point $\boldsymbol{x}_q$ is inside, the transformation between $\boldsymbol{x}_q$ and the corresponding coordinates in the reference cell $\hat{\boldsymbol{x}}_q$ is obtained by the mapping $\boldsymbol{x}_q = \boldsymbol{F}_K(\hat{\boldsymbol{x}}_q)$. In this setting, the evaluation of the solution at an arbitrary point boils down to a cell-local evaluation

$$u(\boldsymbol{x}_q) = \sum_i \hat{N}_i^K(\hat{\boldsymbol{x}}_q) u_i^K \quad \text{with} \quad i \in [0, n_{\text{dofs\_per\_cell}}),$$

with $\hat{N}_i^K$ being the shape functions defined on the reference cell and u_i^K the solution coefficients restricted to the cell K.

Alternatively to point evaluation, evaluating weak-form (integration) operations of the type

$$u_i = (N_i(\boldsymbol{x}), u(\boldsymbol{x}))_\Omega = \int_\Omega N_i(\boldsymbol{x}) u(\boldsymbol{x}) dx = \sum_q N_i(\boldsymbol{x}_q)\, u(\boldsymbol{x}_q)\, |J(\boldsymbol{x}_q)|\, w(\boldsymbol{x}_q) \quad \text{with} \quad i \in [0, n_{\text{dofs}})$$

is possible, with $\boldsymbol{x}_q$ being quadrature points at arbitrary positions. After the values at the quadrature points have been multiplied by the integration weights, this operation can be interpreted as the transpose of the evaluation. Not surprisingly, such an operation can be also implemented as a cell loop.

Setup and communication

To determine the cell K and the reference position $\hat{\boldsymbol{x}}_q$ within the cell for a given point $\boldsymbol{x}_q$ on distributed meshes, deal.II performs a two-level-search approach. First, all processes that might possess the point are determined ("coarse search"). For this purpose, e.g., a distributed tree based on bounding boxes around locally owned domains using "ArborX" **[127]** is applied. After the potentially owning processes have been determined and the points have been sent to them as a request, one can start to find the cells that surround the points among locally owned cells ("fine search"). In order to accelerate this search, an R-tree from "boost::geometry" that is built around the vertices of the mesh is used.

Once the cell K that surrounds point $\boldsymbol{x}_q$ has been found, the reference position $\hat{\boldsymbol{x}}_q$ is obtained by performing the minimization:

$$\min_{\hat{\boldsymbol{x}}_q}(|\boldsymbol{F}_K(\hat{\boldsymbol{x}}_q) - \boldsymbol{x}_q|) \quad \text{with} \quad \hat{\boldsymbol{x}}_q \in [0,1]^{dim}.$$

With the determined pieces of information, the desired evaluation can be performed by the process that owns the cell. The result can now be communicated to the requesting process.

In summary, the coarse search determines, for each point, a list of processes that might own it. The subsequent fine search by each process determines whether the processes actually own these points by the sequence of request ("Does the process own the point?") and answer ("Yes."/"No."). Processes might post any number of point requests and communicate with any process. We propose to collect the point requests to a process to use the dynamic, sparse, scalable consensus-based communication algorithms [105], and to consider the obtained information to set up point-to-point communication patterns.

Implementation: Utilities::MPI::RemotePointEvaluation

The algorithm described above is implemented in **Utilities::MPI::RemotePointEvaluation** (short: "RPE") and related classes/functions. In this section, basic functionalities are briefly summarized. Their advanced capabilities will be shown subsequently based on concrete application cases.

The following code snippet shows the setup steps for the communication pattern:

```
std::vector<Point<dim>> points; // ... (filling of points not shown)

RemotePointEvaluation<dim> rpe;
rpe.reinit(points, triangulation, mapping);
```

All what is needed is a list of evaluation points and the mesh with a mapping.

The following code snippet shows the evaluation steps:

```
const std::function<void(const ArrayView<T> &, const CellData &)>
  evaluation_function;

std::vector<T> output;
rpe.evaluate_and_process(output, evaluation_function);
```

The user provides a function that processes the locally owned points. These values are communicated by **Utilities::MPI::RemotePointEvaluation**.

The relevant class during the local evaluation is **Utilities::MPI::RemotePointEvaluation::CellData**. It allows to loop over cells that surround the points. On these cells, a cell iterator and the positions in the reference cell of the request points can be queried. Furthermore, this class provides controlled access to the output vector of the **Utilities::MPI::RemotePointEvaluation::evaluate_and_process()** function.

```
for (const auto cell_index : cell_data.cell_indices())
  {
    const auto cell         = cell_data.get_active_cell_iterator(cell_index);
    const auto unit_points  = cell_data.get_unit_points(cell_index);
    const auto local_output = cell_data.get_data_view(cell_index, output);
  }
```

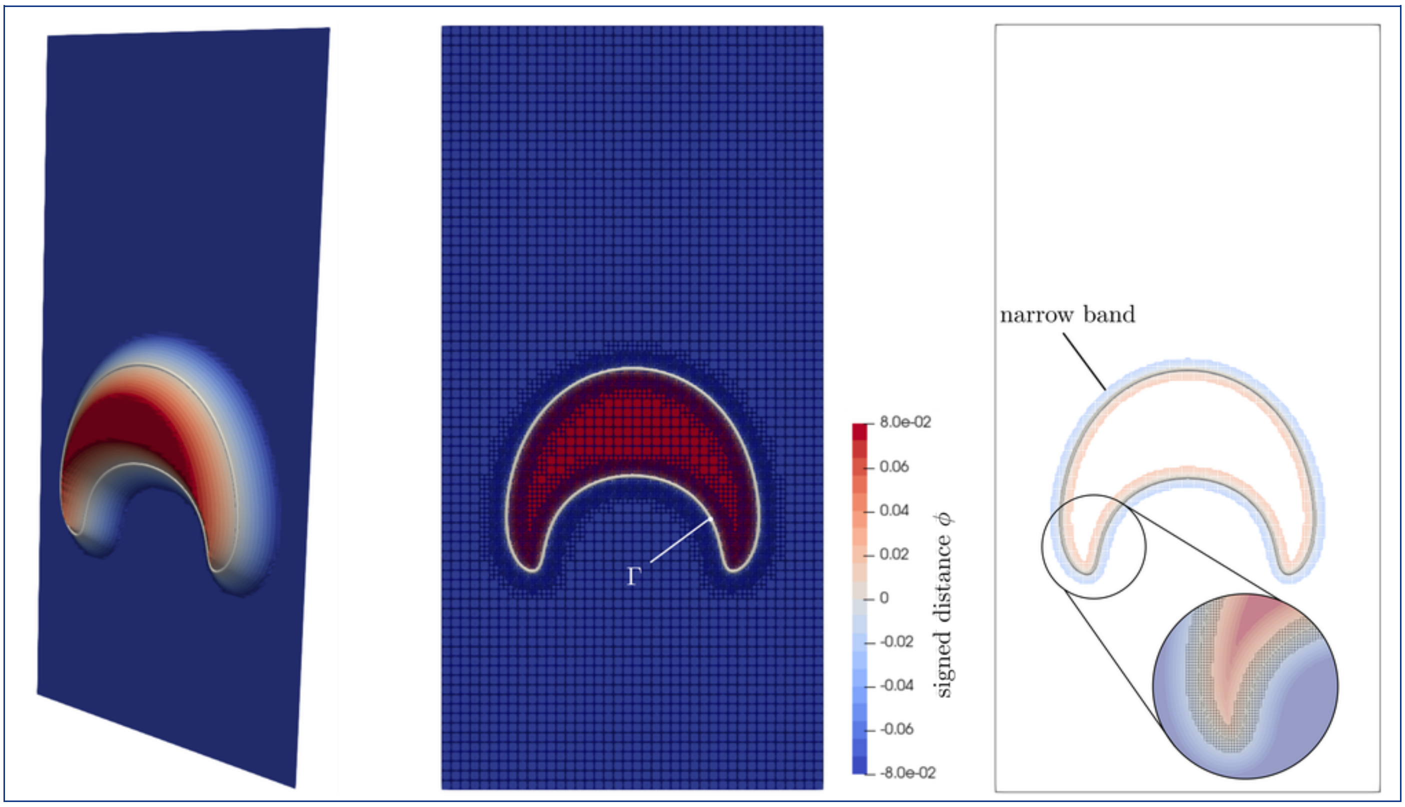
The functions

```
const auto evaluated_values =
  VectorTools::point_values<n_components>(rpe, dof_handler, vector);

const auto evaluated_gradients =
  VectorTools::point_gradients<n_components>(rpe, dof_handler, vector);
```

evaluate the values and gradients of a solution defined by **DoFHandler** and a vector at the request points. Internally, a lambda function is passed to **Utilities::MPI::RemotePointEvaluation**. Additionally it handles the special case, if points belong to multiple cells by taking, e.g., the average, the minimum, or the maximum via an optional argument of type **EvaluationFlags::EvaluationFlags**.

Motivation: two-phase flow

The exemplary mini codes presented in this tutorial are motivated by the application of two-phase-flow simulations formulated in a one-fluid setting using an Eulerian framework. In diffuse interface methods, the two phases may be implicitly described by a level-set function, here chosen as a signed distance function $\phi(\boldsymbol{x})$ in Ω and illustrated for a popular benchmark case of a rising bubble in the following figure.



The discrete interface Γ is represented implicitly through a certain isosurface of the level-set function e.g. for the signed-distance function $\Gamma = \{\boldsymbol{x} \in \Omega \mid \phi(\boldsymbol{x}) = 0\}$. We would like to note that deal.II provides a set of analytical signed distance functions for simple geometries in the **Functions::SignedDistance** namespace. Those can be combined via Boolean operations to describe more complex geometries [48]. The temporal evolution of the level-set field is obtained by the transport equation

$$\frac{\partial\,\phi}{\partial\,t} + \boldsymbol{u}|_\Gamma \cdot \nabla\phi = 0$$

with the transport velocity at the interface $\boldsymbol{u}|_\Gamma$, which might be approximated by the local fluid velocity $\boldsymbol{u}|_\Gamma \approx \boldsymbol{u}(\boldsymbol{x})$. To reobtain the signed-distance property of the level-set field throughout the numerical solution procedure, PDE-based or, alternatively, also geometric reinitialization methods, the latter via closest point projection [104], are used. In one of the mini examples, we describe how to obtain the closest point $\boldsymbol{x}^*$ to the interface Γ for an arbitrary point $\boldsymbol{x}$. For the simplest case, the former can be computed from the following equation

$$\boldsymbol{x}^* = \boldsymbol{x} - \boldsymbol{n}(\boldsymbol{x})\phi(\boldsymbol{x}),$$

assuming that the interface normal vector $\boldsymbol{n}(\boldsymbol{x})$ and $\phi(\boldsymbol{x})$ represent exact quantities. Typically, this projection is only performed for points located within a narrow band region around the interface, indicated in the right panel of the figure above.

Alternatively to the implicit representation of the interface, in sharp interface methods, e.g., via front tracking, the interface Γ is explicitly represented by a surface mesh. The latter is immersed into a background mesh, from which the local velocity at the support points of the surface mesh is extracted and leads to a movement of the support points of the immersed mesh as

$$\boldsymbol{x}_q^{(i+1)} = \boldsymbol{x}_q^{(i)} + \Delta t \cdot \boldsymbol{u}(\boldsymbol{x}_q^{(i)}) \quad \text{for } \boldsymbol{x}_q \in \Gamma$$

considering an explicit Euler time integration scheme from time step i to $i+1$ with time step-size Δt.

For a two-phase-flow model considering the incompressible Navier-Stokes equations, the two phases are usually coupled by a singular surface-tension force $\boldsymbol{F}_S$, which results, together with the difference in fluid properties, in discontinuities across the interface:

$$\boldsymbol{F}_S(\boldsymbol{x}) = \sigma\kappa(\boldsymbol{x})\boldsymbol{n}(\boldsymbol{x})\delta_\Gamma(\boldsymbol{x}).$$

Here σ represents the surface-tension coefficient, $\boldsymbol{n}(\boldsymbol{x})$ the interface normal vector and $\kappa(\boldsymbol{x})$ the interface mean curvature field. The singularity at the interface is imposed by the Dirac delta function

$$\delta_\Gamma(\boldsymbol{x}) = \begin{cases} 1 & \text{on } \Gamma \\ 0 & \text{else} \end{cases}$$

with support on the interface Γ. In a finite element context, the weak form of the surface-tension force is needed. The latter can be applied as a sharp surface-tension force model

$$(\boldsymbol{v}, \boldsymbol{F}_S)_\Omega = (\boldsymbol{v}, \sigma\kappa\boldsymbol{n})_\Gamma,$$

exploiting the property of the Dirac delta function for any smooth function v, i.e., $\int_\Omega \delta_\Gamma\, v \,\mathrm{d}x = \int_\Gamma v \,\mathrm{d}y$. For front-tracking methods, the curvature and the normal vector are directly computed from the surface mesh.

