Dune-MMesh: The DUNE Grid Module for Moving Interfaces

Samuel Burbulla, Andreas Dedner, Maximilian Hörl, Christian Rohde

Apr 13, 2023

Dune-MMesh is an implementation of the DUNE [BBD+21] grid interface that is tailored for numerical applications with moving physical interfaces. The implementation based on CGAL [TCP20] triangulations supports two and three dimensional meshes and can export a predefined set of facets as a separate interface grid. In spatial dimension two, arbitrary movement of vertices is enhanced with a remeshing algorithm that implements non-hierarchical adaptation procedures. We present a collection of *Examples* based on the python bindings of the discretization module dune-fem [DNK+20].

1 Introduction

In several physical and environmental processes that concern multiphase flows, biological systems, and geophysical phenomena, important physical processes occur along thin physical interfaces. These processes include effects that may alter the interface's position or topology over time creating a moving interface, which complicates traditional modelling techniques. Moving interface problems thus require advanced numerical tools with specific treatment of the interface and the simultaneous ability to implement complex physical effects.

Dune-MMesh is tailored for numerical applications with moving physical interfaces. It is an implementation of the well-developed Dune grid interface and is well-suited for the numerical discretization of partial differential equations. The package wraps two and three dimensional CGAL triangulations in high-level objects like intersections of grid entities, index and id sets and geometry transformations and exports a predefined set of facets as a separate interface grid. In two dimensions, the arbitrary movement of vertices is enhanced with a re-meshing algorithm that implements non-hierarchical adaptation procedures. Besides the adaptation of the triangulation, Dune-MMesh provides the necessary data structures to adapt discrete functions defined on the bulk grid or the interface. This adaptation approach complements existing grid implementations within the Dune framework that strictly rely on hierarchical adaptation. Various examples in Python have been implemented based on the discretization module Dune-Fem that demonstrate the versatile applicability of Dune-MMesh. Due to the ability to handle custom PDEs in their weak from written in Unified Form Language (UFL) and the mesh adaptation capabilities, we believe Dune-MMesh provides a useful tool for solving mixed-dimensional PDEs on moving interfaces that arise from various fields of modelling.

More details about the concepts behind Dune-MMesh are described in *Concepts*. The procedure to install and use Dune-MMesh is specified in *Installation*. You can find a collection of examples of what can be done with Dune-MMesh based on the discretization module Dune-Fem in *Examples*. The programming interface is described in the sections *Python* and *C++*.

2 Installation

To install and use Dune-MMesh on your system you need:

C++ compiler (at least C++17 compatible, e.g., clang >= 5 or g++ >= 7), CMake (3.13 or later), Python3 (3.7 or later) + pip (+ venv), pkg-config, Boost (1.66 or later), OpenMPI (optional), SuiteSparse (we use UMFPack) and Gmsh.

On Linux the requirements could be installed as follows:

```
apt install g++ cmake python3 python3-pip python3-venv pkg-config libboost-dev␣
↪libopenmpi-dev openmpi-bin libsuitesparse-dev gmsh git git-lfs
```

On MacOS, you can install the required dependencies by installing the Xcode Command Line Tools and using Homebrew:

```
xcode-select --install
brew install pkg-config boost openmpi suite-sparse gmsh git-lfs
```

There are two ways to install Dune-MMesh, either from PyPI or from source.

2.1 Using Pip

The easiest way to install Dune-MMesh is using pip and the package uploaded to PyPI.

1. Activate a virtual environment (strongly recommended).

```
python3 -m venv dune-env
source dune-env/bin/activate
```

This requires that you have *venv* available (*apt install python3-venv*).

2. Download and build Dune-MMesh and its dependencies.

```
pip install dune-mmesh
```

Note that this takes some time to compile all dependent Dune modules.

Now, you should be able to execute Dune-MMesh's python code. For instance:

```
git clone https://github.com/samuelburbulla/dune-mmesh.git
cd dune-mmesh/doc/examples
python coupling.py
```

Remark that a *dune-py* module will be generated automatically that is necessary to perform the just-in-time compilation of DUNE python modules.

Please be aware that we use *git-lfs* for uploading the *.msh* files. To pull them, please activate large file storage.

2.2 From Source

You can install Dune-MMesh from source to get full access to the source code. It also enables git support if you want to contribute.