Alternatively, in regularized surface-tension-force models [36] [152] [126], the Dirac delta function is approximated by a smooth ansatz

$$(\boldsymbol{v}, \boldsymbol{F}_S)_\Omega \approx (\boldsymbol{v}, \sigma\kappa\boldsymbol{n}\|\nabla H\|)_\Omega$$

considering the absolute value of the gradient of a regularized indicator function $\|\nabla H\|$, which is related to the level-set field. In such models, the interface normal vector

$$\boldsymbol{n}(\boldsymbol{x}) = \nabla\phi(\boldsymbol{x}),$$

and the interface curvature field

$$\kappa(\boldsymbol{x}) = \nabla \cdot \boldsymbol{n}(\boldsymbol{x}) = \Delta\phi(\boldsymbol{x})\,.$$

are derived from the level-set function.

Overview

In the following, we present three simple use cases of **Utilities::MPI::RemotePointEvaluation** (mini examples). We start with discussing a serial code in mini example 0. In the subsequent mini examples, advanced problems are solved on distributed meshes:

- mini example 1: we evaluate values and user quantities along a line;
- mini example 2: we perform a closest-point projection within a narrow band, based on a level-set function, use the information to update the distance and to perform an extrapolation from the interface;
- mini example 3: we compute the surface-tension term sharply with the interface given by an codim-1 mesh, which is advected by the velocity from the background mesh (front tracking; solution transfer between a background mesh and an immersed surface mesh).

The commented program

Include files

The program starts with including all the relevant header files.

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function_lib.h>
#include <deal.II/base/function_signed_distance.h>
#include <deal.II/base/mpi.h>
#include <deal.II/base/mpi.templates.h>

#include <deal.II/distributed/tria.h>

#include <deal.II/dofs/dof_renumbering.h>

#include <deal.II/fe/fe_nothing.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/fe/mapping_q_cache.h>

#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>

#include <deal.II/lac/generic_linear_algebra.h>

#include <deal.II/numerics/data_out.h>

#include <iostream>
```

The files most relevant for this tutorial are the ones that contain **Utilities::MPI::RemotePointEvaluation** and the distributed evaluation functions in the **VectorTools** namespace, which use **Utilities::MPI::RemotePointEvaluation**.

```
#include <deal.II/base/mpi_remote_point_evaluation.h>
#include <deal.II/numerics/vector_tools.h>
```

The following header file provides the class **FEPointEvaluation**, which allows us to evaluate values of a local solution vector at arbitrary unit points of a cell.

```
#include <deal.II/matrix_free/fe_point_evaluation.h>
```

In the following, we define utility functions that are used below in the mini examples.

We pack everything that is specific for this program into a namespace of its own.

```
namespace Step87
{
  using namespace dealii;
```

The minimum requirement of this tutorial is MPI. If deal.II is built with p4est, we use **parallel::distributed::Triangulation** as distributed mesh. The class **parallel::shared::Triangulation** is used if deal.II is built without p4est or if the dimension of the triangulation is 1D, e.g., in the case of codim-1 meshes.

```
#ifdef DEAL_II_WITH_P4EST
  template <int dim, int spacedim = dim>
  using DistributedTriangulation = typename std::conditional<
    dim == 1,
    parallel::shared::Triangulation<dim, spacedim>,
    parallel::distributed::Triangulation<dim, spacedim>>::type;
#else
  template <int dim, int spacedim = dim>
  using DistributedTriangulation =
    parallel::shared::Triangulation<dim, spacedim>;
```

```
#endif
```

A list of points along a line is created by definition of a start point `p0`, an end point `p1`, and the number of subdivisions `n_subdivisions`.

```
  template <int spacedim>
  std::vector<Point<spacedim>>
  create_points_along_line(const Point<spacedim> &p0,
                           const Point<spacedim> &p1,
                           const unsigned int     n_subdivisions)
  {
    Assert(n_subdivisions >= 1, ExcInternalError());

    std::vector<Point<spacedim>> points;
    points.reserve(n_subdivisions + 1);

    points.emplace_back(p0);
    for (unsigned int i = 1; i < n_subdivisions; ++i)
      points.emplace_back(p0 + (p1 - p0) * static_cast<double>(i) /
                                 static_cast<double>(n_subdivisions));
    points.emplace_back(p1);

    return points;
  }
```

A given list of `points` and the corresponding values `values_0` and `values_1` (optional) are printed column-wise to a file `file_name`. In addition, the first column represents the distance of the points from the first point.

```
  template <int spacedim, typename T0, typename T1 = int>
  void print_along_line(const std::string                  &file_name,
                        const std::vector<Point<spacedim>> &points,
                        const std::vector<T0>              &values_0,
                        const std::vector<T1>              &values_1 = {})
  {
    AssertThrow(points.size() == values_0.size() &&
                  (values_1.size() == points.size() || values_1.empty()),
                ExcMessage("The provided vectors must have the same length."));

    std::ofstream file(file_name);

    for (unsigned int i = 0; i < points.size(); ++i)
      {
        file << std::fixed << std::right << std::setw(5) << std::setprecision(3)
             << points[0].distance(points[i]);

        for (unsigned int d = 0; d < spacedim; ++d)
          file << std::fixed << std::right << std::setw(10)
               << std::setprecision(3) << points[i][d];

        file << std::fixed << std::right << std::setw(10)
             << std::setprecision(3) << values_0[i];

        if (!values_1.empty())
          file << std::fixed << std::right << std::setw(10)
               << std::setprecision(3) << values_1[i];
        file << std::endl;
      }
  }
```

Create a unique list of the real coordinates of support points into `support_points` from the provided **Mapping** `mapping` and the **DoFHandler** `dof_handler`.

```
  template <int dim, int spacedim>
  void collect_support_points(
    const Mapping<dim, spacedim>               &mapping,
    const DoFHandler<dim, spacedim>            &dof_handler,
    LinearAlgebra::distributed::Vector<double> &support_points)
  {
    support_points.reinit(dof_handler.locally_owned_dofs(),
                          DoFTools::extract_locally_active_dofs(dof_handler),
                          dof_handler.get_communicator());

    const auto &fe = dof_handler.get_fe();

    FEValues<dim, spacedim> fe_values(
      mapping,
      fe,
      fe.base_element(0).get_unit_support_points(),
      update_quadrature_points);

    std::vector<types::global_dof_index> local_dof_indices(
      fe.n_dofs_per_cell());

    for (const auto &cell : dof_handler.active_cell_iterators() |
                              IteratorFilters::LocallyOwnedCell())
      {
        fe_values.reinit(cell);
        cell->get_dof_indices(local_dof_indices);

        for (const auto q : fe_values.quadrature_point_indices())
          for (unsigned int c = 0; c < spacedim; ++c)
            support_points[local_dof_indices[fe.component_to_system_index(c,
                                                                          q)]] =
              fe_values.quadrature_point(q)[c];
      }
  }
```

From the provided **Mapping** `mapping` and the **DoFHandler** `dof_handler` collect the global DoF indices and corresponding support points within a narrow band around the zero-level-set isosurface. Thereto, the value of the finite element function `signed_distance` corresponding to the **DoFHandler** `dof_handler_support_points` is evaluated at each support point. A support point is only collected if the absolute value is below the value for the `narrow_band_threshold`.

```
  template <int dim, int spacedim, typename T>
  std::tuple<std::vector<Point<spacedim>>, std::vector<types::global_dof_index>>
  collect_support_points_with_narrow_band(
    const Mapping<dim, spacedim>                &mapping,
    const DoFHandler<dim, spacedim>             &dof_handler_signed_distance,
    const LinearAlgebra::distributed::Vector<T> &signed_distance,
    const DoFHandler<dim, spacedim>             &dof_handler_support_points,
    const double                                 narrow_band_threshold)
  {
    AssertThrow(narrow_band_threshold >= 0,
                ExcMessage("The narrow band threshold"
                           " must be larger than or equal to 0."));
    const auto &tria = dof_handler_signed_distance.get_triangulation();
    const Quadrature<dim> quad(dof_handler_support_points.get_fe()
                                 .base_element(0)
                                 .get_unit_support_points());

    FEValues<dim> distance_values(mapping,
                                  dof_handler_signed_distance.get_fe(),
                                  quad,
                                  update_values);

    FEValues<dim> req_values(mapping,
                             dof_handler_support_points.get_fe(),
                             quad,
                             update_quadrature_points);

    std::vector<T>                       temp_distance(quad.size());
    std::vector<types::global_dof_index> local_dof_indices(
      dof_handler_support_points.get_fe().n_dofs_per_cell());

    std::vector<Point<dim>>              support_points;
    std::vector<types::global_dof_index> support_points_idx;

    const bool has_ghost_elements = signed_distance.has_ghost_elements();

    const auto &locally_owned_dofs_req =
      dof_handler_support_points.locally_owned_dofs();
    std::vector<bool> flags(locally_owned_dofs_req.n_elements(), false);

    if (has_ghost_elements == false)
      signed_distance.update_ghost_values();

    for (const auto &cell :
         tria.active_cell_iterators() | IteratorFilters::LocallyOwnedCell())
      {
        const auto cell_distance =
          cell->as_dof_handler_iterator(dof_handler_signed_distance);
        distance_values.reinit(cell_distance);
        distance_values.get_function_values(signed_distance, temp_distance);

        const auto cell_req =
          cell->as_dof_handler_iterator(dof_handler_support_points);
        req_values.reinit(cell_req);
        cell_req->get_dof_indices(local_dof_indices);

        for (const auto q : req_values.quadrature_point_indices())
          if (std::abs(temp_distance[q]) < narrow_band_threshold)
            {
              const auto idx = local_dof_indices[q];

              if (locally_owned_dofs_req.is_element(idx) == false ||
                  flags[locally_owned_dofs_req.index_within_set(idx)])
                continue;

              flags[locally_owned_dofs_req.index_within_set(idx)] = true;

              support_points_idx.emplace_back(idx);
              support_points.emplace_back(req_values.quadrature_point(q));
            }
      }

    if (has_ghost_elements == false)
      signed_distance.zero_out_ghost_values();

    return {support_points, support_points_idx};
  }
```

Convert a distributed vector of support points (`support_points_unrolled`) with a sequential order of the coordinates per point into a list of points.

```
  template <int spacedim>
  std::vector<Point<spacedim>> convert(
    const LinearAlgebra::distributed::Vector<double> &support_points_unrolled)
  {
    const unsigned int n_points =
      support_points_unrolled.locally_owned_size() / spacedim;

    std::vector<Point<spacedim>> points(n_points);

    for (unsigned int i = 0, c = 0; i < n_points; ++i)
      for (unsigned int d = 0; d < spacedim; ++d, ++c)
        points[i][d] = support_points_unrolled.local_element(c);

    return points;
  }
```

Mini example 0: Evaluation at given points for a serial mesh

In this introductory example, we demonstrate basic functionalities to evaluate solution quantities at arbitrary points on a serial (or locally owned) mesh. For this purpose, we first create the typical objects needed for a finite element discretization (defined by mapping, triangulation, and finite element) and a vector containing finite element solution coefficients.

```
  void example_0()
  {
    std::cout << "Running: example 0" << std::endl;

    constexpr unsigned int dim       = 2;
    constexpr unsigned int fe_degree = 3;

    MappingQ1<dim>     mapping;
    Triangulation<dim> tria;
    GridGenerator::subdivided_hyper_cube(tria, 7);

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

    Vector<double> vector(dof_handler.n_dofs());
    VectorTools::interpolate(
      mapping,
      dof_handler,
      Functions::SignedDistance::Sphere<dim>(Point<dim>(0.5, 0.5), 0.25),
      vector);
```

We create a list of points inside the domain at which we would like to evaluate the vector.

```
    const auto points_line =
      create_points_along_line(Point<dim>(0.0, 0.5), Point<dim>(1.0, 0.5), 20);
```

Now, we loop over all evaluation points. In the first step, we determine via **GridTools::find_active_cell_around_point()** the cell K that surrounds the point and translate the given real coordinate $\boldsymbol{x}$ to the corresponding coordinate on the unit cell $\hat{\boldsymbol{x}}_K$ according to the provided mapping. The resulting information is printed to the screen.

```
    std::vector<double> values_line;
    values_line.reserve(points_line.size());

    for (const auto &p_real : points_line)
      {
        const auto [cell, p_unit] =
          GridTools::find_active_cell_around_point(mapping,
                                                   dof_handler,
                                                   p_real);

        {
          AssertThrow(cell != dof_handler.end(), ExcInternalError());
          std::cout << " - Found point with real coordinates: " << p_real
                    << std::endl;
          std::cout << "   - in cell with vertices:";
          for (const auto &v : cell->vertex_indices())
            std::cout << " (" << cell->vertex(v) << ")";
          std::cout << std::endl;
          std::cout << "   - with coordinates on the unit cell: (" << p_unit
                    << ")" << std::endl;
        }
```

Now that we have determined K and $\hat{\boldsymbol{x}}_K$, we can perform the evaluation of the finite element solution at this point. In the following, we show three approaches for this purpose. In the first approach, we follow a traditional technique by using **FEValues** based on a cell-specific quadrature rule consisting of the unit point.

```
        std::cout << " - Values at point:" << std::endl;

        {
          FEValues<dim> fe_values(mapping,
                                  fe,
                                  Quadrature<dim>(p_unit),
                                  update_values);
          fe_values.reinit(cell);

          std::vector<double> quad_values(1);
          fe_values.get_function_values(vector, quad_values);
          const double value_0 = quad_values[0];
          std::cout << "  - " << value_0 << " (w. FEValues)" << std::endl;
          values_line.push_back(value_0);
        }
```

The second approach considers **FEPointEvaluation**, which directly takes a list of unit points for the subsequent evaluation.

```
        {
          std::vector<double> cell_vector(fe.n_dofs_per_cell());
          cell->get_dof_values(vector, cell_vector.begin(), cell_vector.end());

          FEPointEvaluation<1, dim> fe_point(mapping, fe, update_values);
          fe_point.reinit(cell, ArrayView<const Point<dim>>(p_unit));
          fe_point.evaluate(cell_vector, EvaluationFlags::values);
          const auto value_1 = fe_point.get_value(0);
          std::cout << "  - " << value_1 << " (w. FEPointEvaluation)"
                    << std::endl;
        }
```

Finally, in the third approach, the function **VectorTools::point_value()** is considered. It performs both the search of the surrounding cell and the evaluation at the requested point. However, its application is limited to a serial run of the code.

```
        {
          const auto value_2 =
            VectorTools::point_value(dof_handler, vector, p_real);
          std::cout << "  - " << value_2 << " (w. VectorTools::point_value())"
                    << std::endl;
          std::cout << std::endl;
        }
      }
```

We output the requested points together with the corresponding evaluated solution to a CSV file.

```
    std::cout << " - writing csv file" << std::endl;
    print_along_line("example_0_profile.csv", points_line, values_line);
  }
```

Obviously, the code above cannot work for distributed meshes, since the search (which might require communication) is called within a for-loop with loop bounds possibly different on each process. In the following code examples, we present the usage of arbitrary point evaluation in a parallel computation.

Mini example 1: Evaluation at given points on a distributed mesh

Just like in the introductory example, we evaluate the solution along a line, however, on a distributed mesh. We again start with setting up the objects needed for a finite element discretization.

```
  void example_1()
  {
    constexpr unsigned int dim       = 2;
    constexpr unsigned int fe_degree = 3;

    AssertDimension(dim, 2);

    ConditionalOStream pcout(std::cout,
                             Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) ==
                               0);

    pcout << "Running: example 1" << std::endl;

    MappingQ1<dim>                mapping;
    DistributedTriangulation<dim> tria(MPI_COMM_WORLD);
    GridGenerator::subdivided_hyper_cube(tria, 7);

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

We determine a finite element solution representing implicitly the geometry of a sphere with a radius of $r = 0.25$ and the center at $(0.5, 0.5)$ via a signed distance function.

```
    LinearAlgebra::distributed::Vector<double> signed_distance;
    signed_distance.reinit(dof_handler.locally_owned_dofs(),
                           DoFTools::extract_locally_active_dofs(dof_handler),
                           MPI_COMM_WORLD);

    VectorTools::interpolate(
      mapping,
      dof_handler,
      Functions::SignedDistance::Sphere<dim>(Point<dim>(0.5, 0.5), 0.25),
      signed_distance);
```

Next, we fill a vector from an arbitrary function that should represent a possible finite element solution, which we would like to evaluate.

```
    LinearAlgebra::distributed::Vector<double> solution;
    solution.reinit(dof_handler.locally_owned_dofs(),
                    DoFTools::extract_locally_active_dofs(dof_handler),
                    MPI_COMM_WORLD);

    VectorTools::interpolate(mapping,
                             dof_handler,
                             Functions::SignedDistance::Plane<dim>(
                               Point<dim>(), Point<dim>::unit_vector(0)),
                             solution);
```

We create a list of arbitrary (evaluation) points along a horizontal line, which intersects the center of the sphere. We do this only on the root rank, since we intend to output the results to a CSV file by the root rank.

```
    std::vector<Point<dim>> profile;
    if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
      profile = create_points_along_line(Point<dim>(0.0, 0.5),
                                         Point<dim>(1.0, 0.5),
                                         20);
```

Now, we can evaluate the results, e.g., for the signed distance function at all evaluation points in one go. First, we create a modifiable **Utilities::MPI::RemotePointEvaluation** object. We use **VectorTools::point_values()** by specifying the number of components of the solution vector (1 for the present example) as a template parameter. Within this function, the provided object for **Utilities::MPI::RemotePointEvaluation** is automatically reinitialized with the given points (profile). The ghost values of the to be read vector need to be updated from the user.

```
    Utilities::MPI::RemotePointEvaluation<dim, dim> rpe;

    signed_distance.update_ghost_values();
    const std::vector<double> profile_signed_distance =
      VectorTools::point_values<1>(
        mapping, dof_handler, signed_distance, profile, rpe);
    signed_distance.zero_out_ghost_values();
```

In addition to **VectorTools::point_values()**, function gradients can be evaluated via **VectorTools::point_gradient()**. However, for the computation of use-derived quantities, one might need to fall back to the direct usage of **Utilities::MPI::RemotePointEvaluation::evaluate_and_process()** or **Utilities::MPI::RemotePointEvaluation::process_and_evaluate()**. For the sake of demonstration, we use the former to evaluate the values of the solution vector at the request points. First, we define a lambda function for the operation on the surrounding cells. Using the **CellData** object, we can create a **FEPointEvaluation** object to evaluate the solution values at the cell-specific unit coordinates of the request points. Then, we assign the values to the result vector.

```
    const auto evaluate_function = [&](const ArrayView<double> &values,
                                       const auto              &cell_data) {
      FEPointEvaluation<1, dim> fe_point(mapping, fe, update_values);

      std::vector<double>                  local_values;
      std::vector<types::global_dof_index> local_dof_indices;

      for (const auto cell : cell_data.cell_indices())
        {
          const auto cell_dofs =
            cell_data.get_active_cell_iterator(cell)->as_dof_handler_iterator(
              dof_handler);

          const auto unit_points = cell_data.get_unit_points(cell);
          const auto local_value = cell_data.get_data_view(cell, values);

          local_values.resize(cell_dofs->get_fe().n_dofs_per_cell());
          cell_dofs->get_dof_values(solution,
                                    local_values.begin(),
                                    local_values.end());

          fe_point.reinit(cell_dofs, unit_points);
          fe_point.evaluate(local_values, EvaluationFlags::values);

          for (unsigned int q = 0; q < unit_points.size(); ++q)
            local_value[q] = fe_point.get_value(q);
        }
    };
```

The lambda function is passed to **Utilities::MPI::RemotePointEvaluation::evaluate_and_process()**, where the values are processed accordingly and stored within the created output vector. Again, the ghost values of the vector to be read need to be updated by the user.

```
    std::vector<double> output;
    solution.update_ghost_values();
    rpe.evaluate_and_process<double>(output, evaluate_function);
    solution.zero_out_ghost_values();
```

Finally, we output all results: the mesh as a VTU file and the results along the line as a CSV file. You can import the CSV file in ParaView and compare the output with the native line plot of ParaView based on the VTU file.

```
    DataOut<dim> data_out;
    data_out.add_data_vector(dof_handler, signed_distance, "signed_distance");
    data_out.build_patches(mapping);
    data_out.write_vtu_in_parallel("example_1.vtu", MPI_COMM_WORLD);

    if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
      {
        std::cout << " - writing csv file" << std::endl;
        print_along_line("example_1_profile.csv",
                         profile,
                         profile_signed_distance,
                         output);
      }
  }
```

Mini example 2: Closest-point evaluation of a distributed mesh

In this mini example, we perform a closest-point projection for each support point of a mesh within a narrow band by iteratively solving for

$$\boldsymbol{x}^{(i+1)} = \boldsymbol{x}^{(i)} - \boldsymbol{n}(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)}).$$

Once the closest point is determined, we can compute the distance and extrapolate the values from the interface. Note that the demonstrated algorithm does not guarantee that the closest points are collinear (see discussion in [63]). For the latter, one might also need to perform a tangential correction, which we omit here to keep the discussion concise.

We start with creating the objects for the finite element representation of the background mesh.

```
  void example_2()
  {
    constexpr unsigned int dim       = 2;
    constexpr unsigned int fe_degree = 3;

    ConditionalOStream pcout(std::cout,
                             Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) ==
                               0);

    pcout << "Running: example 2" << std::endl;
    pcout << "  - create system" << std::endl;

    FE_Q<dim>                      fe(fe_degree);
    MappingQ1<dim>                 mapping;
    DistributedTriangulation<dim> tria(MPI_COMM_WORLD);
    GridGenerator::subdivided_hyper_cube(tria, 50);

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

We compute an exemplary finite element solution vector, based on an arbitrary function. In addition, a finite element function computed from a signed distance function represents the geometry of a sphere implicitly.

```
    LinearAlgebra::distributed::Vector<double> solution;
    solution.reinit(dof_handler.locally_owned_dofs(),
                    DoFTools::extract_locally_active_dofs(dof_handler),
                    MPI_COMM_WORLD);

    VectorTools::interpolate(mapping,
                             dof_handler,
                             Functions::SignedDistance::Plane<dim>(
                               Point<dim>(), Point<dim>::unit_vector(0)),
                             solution);

    LinearAlgebra::distributed::Vector<double> signed_distance;
    signed_distance.reinit(dof_handler.locally_owned_dofs(),
                           DoFTools::extract_locally_active_dofs(dof_handler),
                           MPI_COMM_WORLD);

    VectorTools::interpolate(
      mapping,
      dof_handler,
      Functions::SignedDistance::Sphere<dim>(Point<dim>(0.5, 0.5), 0.25),
      signed_distance);
    signed_distance.update_ghost_values();
```

In the next step, we collect the points in the narrow band around the zero-level-set isosurface for which we would like to perform a closest point projection. To this end, we loop over all support points and collect the coordinates and the DoF indices of those with a maximum distance of 0.1 from the zero-level-set isosurface.

```
    pcout << "  - determine narrow band" << std::endl;

    const auto [support_points, support_points_idx] =
      collect_support_points_with_narrow_band(mapping,
                                              dof_handler,
                                              signed_distance,
                                              dof_handler,
                                              0.1 /*narrow_band_threshold*/);
```

For the iterative solution procedure of the closest-point projection, the maximum number of iterations and the tolerance for the maximum absolute acceptable change in the distance in one iteration are set.

```
    pcout << "  - determine closest point iteratively" << std::endl;
    constexpr int    max_iter     = 30;
    constexpr double tol_distance = 1e-6;
```

Now, we are ready to perform the algorithm by setting an initial guess for the projection points simply corresponding to the collected support points. We collect the global indices of the support points and the total number of points that need to be processed and do not fulfill the required tolerance. Those will be gradually reduced upon the iterative process.

```
    std::vector<Point<dim>> closest_points = support_points; // initial guess

    std::vector<unsigned int> unmatched_points_idx(closest_points.size());
    std::iota(unmatched_points_idx.begin(), unmatched_points_idx.end(), 0);

    int n_unmatched_points =
        Utilities::MPI::sum(unmatched_points_idx.size(), MPI_COMM_WORLD);
```

Now, we create a **Utilities::MPI::RemotePointEvaluation** cache object and start the loop for the fix-point iteration. We update the list of points that still need to be processed and subsequently pass this information to the **Utilities::MPI::RemotePointEvaluation** object. For the sake of illustration, we export the coordinates of the points to be updated for each iteration to a CSV file. Next, we can evaluate the signed distance function and the gradient at those points to update the current solution for the closest points. We perform the update only if the signed distance of the closest point is not already within the tolerance and register those points that still need to be processed.