1. Clone the Dune modules dune-common, dune-geometry, dune-grid, dune-istl, dune-localfunctions, dune-alugrid dune-fem and dune-mmesh.

```
git clone https://gitlab.dune-project.org/core/dune-common.git
git clone https://gitlab.dune-project.org/core/dune-geometry.git
git clone https://gitlab.dune-project.org/core/dune-grid.git
git clone https://gitlab.dune-project.org/core/dune-istl.git
git clone https://gitlab.dune-project.org/core/dune-localfunctions.git
git clone https://gitlab.dune-project.org/extensions/dune-alugrid.git
git clone https://gitlab.dune-project.org/dune-fem/dune-fem.git
git clone https://gitlab.dune-project.org/samuel.burbulla/dune-mmesh.git
```

2. Build the modules. This will create an internal virtual environment and install the python bindings.

```
./dune-common/bin/dunecontrol --opts=dune-mmesh/cmake/config.opts all
```

3. Activate the DUNE internal virtual environment.

```
source ./dune-common/build-cmake/dune-env/bin/activate
```

2.3 Using Docker

The easiest starting point is to use Docker with a preconfigured setup.

Using the pre-built Docker container you can simply run:

```
docker run -it ghcr.io/samuelburbulla/dune-mmesh:master
```

Alternatively, you can build the corresponding Docker container yourself:

```
docker build -t mmesh \
  https://raw.githubusercontent.com/samuelburbulla/dune-mmesh/master/scripts/Dockerfile
docker run -it mmesh
```

This will open an interactive shell in the Dune-MMesh's examples directory.

3 Concepts

There are a few concepts behind the implementation of Dune-MMesh that we shall describe in more detail. The concepts can be split into three main parts: *CGAL Wrapper*, *Interface Grid* and *Moving Mesh*.

3.1 CGAL Wrapper

In its core, Dune-MMesh is a wrapper of CGAL Triangulations in $\mathbb{R}^d$, $d = 2, 3$, that implements the Dune grid interface. Therefore, it is essential to understand how CGAL triangulation objects are translated into Dune entities.

First of all, a CGAL triangulation is a set of simplicial cells and vertices. Each cell gives access to its $d + 1$ incident vertices and its $d+1$ adjacent cells. Each vertex gives access to one of its incident cells. The $d+1$ vertices are indexed with $0, 1, \dots, d$ in positive orientation being defined by the orientation of the underlying Euclidian space $\mathbb{R}^d$. The neighbors of a cell are also indexed with $0, 1, \dots, d$ in such a way that the neighbor is opposite to the vertex with the same index. Facets are not explicitly represented: a facet is given by the pair of a cell c and an index i. Here, the facet i of cell c is the facet of c that is opposite to the vertex with index i. Remark that a facet has two implicit representations. For $d = 3$, edges are represented by triples of a cell c and two indices i and j that indicate the two vertices of the edge. In order to match the Dune grid interface we have to follow the reference element numbering. Fortunately,

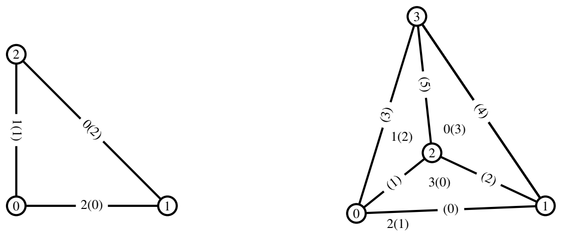


Fig. 1: CGAL representation of cells and differing Dune numbering in brackets.

the vertex numbering of cells can be retained. However, each facet i of the CGAL representation corresponds to the codim-1 subentity $d-i$ in the Dune reference element. For the representation of Dune intersections we can directly use CGAL's cell-index representation of facets which is already equipped with an orientation. With this reference mapping all geometry and sub-entity objects of the Dune grid interface can be specified.

Various iterators of CGAL triangulations can directly be used to construct the Dune grid range generators. For instance, the element iterator coincides with the `finite_faces_iterator` or `finite_cells_iterator`. Additional (non-standard Dune) iterators could be added easily, e.g. `incidentElements` or `incidentVertices` of a vertex.

The main large objects that have to be implemented are the index and id sets. For this purpose, we define ids of entities as follows. At creation, each vertex is equipped with a unique integer id. Each higher dimensional entity's id is defined by the sorted tuple of corresponding vertex ids.