```
      Utilities::MPI::RemotePointEvaluation<dim, dim> rpe;

      for (int it = 0; it < max_iter && n_unmatched_points > 0; ++it)
        {
          pcout << "     - iteration " << it << ": " << n_unmatched_points;

          std::vector<Point<dim>> unmatched_points(unmatched_points_idx.size());
          for (unsigned int i = 0; i < unmatched_points_idx.size(); ++i)
            unmatched_points[i] = closest_points[unmatched_points_idx[i]];

          const auto all_unmatched_points =
            Utilities::MPI::reduce<std::vector<Point<dim>>>(
              unmatched_points, MPI_COMM_WORLD, [](const auto &a, const auto &b) {
                auto result = a;
                result.insert(result.end(), b.begin(), b.end());
                return result;
              });

          if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
            {
              std::ofstream file("example_2_" + std::to_string(it) + ".csv");
              for (const auto &p : all_unmatched_points)
                file << p << std::endl;
              file.close();
            }

          rpe.reinit(unmatched_points, tria, mapping);

          AssertThrow(rpe.all_points_found(),
                      ExcMessage("Processed point is outside domain."));

          const auto eval_values =
            VectorTools::point_values<1>(rpe, dof_handler, signed_distance);

          const auto eval_gradient =
            VectorTools::point_gradients<1>(rpe, dof_handler, signed_distance);

          std::vector<unsigned int> unmatched_points_idx_next;

          for (unsigned int i = 0; i < unmatched_points_idx.size(); ++i)
            if (std::abs(eval_values[i]) > tol_distance)
              {
                closest_points[unmatched_points_idx[i]] -=
                  eval_values[i] * eval_gradient[i];

                unmatched_points_idx_next.emplace_back(unmatched_points_idx[i]);
              }

          unmatched_points_idx.swap(unmatched_points_idx_next);

          n_unmatched_points =
            Utilities::MPI::sum(unmatched_points_idx.size(), MPI_COMM_WORLD);

          pcout << " -> " << n_unmatched_points << std::endl;
        }
```

We print a warning message if we exceed the maximum number of allowed iterations and if there are still projection points with a distance value exceeding the tolerance.

```
      if (n_unmatched_points > 0)
        pcout << "WARNING: The tolerance of " << n_unmatched_points
              << " points is not yet attained." << std::endl;
```

As a result, we obtain a list of support points and corresponding closest points at the zero-isosurface level set. This information can be used to update the signed distance function, i.e., the reinitialization the values of the level-set function to maintain the signed distance property **[104]**.

```
      pcout << "  - determine distance in narrow band" << std::endl;
      LinearAlgebra::distributed::Vector<double> solution_distance;
      solution_distance.reinit(solution);

      for (unsigned int i = 0; i < closest_points.size(); ++i)
        solution_distance[support_points_idx[i]] =
          support_points[i].distance(closest_points[i]);
```

In addition, we use the information of the closest point to extrapolate values from the interface, i.e., the zero-level set isosurface, to the support points within the narrow band. This might be helpful to improve accuracy, e.g., for diffuse interface fluxes where certain quantities are only accurately determined at the interface (e.g. curvature for surface tension **[63]**).

```
      pcout << "  - perform extrapolation in narrow band" << std::endl;
      rpe.reinit(closest_points, tria, mapping);
```

```
    solution.update_ghost_values();
    const auto vals = VectorTools::point_values<1>(rpe, dof_handler, solution);
    solution.zero_out_ghost_values();

    LinearAlgebra::distributed::Vector<double> solution_extrapolated;
    solution_extrapolated.reinit(solution);

    for (unsigned int i = 0; i < closest_points.size(); ++i)
      solution_extrapolated[support_points_idx[i]] = vals[i];
```

Finally, we output the results to a VTU file.

```
    pcout << "  - output results" << std::endl;
    DataOut<dim> data_out;
    data_out.add_data_vector(dof_handler, signed_distance, "signed_distance");
    data_out.add_data_vector(dof_handler, solution, "solution");
    data_out.add_data_vector(dof_handler,
                             solution_distance,
                             "solution_distance");
    data_out.add_data_vector(dof_handler,
                             solution_extrapolated,
                             "solution_extrapolated");
    data_out.build_patches(mapping);
    data_out.write_vtu_in_parallel("example_2.vtu", MPI_COMM_WORLD);

    pcout << std::endl;
  }
```

Mini example 3: Sharp interface method on the example of surface tension for front tracking

The final mini example presents a basic implementation of front tracking [157], [183] of a surface mesh $\mathbb{T}_\Gamma$ immersed in an Eulerian background fluid mesh $\mathbb{T}_\Omega$.

We assume that the immersed surface is transported according to a prescribed velocity field from the background mesh. Subsequently, we perform a sharp computation of the surface-tension force:

$$(\boldsymbol{v}_i(\boldsymbol{x}), \boldsymbol{F}_S(\boldsymbol{x}))_\Omega = (\boldsymbol{v}_i(\boldsymbol{x}), \sigma(\boldsymbol{x})\kappa(\boldsymbol{x})\boldsymbol{n}(\boldsymbol{x}))_\Gamma \approx \sum_{q\in\mathbb{T}_\Gamma} \boldsymbol{v}_i^T(\boldsymbol{x}_q)\sigma(\boldsymbol{x}_q)\kappa(\boldsymbol{x}_q)\boldsymbol{n}(\boldsymbol{x}_q)|J(\boldsymbol{x}_q)|w(\boldsymbol{x}_q) \quad \forall i \in \mathbb{T}_\Omega.$$

We decompose this operation into two steps. In the first step, we evaluate the force contributions $\sigma(\boldsymbol{x}_q)\kappa(\boldsymbol{x}_q)\boldsymbol{n}(\boldsymbol{x}_q)$ at the quadrature points defined on the immersed mesh and multiply them with the mapped quadrature weight $|J(\boldsymbol{x}_q)|w_q$:

$$\boldsymbol{F}_S(\boldsymbol{x}_q) \leftarrow \sigma(\boldsymbol{x}_q)\kappa(\boldsymbol{x}_q)\boldsymbol{n}(\boldsymbol{x}_q)|J(\boldsymbol{x}_q)|w_q \quad \forall q \in \mathbb{T}_\Gamma.$$

In the second step, we compute the discretized weak form by multiplying with test functions on the background mesh:

$$(\boldsymbol{v}_i(\boldsymbol{x}), \boldsymbol{F}_S(\boldsymbol{x}))_\Omega \leftarrow \sum_q \boldsymbol{v}_i^T(\boldsymbol{x}_q)\boldsymbol{F}_S(\boldsymbol{x}_q) \quad \forall i \in \mathbb{T}_\Omega.$$

Obviously, we need to communicate between the two steps. The second step can be handled completely by **Utilities::MPI::RemotePointEvaluation**, which provides the function **Utilities::MPI::RemotePointEvaluation::process_and_evaluate()** for this purpose.

We start with setting the parameters consisting of the polynomial degree of the shape functions, the dimension of the background mesh, the time-step size to be considered for transporting the surface mesh and the number of time steps.

```
  void example_3()
  {
    const unsigned int degree       = 3;
    const unsigned int dim          = 2;
    const double       dt           = 0.2;
    const unsigned int n_time_steps = 10;

    ConditionalOStream pcout(std::cout,
                             Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) ==
                               0);

    pcout << "Running: example 3" << std::endl;
```

Next, we create the standard objects necessary for the finite element representation of the background mesh

```
    pcout << "  - creating background mesh" << std::endl;
    DistributedTriangulation<dim> tria_background(MPI_COMM_WORLD);
    GridGenerator::hyper_cube(tria_background);
    tria_background.refine_global(5);

    MappingQ1<dim>  mapping_background;
    FESystem<dim>   fe_background(FE_Q<dim>(degree), dim);
    DoFHandler<dim> dof_handler_background(tria_background);
    dof_handler_background.distribute_dofs(fe_background);
```

and, similarly, for the immersed surface mesh. We use a sphere with radius $r = 0.75$ which is placed in the center of the top half of the cubic background domain.

```
    pcout << "  - creating immersed mesh" << std::endl;
    const Point<dim> center(0.5, 0.75);
    const double     radius = 0.15;

    DistributedTriangulation<dim - 1, dim> tria_immersed(MPI_COMM_WORLD);
    GridGenerator::hyper_sphere(tria_immersed, center, radius);
    tria_immersed.refine_global(4);
```

Two different mappings are considered for the immersed surface mesh: one valid for the initial configuration and one that is updated in every time step according to the nodal displacements. Two types of finite elements are used to represent scalar and vector-valued DoF values.

```
    MappingQ<dim - 1, dim>      mapping_immersed_base(3);
    MappingQCache<dim - 1, dim> mapping_immersed(3);
    mapping_immersed.initialize(mapping_immersed_base, tria_immersed);
    QGauss<dim - 1> quadrature_immersed(degree + 1);

    FE_Q<dim - 1, dim>       fe_scalar_immersed(degree);
    FESystem<dim - 1, dim>   fe_immersed(fe_scalar_immersed, dim);
    DoFHandler<dim - 1, dim> dof_handler_immersed(tria_immersed);
    dof_handler_immersed.distribute_dofs(fe_immersed);
```

We renumber the DoFs related to the vector-valued problem to simplify access to the individual components.

```
    DoFRenumbering::support_point_wise(dof_handler_immersed);
```

We fill a DoF vector on the background mesh with an analytical velocity field considering the Rayleigh-Kothe vortex flow and initialize a DoF vector for the weak form of the surface-tension force.

```
    LinearAlgebra::distributed::Vector<double> velocity;
    velocity.reinit(dof_handler_background.locally_owned_dofs(),
                    DoFTools::extract_locally_active_dofs(
                      dof_handler_background),
                    MPI_COMM_WORLD);
    Functions::RayleighKotheVortex<dim> vortex(2);

    LinearAlgebra::distributed::Vector<double> force_vector(
      dof_handler_background.locally_owned_dofs(),
      DoFTools::extract_locally_active_dofs(dof_handler_background),
      MPI_COMM_WORLD);
```

Next, we collect the real positions $\boldsymbol{x}_q$ of the quadrature points of the surface mesh in a vector.

```
    LinearAlgebra::distributed::Vector<double> immersed_support_points;
    collect_support_points(mapping_immersed,
                           dof_handler_immersed,
                           immersed_support_points);
```

We initialize a **Utilities::MPI::RemotePointEvaluation** object and start the time loop. For any other step than the initial one, we first move the support points of the surface mesh according to the fluid velocity of the background mesh. Thereto, we first update the time of the velocity function. Then, we update the internal data structures of the **Utilities::MPI::RemotePointEvaluation** object with the collected support points of the immersed mesh. We throw an exception if one of the points cannot be found within the domain of the background mesh. Next, we evaluate the velocity at the surface-mesh support points and compute the resulting update of the coordinates. Finally, we update the mapping of the immersed surface mesh to the current position.

```
    Utilities::MPI::RemotePointEvaluation<dim> rpe;
    double                                     time = 0.0;
    for (unsigned int it = 0; it <= n_time_steps; ++it, time += dt)
      {
        pcout << "time: " << time << std::endl;
        if (it > 0)
          {
            pcout << "  - move support points (immersed mesh)" << std::endl;
            vortex.set_time(time);
            VectorTools::interpolate(mapping_background,
                                     dof_handler_background,
                                     vortex,
                                     velocity);
            rpe.reinit(convert<dim>(immersed_support_points),
                       tria_background,
                       mapping_background);

            AssertThrow(rpe.all_points_found(),
                        ExcMessage(
                          "Immersed domain leaves background domain!"));

            velocity.update_ghost_values();
            const auto immersed_velocity =
              VectorTools::point_values<dim>(rpe,
                                             dof_handler_background,
                                             velocity);
            velocity.zero_out_ghost_values();

            for (unsigned int i = 0, c = 0;
                 i < immersed_support_points.locally_owned_size() / dim;
                 ++i)
              for (unsigned int d = 0; d < dim; ++d, ++c)
                immersed_support_points.local_element(c) +=
                  immersed_velocity[i][d] * dt;

            mapping_immersed.initialize(mapping_immersed_base,
                                        dof_handler_immersed,
                                        immersed_support_points,
                                        false);
          }
```

Next, we loop over all locally owned cells of the immersed mesh and its quadrature points to compute the value for the local surface tension force contribution $\boldsymbol{F}_S(\boldsymbol{x}_q)$. We store the real coordinates of the quadrature points and the corresponding force contributions in two individual vectors. For computation of the latter, the normal vector $\boldsymbol{n}(\boldsymbol{x}_q)$ can be directly extracted from the surface mesh via **FEValues** and, for the curvature, we use the following approximation:

$$\kappa(\boldsymbol{x}_q) = \nabla \cdot \boldsymbol{n}(\boldsymbol{x}_q) = \operatorname{tr}\left(\nabla \boldsymbol{n}(\boldsymbol{x}_q)\right) \approx \operatorname{tr}\left(\nabla \sum_i \boldsymbol{N}_i(\boldsymbol{x}_q)\boldsymbol{n}_i\right) = \sum_i \operatorname{tr}\left(\nabla \boldsymbol{N}_i(\boldsymbol{x}_q)\boldsymbol{n}_i\right) \text{ with } i \in [0, n_{\text{dofs\_per\_cell}}),$$

which we can apply since the immersed mesh is consistently orientated. The surface tension coefficient is set to 1 for the sake of demonstration.

```
        pcout << "  - compute to be tested values (immersed mesh)" << std::endl;
        using value_type = Tensor<1, dim, double>;

        std::vector<Point<dim>> integration_points;
        std::vector<value_type> integration_values;

        FEValues<dim - 1, dim> fe_values(mapping_immersed,
                                         fe_immersed,
                                         quadrature_immersed,
                                         update_JxW_values | update_gradients |
                                           update_normal_vectors |
                                           update_quadrature_points);

        FEValues<dim - 1, dim> fe_values_co(
          mapping_immersed,
          fe_scalar_immersed,
          fe_scalar_immersed.get_unit_support_points(),
          update_JxW_values | update_normal_vectors);

        for (const auto &cell : tria_immersed.active_cell_iterators() |
                                  IteratorFilters::LocallyOwnedCell())
          {
            fe_values.reinit(cell);
            fe_values_co.reinit(cell);

            for (const auto &q : fe_values.quadrature_point_indices())
              {
                integration_points.emplace_back(fe_values.quadrature_point(q));

                const auto sigma = 1.0; // surface tension coefficient

                const auto normal = fe_values.normal_vector(q);

                double curvature = 0;
                for (unsigned int i = 0;
                     i < fe_scalar_immersed.n_dofs_per_cell();
                     ++i)
                  for (unsigned int d = 0; d < dim; ++d)
                    curvature +=
                      fe_values.shape_grad_component(
                        fe_immersed.component_to_system_index(d, i), q, d)[d] *
                      fe_values_co.normal_vector(i)[d];

                const auto FxJxW =
                  sigma * curvature * normal * fe_values.JxW(q);

                integration_values.emplace_back(FxJxW);
              }
          }
```

Before we evaluate the weak form of the surface-tension force, the communication pattern of **Utilities::MPI::RemotePointEvaluation** is set up from the quadrature points of the immersed mesh, determining the surrounding cells on the background mesh.

```
        pcout << "  - test values (background mesh)" << std::endl;

        rpe.reinit(integration_points, tria_background, mapping_background);
```

In preparation for utilizing **Utilities::MPI::RemotePointEvaluation::process_and_evaluate** that performs the multiplication with the test function, we set up a callback function that contains the operation on the intersected cells of the background mesh. Within this function, we initialize a **FEPointEvaluation** object that allows us to integrate values at arbitrary points within a cell. We loop over the cells that surround quadrature points of the immersed mesh – provided by the callback function. From the provided **CellData** object, we retrieve the unit points, i.e., the quadrature points of the immersed mesh that lie within the current cell and a pointer to the stored values on the current cell (local surface-tension force) for convenience. We reinitialize the data structure of **FEPointEvaluation** on every cell according to the unit points. Next, we loop over the quadrature points attributed to the cell and submit the local surface-tension force to the **FEPointEvaluation** object. Via **FEPointEvaluation::test_and_sum()**, the submitted values are multiplied by the values of the test function and a summation over all given points is performed. Subsequently, the contributions are assembled into the global vector containing the weak form of the surface-tension force.

```
        const auto integration_function = [&](const auto &values,
                                              const auto &cell_data) {
          FEPointEvaluation<dim, dim> phi_force(mapping_background,
                                                fe_background,
                                                update_values);

          std::vector<double>                  local_values;
          std::vector<types::global_dof_index> local_dof_indices;

          for (const auto cell : cell_data.cell_indices())
            {
              const auto cell_dofs =
                cell_data.get_active_cell_iterator(cell)
                  ->as_dof_handler_iterator(dof_handler_background);

              const auto unit_points = cell_data.get_unit_points(cell);
              const auto FxJxW       = cell_data.get_data_view(cell, values);

              phi_force.reinit(cell_dofs, unit_points);

              for (const auto q : phi_force.quadrature_point_indices())
                phi_force.submit_value(FxJxW[q], q);

              local_values.resize(cell_dofs->get_fe().n_dofs_per_cell());
              phi_force.test_and_sum(local_values, EvaluationFlags::values);

              local_dof_indices.resize(cell_dofs->get_fe().n_dofs_per_cell());
              cell_dofs->get_dof_indices(local_dof_indices);
              AffineConstraints<double>().distribute_local_to_global(
                local_values, local_dof_indices, force_vector);
            }
        };
```

The callback function is passed together with the vector holding the surface-tension force contribution at each quadrature point of the immersed mesh to **Utilities::MPI::RemotePointEvaluation::process_and_evaluate**. The only missing step is to compress the distributed force vector.

```
        rpe.process_and_evaluate<value_type>(integration_values,
                                             integration_function);
        force_vector.compress(VectorOperation::add);
```

After every tenth step or at the beginning/end of the time loop, we output the force vector and the velocity of the background mesh to a VTU file. In addition, we also export the geometry of the (deformed) immersed surface mesh to a separate VTU file.

```
        if (it % 10 == 0 || it == n_time_steps)
          {
            std::vector<
              DataComponentInterpretation::DataComponentInterpretation>
              vector_component_interpretation(
                dim, DataComponentInterpretation::component_is_part_of_vector);
            pcout << "  - write data (background mesh)" << std::endl;
            DataOut<dim>          data_out_background;
            DataOutBase::VtkFlags flags_backround;
            flags_backround.write_higher_order_cells = true;
            data_out_background.set_flags(flags_backround);
            data_out_background.add_data_vector(
              dof_handler_background,
              force_vector,
              std::vector<std::string>(dim, "force"),
              vector_component_interpretation);
            data_out_background.add_data_vector(
              dof_handler_background,
              velocity,
              std::vector<std::string>(dim, "velocity"),
              vector_component_interpretation);
            data_out_background.build_patches(mapping_background, 3);
            data_out_background.write_vtu_in_parallel("example_3_background_" +
                                                        std::to_string(it) +
                                                        ".vtu",
                                                      MPI_COMM_WORLD);

            pcout << "  - write mesh (immersed mesh)" << std::endl;
            DataOut<dim - 1, dim> data_out_immersed;
            data_out_immersed.attach_triangulation(tria_immersed);
            data_out_immersed.build_patches(mapping_immersed, 3);
            data_out_immersed.write_vtu_in_parallel("example_3_immersed_" +
                                                      std::to_string(it) +
                                                      ".vtu",
                                                    MPI_COMM_WORLD);
          }
        pcout << std::endl;
      }
  }
} // namespace Step87
```

Driver

Finally, the driver of the program executes the four mini examples.

```
int main(int argc, char **argv)
{
  using namespace dealii;
  Utilities::MPI::MPI_InitFinalize mpi(argc, argv, 1);
  std::cout.precision(5);

  if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
    Step87::example_0(); // only run on root process

  Step87::example_1();
  Step87::example_2();
  Step87::example_3();
}
```

Results

Mini example 0

We present a part of the screen output. It shows, for each point, the determined cell and reference position. Also, one can see that the values evaluated with **FEValues**, **FEPointEvaluation**, and **VectorTools::point_values()** are identical, as expected.

```
Running: example 0
 - Found point with real coordinates: 0 0.5
   - in cell with vertices: (0 0.4) (0.2 0.4) (0 0.6) (0.2 0.6)
   - with coordinates on the unit cell: (0 0.5)
 - Values at point:
  - 0.25002 (w. FEValues)
  - 0.25002 (w. FEPointEvaluation)
  - 0.25002 (w. VectorTools::point_value())

 - Found point with real coordinates: 0.05 0.5
   - in cell with vertices: (0 0.4) (0.2 0.4) (0 0.6) (0.2 0.6)
   - with coordinates on the unit cell: (0.25 0.5)
 - Values at point:
  - 0.20003 (w. FEValues)
  - 0.20003 (w. FEPointEvaluation)
  - 0.20003 (w. VectorTools::point_value())

...

 - Found point with real coordinates: 1 0.5
   - in cell with vertices: (0.8 0.4) (1 0.4) (0.8 0.6) (1 0.6)
   - with coordinates on the unit cell: (1 0.5)
 - Values at point:
  - 0.25002 (w. FEValues)
  - 0.25002 (w. FEPointEvaluation)
  - 0.25002 (w. VectorTools::point_value())

 - writing csv file
```

The CSV output is as follows and contains, in the first column, the distances with respect to the first point, the second and the third column represent the coordinates of the points and the fourth column the evaluated solution values at those points.

```
0.000     0.000     0.500     0.250
0.050     0.050     0.500     0.200
0.100     0.100     0.500     0.150
0.150     0.150     0.500     0.100
0.200     0.200     0.500     0.050
0.250     0.250     0.500     0.000
0.300     0.300     0.500    -0.050
0.350     0.350     0.500    -0.100
0.400     0.400     0.500    -0.149
0.450     0.450     0.500    -0.200
0.500     0.500     0.500    -0.222
0.550     0.550     0.500    -0.200
0.600     0.600     0.500    -0.149
0.650     0.650     0.500    -0.100
0.700     0.700     0.500    -0.050
0.750     0.750     0.500     0.000
0.800     0.800     0.500     0.050
0.850     0.850     0.500     0.100
0.900     0.900     0.500     0.150
0.950     0.950     0.500     0.200
1.000     1.000     0.500     0.250
```

Mini example 1

We show the screen output.

```
Running: example 1
 - writing csv file
```

The CSV output is as follows and identical to the results of the serial case presented in mini example 0. The fifth column shows the user quantity evaluated additionally in this mini example.