As CGAL vertices and cells allow to append data (called: *info*) to the objects, we can store and access the vertex ids directly within the vertex objects. Entity indices are consecutively distributed at grid creation (or after adaptation) and also can be stored in the corresponding cell or vertex info. For entities of codimensions different than 0 and d, an id-index mapping is used.

The geometrical representation of entities that are not intrinsically CGAL entities (i.e., codimensions $1, ..., d-1$) is made unique by an ascending order of vertex ids. In additon, this prevents twists of intersections and we obtain a twist free grid implementation.

We extend the above described concepts of wrapping the CGAL triangulation to export a set of facets as *Interface Grid*.

3.2 Interface Grid

Consider a domain $\Omega \subset \mathbb{R}^d, d \in \{2,3\}$, that includes a $(d-1)$-dimensional interface $\Gamma \subset \Omega$. We assume the domain is triangulated conforming to the interface Γ. Let us denote this triangulation by $\mathcal{T}$ and the set of facets by $\mathcal{F}$. Due

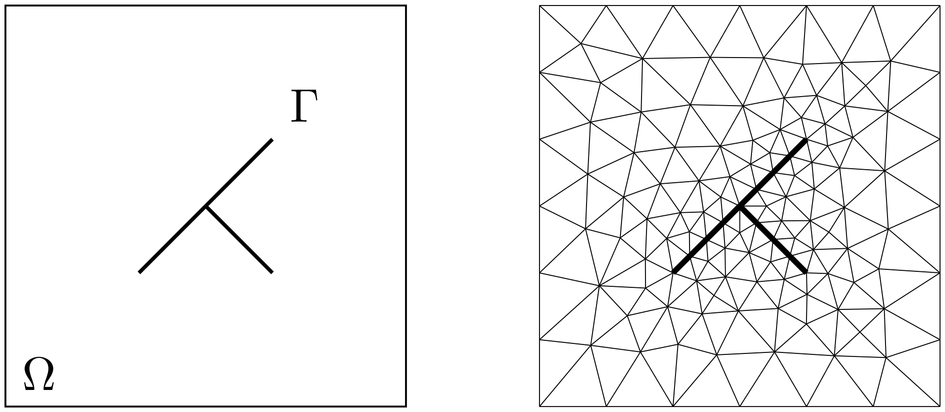


Fig. 2: A domain with a T-shaped interface and an example for a conforming triangulation.

to conforming meshing, there exists a subset of facets $\mathcal{F}_\Gamma \subset \mathcal{F}$ that belong to the interface Γ. Therefore, these facets in $\mathcal{F}_\Gamma$ can also be interpreted as a triangulation of a surface. We call this surface triangulation the *interface grid* and denote it by $\mathcal{T}_\Gamma$.

Dune-MMesh features a second implementation of the Dune grid interface that represents the interface triangulation $\mathcal{T}_\Gamma$. Therefore, facets have to be marked as belonging to the interface - usually this is done when parsing a .msh file.

The interface grid can be used like any other Dune grid, as it implements all necessary functionality.

A codim-0 entity of the interface grid is represented by a CGAL cell-index pair, as used for the codim-1 entities of the wrapper implementation. This representation is made unique by taking the representation where the cell has the lower index - which is also considered to be the positive side of the facet.

All subentity objects can be generated by this representation using the right indexing of vertices. The geometry representations and element ids are made unique by ascending order of vertex ids, as it is done in the full-dimensional wrapper implementation.

For iteration, CGAL's `finite_edges_iterator` or `finite_facets_iterator` is used, skipping all facets not belonging to the interface. Intersections and neighbor relationships are obtained by CGAL's `incident_edges` or `incident_facets` iterators. Index sets are implemented by mappings of vertex ids.

The interface grid also supports networks. For this purpose, the intersection iterator returns all common intersections with adjacent cells. Note that this can be more than one for a single codim-1 subentity. However, the intersection's outer normal is always independent of the neighbor entity. Each bulk grid intersection can be identified belonging to

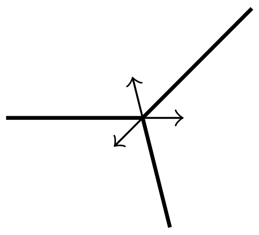


Fig. 3: Outer normals at junctions.

the interface or not. It is also possible to convert bulk intersections to interface grid elements and vice versa, as the underlying representation is the same. When converting an interface grid entity to a bulk intersection, Dune-MMesh returns the intersection as seen from the cell with the lower index.