```
0.000     0.000     0.500     0.250     0.000
0.050     0.050     0.500     0.200     0.050
0.100     0.100     0.500     0.150     0.100
0.150     0.150     0.500     0.100     0.150
0.200     0.200     0.500     0.050     0.200
0.250     0.250     0.500     0.000     0.250
0.300     0.300     0.500    -0.050     0.300
0.350     0.350     0.500    -0.100     0.350
0.400     0.400     0.500    -0.149     0.400
```

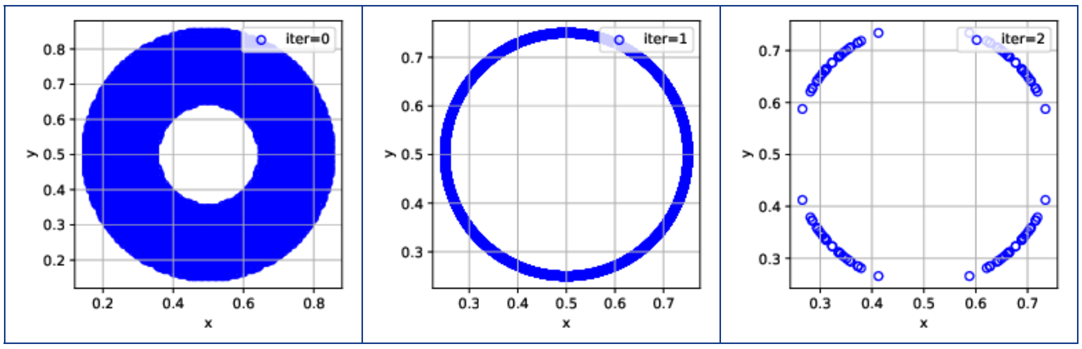
```
0.450     0.450     0.500    -0.200     0.450
0.500     0.500     0.500    -0.222     0.500
0.550     0.550     0.500    -0.200     0.550
0.600     0.600     0.500    -0.149     0.600
0.650     0.650     0.500    -0.100     0.650
0.700     0.700     0.500    -0.050     0.700
0.750     0.750     0.500     0.000     0.750
0.800     0.800     0.500     0.050     0.800
0.850     0.850     0.500     0.100     0.850
0.900     0.900     0.500     0.150     0.900
0.950     0.950     0.500     0.200     0.950
1.000     1.000     0.500     0.250     1.000
```

Mini example 2

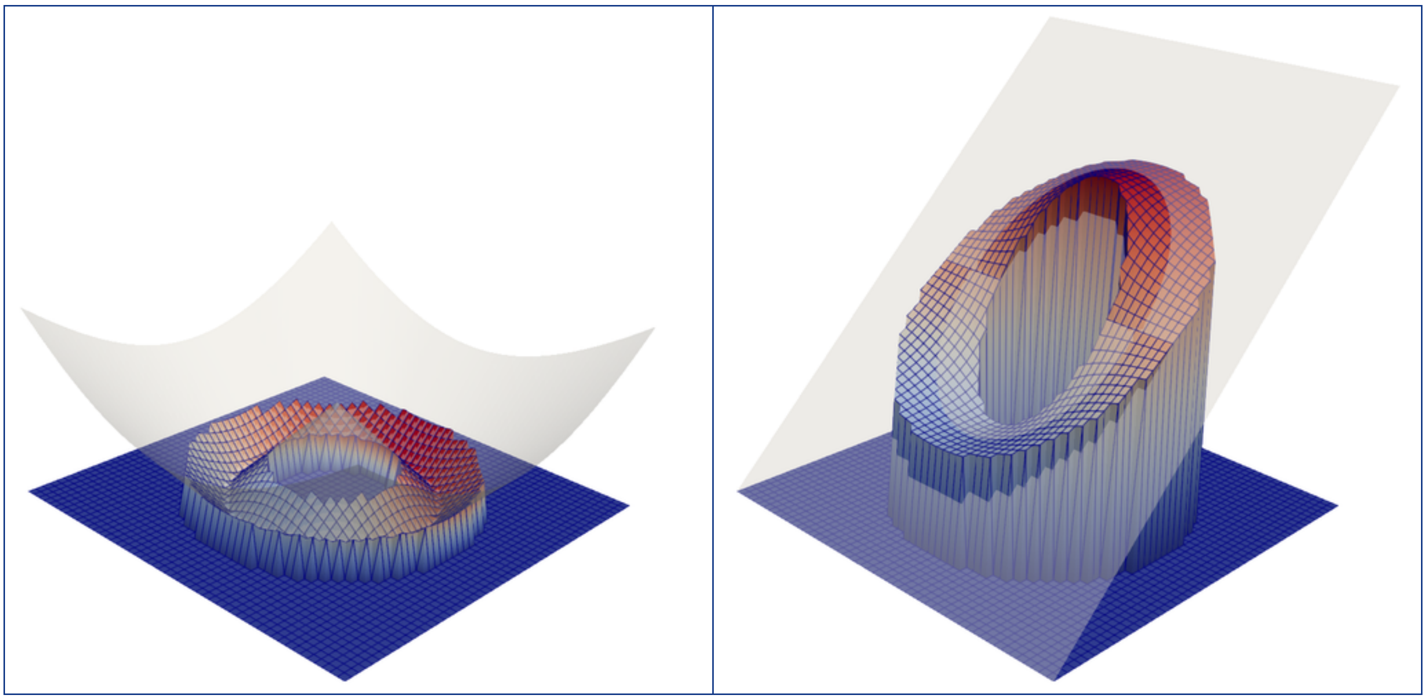
We show the screen output.

```
Running: example 2
  - create system
  - determine narrow band
  - determine closest point iteratively
    - iteration 0: 7076 -> 7076
    - iteration 1: 7076 -> 104
    - iteration 2: 104 -> 0
  - determine distance in narrow band
  - perform extrapolation in narrow band
  - output results
```

The following three plots, representing the performed iterations of the closest-point projection, show the current position of the closest points exceeding the required tolerance of the discrete interface of the circle and still need to be corrected. It can be seen that the numbers of points to be processed decrease from iteration to iteration.



The output visualized in Paraview looks like the following: On the left, the original distance function is shown as the light gray surface. In addition, the contour values refer to the distance values determined from calculation of the distance to the closest points at the interface in the narrow band. It can be seen that the two functions coincide. Similarly, on the right, the original solution and the extrapolated solution from the interface is shown.

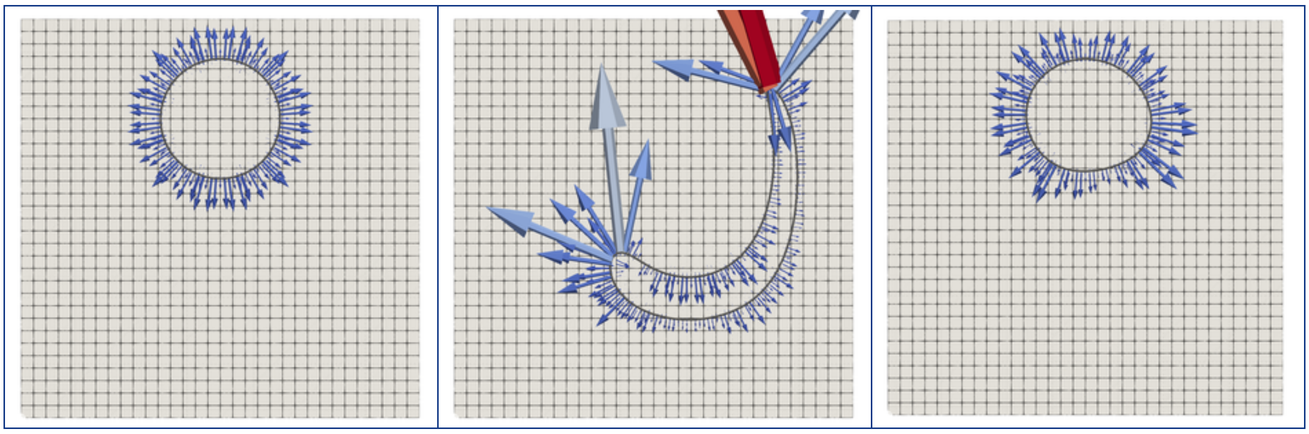


Mini example 3

We show a shortened version of the screen output.

```
Running: example 3
  - creating background mesh
  - creating immersed mesh
time: 0
  - compute to be tested values (immersed mesh)
  - test values (background mesh)
  - write data (background mesh)
  - write mesh (immersed mesh)

time: 0.01
  - move support points (immersed mesh)
  - compute to be tested values (immersed mesh)
  - test values (background mesh)

time: 0.02
  - move support points (immersed mesh)
  - compute to be tested values (immersed mesh)
  - test values (background mesh)

...

time: 2
  - move support points (immersed mesh)
  - compute to be tested values (immersed mesh)
  - test values (background mesh)
  - write data (background mesh)
  - write mesh (immersed mesh)
```

The output visualized in Paraview looks like the following: The deformation of the immersed mesh by the reversible vortex flow can be seen. Due to discretization errors, the circular shape is not exactly reobtained at the end, illustrated in the right figure. The sharp nature of the surface-tension force vector, shown as vector plots, can be seen by its restriction to cells that are intersected by the immersed mesh.



Possibilities for extension

This program highlights some of the main capabilities of the distributed evaluation routines in deal.II. However, there are many related topics worth mentioning:

- Performing a distributed search is an expensive step. That is why we suggest to provide hints to **Utilities::MPI::RemotePointEvaluation** and to reuse **Utilities::MPI::RemotePointEvaluation** instances in the case that the communication pattern has not changed. Furthermore, there are instances where no search is needed and the points are already sorted into the right cells. This is the case if the points are generated on the cell level (see **step-85**; CutFEM) or the points are automatically sorted into the correct (neighboring) cell (see **step-68**; PIC with **Particles::ParticleHandler**). Having said that, the **Particles::ParticleHandler** might use the described infrastructure to perform the initial sorting of particles into cells.
- We concentrated on parallelization aspects in this tutorial. However, we would like to point out the need for fast evaluation on cell level. The task for this in deal.II is **FEPointEvaluation**. It exploits the structure of

  $$\hat{u}(\hat{\boldsymbol{x}}) = \sum_i \hat{N}_i(\hat{\boldsymbol{x}})\hat{u}_i$$

  to derive fast implementations, e.g., for tensor-product elements

  $$\hat{u}(\hat{x}_0, \hat{x}_1, \hat{x}_2) = \sum_k \hat{N}_k^{\text{1D}}(\hat{x}_2) \sum_j \hat{N}_j^{\text{1D}}(\hat{x}_1) \sum_i \hat{N}_i^{\text{1D}}(\hat{x}_0)\hat{u}_{ijk}.$$

  Since only 1D shape functions need to be tabulated, this formulation is faster than the original one but still slower than as if the tensor-product structure of evaluation points would be exploited as in the case of **FEEvaluation**.
- **Utilities::MPI::RemotePointEvaluation** is used in multiple places in deal.II. The class **DataOutResample** allows to output results on a different mesh than the computational mesh. This is useful if one wants to output the results on a coarser mesh or one does not want to output 3D results but instead 2D slices. In

addition, MGTwoLevelTransferNonNested allows to prolongate solutions and restrict residuals between two independent meshes. By passing a sequence of such two-level transfer operators to MGTransferMF and, finally, to Multigrid, non-nested multigrid can be computed.

- Utilities::MPI::RemotePointEvaluation can be used to couple non-matching grids via surfaces (example: fluid-structure interaction, independently created grids). The evaluation points can come from any side (pointwise interpolation) or are defined on intersected meshes (Nitsche-type mortaring [101]). On how to create such intersected meshes and evaluation points on these, see GridTools::internal::distributed_compute_intersection_locations().
- Alternatively to the coupling via Utilities::MPI::RemotePointEvaluation, preCICE [45] [52] can be used. The code-gallery program "Laplace equation coupled to an external simulation program" shows how to use this library with deal.II.

The plain program

```
/* ---------------------------------------------------------------------
 *
 * Copyright (C) 2023 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.
 *
 * ---------------------------------------------------------------------
 *
 *
 * Authors: Magdalena Schreter-Fleischhacker, Technical University of
 *          Munich, 2023
 *          Peter Munch, University of Augsburg, 2023
 */

#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function_lib.h>
#include <deal.II/base/function_signed_distance.h>
#include <deal.II/base/mpi.h>
#include <deal.II/base/mpi.templates.h>

#include <deal.II/distributed/tria.h>

#include <deal.II/dofs/dof_renumbering.h>

#include <deal.II/fe/fe_nothing.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/fe/mapping_q_cache.h>

#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_tools.h>

#include <deal.II/lac/generic_linear_algebra.h>

#include <deal.II/numerics/data_out.h>

#include <iostream>

#include <deal.II/base/mpi_remote_point_evaluation.h>
#include <deal.II/numerics/vector_tools.h>

#include <deal.II/matrix_free/fe_point_evaluation.h>


namespace Step87
{
  using namespace dealii;

#ifdef DEAL_II_WITH_P4EST
  template <int dim, int spacedim = dim>
  using DistributedTriangulation = typename std::conditional<
    dim == 1,
    parallel::shared::Triangulation<dim, spacedim>,
    parallel::distributed::Triangulation<dim, spacedim>>::type;
#else
  template <int dim, int spacedim = dim>
  using DistributedTriangulation =
    parallel::shared::Triangulation<dim, spacedim>;
#endif

  template <int spacedim>
  std::vector<Point<spacedim>>
  create_points_along_line(const Point<spacedim> &p0,
                           const Point<spacedim> &p1,
                           const unsigned int     n_subdivisions)
  {
    Assert(n_subdivisions >= 1, ExcInternalError());

    std::vector<Point<spacedim>> points;
    points.reserve(n_subdivisions + 1);

    points.emplace_back(p0);
    for (unsigned int i = 1; i < n_subdivisions; ++i)
      points.emplace_back(p0 + (p1 - p0) * static_cast<double>(i) /
                                 static_cast<double>(n_subdivisions));
    points.emplace_back(p1);

    return points;
  }

  template <int spacedim, typename T0, typename T1 = int>
  void print_along_line(const std::string                  &file_name,
                        const std::vector<Point<spacedim>> &points,
                        const std::vector<T0>              &values_0,
                        const std::vector<T1>              &values_1 = {})
  {
    AssertThrow(points.size() == values_0.size() &&
                  (values_1.size() == points.size() || values_1.empty()),
                ExcMessage("The provided vectors must have the same length."));

    std::ofstream file(file_name);

    for (unsigned int i = 0; i < points.size(); ++i)
      {
        file << std::fixed << std::right << std::setw(5) << std::setprecision(3)
             << points[0].distance(points[i]);

        for (unsigned int d = 0; d < spacedim; ++d)
          file << std::fixed << std::right << std::setw(10)
               << std::setprecision(3) << points[i][d];

        file << std::fixed << std::right << std::setw(10)
             << std::setprecision(3) << values_0[i];

        if (!values_1.empty())
          file << std::fixed << std::right << std::setw(10)
               << std::setprecision(3) << values_1[i];
        file << std::endl;
      }
  }

  template <int dim, int spacedim>
  void collect_support_points(
    const Mapping<dim, spacedim>               &mapping,
    const DoFHandler<dim, spacedim>            &dof_handler,
    LinearAlgebra::distributed::Vector<double> &support_points)
  {
    support_points.reinit(dof_handler.locally_owned_dofs(),
                          DoFTools::extract_locally_active_dofs(dof_handler),
                          dof_handler.get_communicator());

    const auto &fe = dof_handler.get_fe();

    FEValues<dim, spacedim> fe_values(
      mapping,
      fe,
      fe.base_element(0).get_unit_support_points(),
      update_quadrature_points);

    std::vector<types::global_dof_index> local_dof_indices(
      fe.n_dofs_per_cell());

    for (const auto &cell : dof_handler.active_cell_iterators() |
                              IteratorFilters::LocallyOwnedCell())
      {
        fe_values.reinit(cell);
        cell->get_dof_indices(local_dof_indices);

        for (const auto q : fe_values.quadrature_point_indices())
          for (unsigned int c = 0; c < spacedim; ++c)
            support_points[local_dof_indices[fe.component_to_system_index(c,
                                                                          q)]] =
              fe_values.quadrature_point(q)[c];
      }
  }

  template <int dim, int spacedim, typename T>
  std::tuple<std::vector<Point<spacedim>>, std::vector<types::global_dof_index>>
  collect_support_points_with_narrow_band(
    const Mapping<dim, spacedim>                &mapping,
    const DoFHandler<dim, spacedim>             &dof_handler_signed_distance,
    const LinearAlgebra::distributed::Vector<T> &signed_distance,
    const DoFHandler<dim, spacedim>             &dof_handler_support_points,
    const double                                 narrow_band_threshold)
  {
    AssertThrow(narrow_band_threshold >= 0,
                ExcMessage("The narrow band threshold"
                           " must be larger than or equal to 0."));
    const auto &tria = dof_handler_signed_distance.get_triangulation();
    const Quadrature<dim> quad(dof_handler_support_points.get_fe()
                                 .base_element(0)
                                 .get_unit_support_points());

    FEValues<dim> distance_values(mapping,
                                  dof_handler_signed_distance.get_fe(),
                                  quad,
                                  update_values);

    FEValues<dim> req_values(mapping,
                             dof_handler_support_points.get_fe(),
                             quad,
                             update_quadrature_points);
    std::vector<T>                       temp_distance(quad.size());
    std::vector<types::global_dof_index> local_dof_indices(
      dof_handler_support_points.get_fe().n_dofs_per_cell());

    std::vector<Point<dim>>              support_points;
    std::vector<types::global_dof_index> support_points_idx;

    const bool has_ghost_elements = signed_distance.has_ghost_elements();

    const auto &locally_owned_dofs_req =
      dof_handler_support_points.locally_owned_dofs();
    std::vector<bool> flags(locally_owned_dofs_req.n_elements(), false);

    if (has_ghost_elements == false)
      signed_distance.update_ghost_values();

    for (const auto &cell :
         tria.active_cell_iterators() | IteratorFilters::LocallyOwnedCell())
      {
        const auto cell_distance =
          cell->as_dof_handler_iterator(dof_handler_signed_distance);
        distance_values.reinit(cell_distance);
        distance_values.get_function_values(signed_distance, temp_distance);

        const auto cell_req =
          cell->as_dof_handler_iterator(dof_handler_support_points);
        req_values.reinit(cell_req);
        cell_req->get_dof_indices(local_dof_indices);

        for (const auto q : req_values.quadrature_point_indices())
          if (std::abs(temp_distance[q]) < narrow_band_threshold)
            {
              const auto idx = local_dof_indices[q];

              if (locally_owned_dofs_req.is_element(idx) == false ||
                  flags[locally_owned_dofs_req.index_within_set(idx)])
                continue;

              flags[locally_owned_dofs_req.index_within_set(idx)] = true;

              support_points_idx.emplace_back(idx);
              support_points.emplace_back(req_values.quadrature_point(q));
            }
      }

    if (has_ghost_elements == false)
      signed_distance.zero_out_ghost_values();

    return {support_points, support_points_idx};
  }

  template <int spacedim>
  std::vector<Point<spacedim>> convert(
    const LinearAlgebra::distributed::Vector<double> &support_points_unrolled)
  {
    const unsigned int n_points =
      support_points_unrolled.locally_owned_size() / spacedim;

    std::vector<Point<spacedim>> points(n_points);

    for (unsigned int i = 0, c = 0; i < n_points; ++i)
      for (unsigned int d = 0; d < spacedim; ++d, ++c)
        points[i][d] = support_points_unrolled.local_element(c);

    return points;
  }

  void example_0()
  {
    std::cout << "Running: example 0" << std::endl;

    constexpr unsigned int dim       = 2;
    constexpr unsigned int fe_degree = 3;

    MappingQ1<dim>     mapping;
    Triangulation<dim> tria;
    GridGenerator::subdivided_hyper_cube(tria, 7);

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

    Vector<double> vector(dof_handler.n_dofs());
    VectorTools::interpolate(
      mapping,
      dof_handler,
      Functions::SignedDistance::Sphere<dim>(Point<dim>(0.5, 0.5), 0.25),
      vector);

    const auto points_line =
      create_points_along_line(Point<dim>(0.0, 0.5), Point<dim>(1.0, 0.5), 20);

    std::vector<double> values_line;
    values_line.reserve(points_line.size());

    for (const auto &p_real : points_line)
      {
        const auto [cell, p_unit] =
          GridTools::find_active_cell_around_point(mapping,
                                                   dof_handler,
                                                   p_real);

        {
          AssertThrow(cell != dof_handler.end(), ExcInternalError());
          std::cout << " - Found point with real coordinates: " << p_real
                    << std::endl;
          std::cout << "   - in cell with vertices:";
          for (const auto &v : cell->vertex_indices())
            std::cout << " (" << cell->vertex(v) << ")";
          std::cout << std::endl;
          std::cout << "   - with coordinates on the unit cell: (" << p_unit
                    << ")" << std::endl;
        }

        std::cout << " - Values at point:" << std::endl;

        {
          FEValues<dim> fe_values(mapping,
                                  fe,
                                  Quadrature<dim>(p_unit),
                                  update_values);
          fe_values.reinit(cell);

          std::vector<double> quad_values(1);
          fe_values.get_function_values(vector, quad_values);
          const double value_0 = quad_values[0];
          std::cout << "  - " << value_0 << " (w. FEValues)" << std::endl;
          values_line.push_back(value_0);
        }

        {
          std::vector<double> cell_vector(fe.n_dofs_per_cell());
          cell->get_dof_values(vector, cell_vector.begin(), cell_vector.end());

          FEPointEvaluation<1, dim> fe_point(mapping, fe, update_values);
          fe_point.reinit(cell, ArrayView<const Point<dim>>(p_unit));
          fe_point.evaluate(cell_vector, EvaluationFlags::values);
          const auto value_1 = fe_point.get_value(0);
          std::cout << "  - " << value_1 << " (w. FEPointEvaluation)"
                    << std::endl;
        }

        {
          const auto value_2 =
            VectorTools::point_value(dof_handler, vector, p_real);
          std::cout << "  - " << value_2 << " (w. VectorTools::point_value())"
                    << std::endl;
          std::cout << std::endl;
        }
      }

    std::cout << " - writing csv file" << std::endl;
    print_along_line("example_0_profile.csv", points_line, values_line);
  }

  void example_1()
  {
    constexpr unsigned int dim       = 2;
    constexpr unsigned int fe_degree = 3;

    AssertDimension(dim, 2);

    ConditionalOStream pcout(std::cout,
                             Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) ==
                               0);

    pcout << "Running: example 1" << std::endl;

    MappingQ1<dim>                mapping;
    DistributedTriangulation<dim> tria(MPI_COMM_WORLD);
    GridGenerator::subdivided_hyper_cube(tria, 7);

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

    LinearAlgebra::distributed::Vector<double> signed_distance;
    signed_distance.reinit(dof_handler.locally_owned_dofs(),
                           DoFTools::extract_locally_active_dofs(dof_handler),
                           MPI_COMM_WORLD);

    VectorTools::interpolate(
      mapping,
      dof_handler,
      Functions::SignedDistance::Sphere<dim>(Point<dim>(0.5, 0.5), 0.25),
      signed_distance);

    LinearAlgebra::distributed::Vector<double> solution;
    solution.reinit(dof_handler.locally_owned_dofs(),
                    DoFTools::extract_locally_active_dofs(dof_handler),
                    MPI_COMM_WORLD);

    VectorTools::interpolate(mapping,
                             dof_handler,
                             Functions::SignedDistance::Plane<dim>(
                               Point<dim>(), Point<dim>::unit_vector(0)),
                             solution);

    std::vector<Point<dim>> profile;
    if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
      profile = create_points_along_line(Point<dim>(0.0, 0.5),
                                         Point<dim>(1.0, 0.5),
                                         20);

    Utilities::MPI::RemotePointEvaluation<dim, dim> rpe;

    signed_distance.update_ghost_values();
    const std::vector<double> profile_signed_distance =
      VectorTools::point_values<1>(
        mapping, dof_handler, signed_distance, profile, rpe);
    signed_distance.zero_out_ghost_values();

    const auto evaluate_function = [&](const ArrayView<double> &values,
                                       const auto              &cell_data) {
      FEPointEvaluation<1, dim> fe_point(mapping, fe, update_values);

      std::vector<double>                  local_values;
      std::vector<types::global_dof_index> local_dof_indices;

      for (const auto cell : cell_data.cell_indices())
        {
          const auto cell_dofs =
            cell_data.get_active_cell_iterator(cell)->as_dof_handler_iterator(
              dof_handler);

          const auto unit_points = cell_data.get_unit_points(cell);
          const auto local_value = cell_data.get_data_view(cell, values);

          local_values.resize(cell_dofs->get_fe().n_dofs_per_cell());
          cell_dofs->get_dof_values(solution,
                                    local_values.begin(),
                                    local_values.end());

          fe_point.reinit(cell_dofs, unit_points);
          fe_point.evaluate(local_values, EvaluationFlags::values);

          for (unsigned int q = 0; q < unit_points.size(); ++q)
            local_value[q] = fe_point.get_value(q);
        }
    };

    std::vector<double> output;
    solution.update_ghost_values();
    rpe.evaluate_and_process<double>(output, evaluate_function);
    solution.zero_out_ghost_values();

    DataOut<dim> data_out;
    data_out.add_data_vector(dof_handler, signed_distance, "signed_distance");
    data_out.build_patches(mapping);
    data_out.write_vtu_in_parallel("example_1.vtu", MPI_COMM_WORLD);

    if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
      {
        std::cout << " - writing csv file" << std::endl;
        print_along_line("example_1_profile.csv",
                         profile,
                         profile_signed_distance,
                         output);
      }
  }

  void example_2()
  {
    constexpr unsigned int dim       = 2;
    constexpr unsigned int fe_degree = 3;

    ConditionalOStream pcout(std::cout,
                             Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) ==
                               0);

    pcout << "Running: example 2" << std::endl;
    pcout << "  - create system" << std::endl;

    FE_Q<dim>                      fe(fe_degree);
    MappingQ1<dim>                 mapping;
    DistributedTriangulation<dim> tria(MPI_COMM_WORLD);
    GridGenerator::subdivided_hyper_cube(tria, 50);

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

    LinearAlgebra::distributed::Vector<double> solution;
    solution.reinit(dof_handler.locally_owned_dofs(),
                    DoFTools::extract_locally_active_dofs(dof_handler),
                    MPI_COMM_WORLD);

    VectorTools::interpolate(mapping,
                             dof_handler,
                             Functions::SignedDistance::Plane<dim>(
                               Point<dim>(), Point<dim>::unit_vector(0)),
                           solution);

    LinearAlgebra::distributed::Vector<double> signed_distance;
    signed_distance.reinit(dof_handler.locally_owned_dofs(),
                           DoFTools::extract_locally_active_dofs(dof_handler),
                           MPI_COMM_WORLD);

    VectorTools::interpolate(
      mapping,
      dof_handler,
      Functions::SignedDistance::Sphere<dim>(Point<dim>(0.5, 0.5), 0.25),
      signed_distance);
    signed_distance.update_ghost_values();

    pcout << "  - determine narrow band" << std::endl;

    const auto [support_points, support_points_idx] =
      collect_support_points_with_narrow_band(mapping,
                                              dof_handler,
                                              signed_distance,
                                              dof_handler,
                                              0.1 /*narrow_band_threshold*/);

    pcout << "  - determine closest point iteratively" << std::endl;
    constexpr int    max_iter     = 30;
    constexpr double tol_distance = 1e-6;

    std::vector<Point<dim>> closest_points = support_points; // initial guess

    std::vector<unsigned int> unmatched_points_idx(closest_points.size());
    std::iota(unmatched_points_idx.begin(), unmatched_points_idx.end(), 0);

    int n_unmatched_points =
      Utilities::MPI::sum(unmatched_points_idx.size(), MPI_COMM_WORLD);

    Utilities::MPI::RemotePointEvaluation<dim, dim> rpe;

    for (int it = 0; it < max_iter && n_unmatched_points > 0; ++it)
      {
        pcout << "    - iteration " << it << ": " << n_unmatched_points;

        std::vector<Point<dim>> unmatched_points(unmatched_points_idx.size());
        for (unsigned int i = 0; i < unmatched_points_idx.size(); ++i)
          unmatched_points[i] = closest_points[unmatched_points_idx[i]];

        const auto all_unmatched_points =
          Utilities::MPI::reduce<std::vector<Point<dim>>>(
            unmatched_points, MPI_COMM_WORLD, [](const auto &a, const auto &b) {
              auto result = a;
              result.insert(result.end(), b.begin(), b.end());
              return result;
            });

        if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
          {
            std::ofstream file("example_2_" + std::to_string(it) + ".csv");
            for (const auto &p : all_unmatched_points)
              file << p << std::endl;
            file.close();
          }

        rpe.reinit(unmatched_points, tria, mapping);

        AssertThrow(rpe.all_points_found(),
                    ExcMessage("Processed point is outside domain."));

        const auto eval_values =
          VectorTools::point_values<1>(rpe, dof_handler, signed_distance);

        const auto eval_gradient =
          VectorTools::point_gradients<1>(rpe, dof_handler, signed_distance);

        std::vector<unsigned int> unmatched_points_idx_next;

        for (unsigned int i = 0; i < unmatched_points_idx.size(); ++i)
          if (std::abs(eval_values[i]) > tol_distance)
            {
              closest_points[unmatched_points_idx[i]] -=
                eval_values[i] * eval_gradient[i];

              unmatched_points_idx_next.emplace_back(unmatched_points_idx[i]);
            }

        unmatched_points_idx.swap(unmatched_points_idx_next);

        n_unmatched_points =
          Utilities::MPI::sum(unmatched_points_idx.size(), MPI_COMM_WORLD);

        pcout << " -> " << n_unmatched_points << std::endl;
      }

    if (n_unmatched_points > 0)
      pcout << "WARNING: The tolerance of " << n_unmatched_points
            << " points is not yet attained." << std::endl;

    pcout << "  - determine distance in narrow band" << std::endl;
    LinearAlgebra::distributed::Vector<double> solution_distance;
    solution_distance.reinit(solution);

    for (unsigned int i = 0; i < closest_points.size(); ++i)
      solution_distance[support_points_idx[i]] =
        support_points[i].distance(closest_points[i]);

    pcout << "  - perform extrapolation in narrow band" << std::endl;
    rpe.reinit(closest_points, tria, mapping);
    solution.update_ghost_values();
    const auto vals = VectorTools::point_values<1>(rpe, dof_handler, solution);
    solution.zero_out_ghost_values();

    LinearAlgebra::distributed::Vector<double> solution_extrapolated;
    solution_extrapolated.reinit(solution);

    for (unsigned int i = 0; i < closest_points.size(); ++i)
      solution_extrapolated[support_points_idx[i]] = vals[i];

    pcout << "  - output results" << std::endl;
    DataOut<dim> data_out;
    data_out.add_data_vector(dof_handler, signed_distance, "signed_distance");
    data_out.add_data_vector(dof_handler, solution, "solution");
    data_out.add_data_vector(dof_handler,
                             solution_distance,
                             "solution_distance");
    data_out.add_data_vector(dof_handler,
                             solution_extrapolated,
                             "solution_extrapolated");
    data_out.build_patches(mapping);
    data_out.write_vtu_in_parallel("example_2.vtu", MPI_COMM_WORLD);

    pcout << std::endl;
  }


  void example_3()
  {
    const unsigned int degree       = 3;
    const unsigned int dim          = 2;
    const double       dt           = 0.2;
    const unsigned int n_time_steps = 10;

    ConditionalOStream pcout(std::cout,
                             Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) ==
                               0);

    pcout << "Running: example 3" << std::endl;

    pcout << "  - creating background mesh" << std::endl;
    DistributedTriangulation<dim> tria_background(MPI_COMM_WORLD);
    GridGenerator::hyper_cube(tria_background);
    tria_background.refine_global(5);

    MappingQ1<dim>  mapping_background;
    FESystem<dim>   fe_background(FE_Q<dim>(degree), dim);
    DoFHandler<dim> dof_handler_background(tria_background);
    dof_handler_background.distribute_dofs(fe_background);

    pcout << "  - creating immersed mesh" << std::endl;
    const Point<dim> center(0.5, 0.75);
    const double     radius = 0.15;

    DistributedTriangulation<dim - 1, dim> tria_immersed(MPI_COMM_WORLD);
    GridGenerator::hyper_sphere(tria_immersed, center, radius);
    tria_immersed.refine_global(4);

    MappingQ<dim - 1, dim>      mapping_immersed_base(3);
    MappingQCache<dim - 1, dim> mapping_immersed(3);
    mapping_immersed.initialize(mapping_immersed_base, tria_immersed);
    QGauss<dim - 1> quadrature_immersed(degree + 1);

    FE_Q<dim - 1, dim>       fe_scalar_immersed(degree);
    FESystem<dim - 1, dim>   fe_immersed(fe_scalar_immersed, dim);
    DoFHandler<dim - 1, dim> dof_handler_immersed(tria_immersed);
    dof_handler_immersed.distribute_dofs(fe_immersed);

    DoFRenumbering::support_point_wise(dof_handler_immersed);

    LinearAlgebra::distributed::Vector<double> velocity;
    velocity.reinit(dof_handler_background.locally_owned_dofs(),
                    DoFTools::extract_locally_active_dofs(
                      dof_handler_background),
                    MPI_COMM_WORLD);
    Functions::RayleighKotheVortex<dim> vortex(2);

    LinearAlgebra::distributed::Vector<double> force_vector(
      dof_handler_background.locally_owned_dofs(),
      DoFTools::extract_locally_active_dofs(dof_handler_background),
      MPI_COMM_WORLD);

    LinearAlgebra::distributed::Vector<double> immersed_support_points;
    collect_support_points(mapping_immersed,
                           dof_handler_immersed,
                           immersed_support_points);

    Utilities::MPI::RemotePointEvaluation<dim> rpe;
    double                                     time = 0.0;
    for (unsigned int it = 0; it <= n_time_steps; ++it, time += dt)
      {
        pcout << "time: " << time << std::endl;
        if (it > 0)
          {
            pcout << "  - move support points (immersed mesh)" << std::endl;
            vortex.set_time(time);
            VectorTools::interpolate(mapping_background,
                                     dof_handler_background,
                                     vortex,
                                     velocity);
            rpe.reinit(convert<dim>(immersed_support_points),
                       tria_background,
                       mapping_background);

            AssertThrow(rpe.all_points_found(),
                        ExcMessage(
                          "Immersed domain leaves background domain!"));

            velocity.update_ghost_values();
            const auto immersed_velocity =
              VectorTools::point_values<dim>(rpe,
                                             dof_handler_background,
                                             velocity);
            velocity.zero_out_ghost_values();

            for (unsigned int i = 0, c = 0;
                 i < immersed_support_points.locally_owned_size() / dim;
                 ++i)
              for (unsigned int d = 0; d < dim; ++d, ++c)
                immersed_support_points.local_element(c) +=
                  immersed_velocity[i][d] * dt;

            mapping_immersed.initialize(mapping_immersed_base,
                                        dof_handler_immersed,
                                        immersed_support_points,
                                        false);
          }

        pcout << "  - compute to be tested values (immersed mesh)" << std::endl;
        using value_type = Tensor<1, dim, double>;

        std::vector<Point<dim>> integration_points;
        std::vector<value_type> integration_values;

        FEValues<dim - 1, dim> fe_values(mapping_immersed,
                                         fe_immersed,
                                         quadrature_immersed,
                                         update_JxW_values | update_gradients |
                                           update_normal_vectors |
                                           update_quadrature_points);

        FEValues<dim - 1, dim> fe_values_co(
          mapping_immersed,
          fe_scalar_immersed,
          fe_scalar_immersed.get_unit_support_points(),
          update_JxW_values | update_normal_vectors);

        for (const auto &cell : tria_immersed.active_cell_iterators() |
                                  IteratorFilters::LocallyOwnedCell())
          {
            fe_values.reinit(cell);
            fe_values_co.reinit(cell);

            for (const auto &q : fe_values.quadrature_point_indices())
              {
                integration_points.emplace_back(fe_values.quadrature_point(q));

                const auto sigma = 1.0; // surface tension coefficient

                const auto normal = fe_values.normal_vector(q);

                double curvature = 0;
                for (unsigned int i = 0;
                     i < fe_scalar_immersed.n_dofs_per_cell();
                     ++i)
                  for (unsigned int d = 0; d < dim; ++d)
                    curvature +=
                      fe_values.shape_grad_component(
                        fe_immersed.component_to_system_index(d, i), q, d)[d] *
                      fe_values_co.normal_vector(i)[d];

                const auto FxJxW =
                  sigma * curvature * normal * fe_values.JxW(q);

                integration_values.emplace_back(FxJxW);
              }
          }

        pcout << "  - test values (background mesh)" << std::endl;

        rpe.reinit(integration_points, tria_background, mapping_background);

        const auto integration_function = [&](const auto &values,
                                              const auto &cell_data) {
          FEPointEvaluation<dim, dim> phi_force(mapping_background,
                                                fe_background,
                                                update_values);

          std::vector<double>                  local_values;
          std::vector<types::global_dof_index> local_dof_indices;

          for (const auto cell : cell_data.cell_indices())
            {
              const auto cell_dofs =
                cell_data.get_active_cell_iterator(cell)
                  ->as_dof_handler_iterator(dof_handler_background);

              const auto unit_points = cell_data.get_unit_points(cell);
              const auto FxJxW       = cell_data.get_data_view(cell, values);

              phi_force.reinit(cell_dofs, unit_points);

              for (const auto q : phi_force.quadrature_point_indices())
                phi_force.submit_value(FxJxW[q], q);

              local_values.resize(cell_dofs->get_fe().n_dofs_per_cell());
              phi_force.test_and_sum(local_values, EvaluationFlags::values);

              local_dof_indices.resize(cell_dofs->get_fe().n_dofs_per_cell());
              cell_dofs->get_dof_indices(local_dof_indices);
              AffineConstraints<double>().distribute_local_to_global(
                local_values, local_dof_indices, force_vector);
            }
        };

        rpe.process_and_evaluate<value_type>(integration_values,
                                             integration_function);
        force_vector.compress(VectorOperation::add);

        if (it % 10 == 0 || it == n_time_steps)
          {
            std::vector<
              DataComponentInterpretation::DataComponentInterpretation>
              vector_component_interpretation(
                dim, DataComponentInterpretation::component_is_part_of_vector);
            pcout << "  - write data (background mesh)" << std::endl;
            DataOut<dim>          data_out_background;
            DataOutBase::VtkFlags flags_backround;
            flags_backround.write_higher_order_cells = true;
            data_out_background.set_flags(flags_backround);
            data_out_background.add_data_vector(
              dof_handler_background,
              force_vector,
              std::vector<std::string>(dim, "force"),
              vector_component_interpretation);
            data_out_background.add_data_vector(
              dof_handler_background,
              velocity,
              std::vector<std::string>(dim, "velocity"),
              vector_component_interpretation);
            data_out_background.build_patches(mapping_background, 3);
            data_out_background.write_vtu_in_parallel("example_3_background_" +
                                                        std::to_string(it) +
                                                        ".vtu",
                                                      MPI_COMM_WORLD);

            pcout << "  - write mesh (immersed mesh)" << std::endl;
            DataOut<dim - 1, dim> data_out_immersed;
            data_out_immersed.attach_triangulation(tria_immersed);
            data_out_immersed.build_patches(mapping_immersed, 3);
            data_out_immersed.write_vtu_in_parallel("example_3_immersed_" +
                                                      std::to_string(it) +
                                                      ".vtu",
                                                    MPI_COMM_WORLD);
          }
        pcout << std::endl;
      }
  }
} // namespace Step87

int main(int argc, char **argv)
{
  using namespace dealii;
  Utilities::MPI::MPI_InitFinalize mpi(argc, argv, 1);
  std::cout.precision(5);

  if (Utilities::MPI::this_mpi_process(MPI_COMM_WORLD) == 0)
    Step87::example_0(); // only run on root process

  Step87::example_1();
  Step87::example_2();
  Step87::example_3();
}
```

Generated by doxygen 1.9.7