3.3 Moving Mesh

Most interface driven-problems have time-dependent interfaces $\Gamma = \Gamma(t)$. Therefore, Dune-MMesh features capabilities of moving and re-meshing in spatial dimension two.

Note: The re-meshing feature is not (yet) supported in spatial dimension three because the removal of a vertex is not offered by the underlying CGAL Triangulation_3 class. In fact, it could appear that the region formed by its adjacent tetrahedrons is an instance of the untetrahedralizable Schönhardt's polyhedron. In this case, the removal of the vertex might be impossible without rebuilding the whole triangulation.

3.3.1 Moving Vertices

Dune-MMesh allows the movement of interface vertices (or all grid vertices) by a pre-described movement.

For this, we assume that movement is given by the shift of vertices. This movement can be performed by simply changing the coordinates of the vertices. Dune-MMesh provides the method `moveInterface(shifts)` that takes a vector of shift coordinates indexed by interface vertex indices. A second method `moveVertices(shifts)` is available

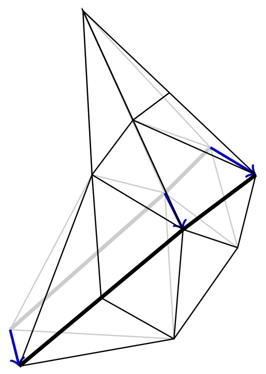


Fig. 4: Moving the interface.

for moving all vertices of the triangulation that is indexed by bulk vertex indices.

Remark that moving vertices might lead to degeneration of the triangulation, i.e., cells can have non-positive volume. To prevent that, Dune-MMesh is equipped with re-meshing routines we describe in the following.

3.3.2 Adaptation

Adaption in DUNE is hierarchical by definition. Whenever a grid element is supposed to be refined, it is split into smaller cells belonging to a higher level of the grid hierarchy. If all children in the highest refinement level of a grid element are supposed to be coarsened, the children cells are put together to form a parent cell one level lower.

Hereby, the adaptation procedure is performed in two stages:

1. Mark: Grid elements are marked for coarsening or refinement.
2. Adapt: The elements are adapted due to their markers, and discrete functions are restricted or prolongated.

In Dune-MMesh, due to the moving mesh, non-hierarchic adaptation is in-avoidable. However, we will try to follow the general DUNE approach of adaptation as good as possible. For this reason, we similarly separate the adaptation into two stages.

1. Mark

In advance of moving, two methods are provided for marking elements in a convenient way. First, the method

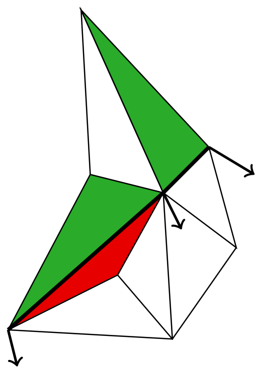


Fig. 5: Marking elements. Green: refine. Red: coarsen.

`ensureInterfaceMovement(shifts)` (respectively `ensureVertexMovement(shifts)`) can be called to prepare Dune-MMesh for moving the vertices. The routine takes the vertex shifts as argument and marks pre-sumbly degenerate cells for coarsening. Hence, they will be somehow removed during adaptation.

The second method available for marking elements is `markElements()`. This method uses a default indicator that marks elements depending on their current geometrical properties.

This indicator considers primarily maximal and minimal edge length and aims at an objective edge length between h_{max} and h_{min}.

- If an edge is longer than the maximum edge length h_{max}, the cell will be marked for refine.
- If an edge is shorter than the minimum edge length h_{min}, the cell will be marked for coarsening.

Additionally, if the ratio of longest to shortest edge is larger than 4, the cell is marked for coarsening. The number 4 occurs from the fact that we will use bisection and a triangle where two edges are longer then h_{max} should not be split into smaller triangles where an edge is shorter than h_{min}.

Finally, a maximal radius ratio is taken into account to remove very ugly cells. Always coarsening has priority before refinement because refinement would not remove ugly cells.

The minimal and maximal edge lengths h_{max} and h_{min} are initialized automatically when constructing a mesh by determining the range of edge lengths occurring in the grid.

Remark that `markElements()` also checks the elements of the interface grid. Therefore, the interface will be refined and coarsened as well if edges of the interface get too long or too short.

Note: The methods `ensureInterfaceMovement(shifts)` and `markElements()` are just convenience methods. Instead, one can also use a proprietary procedure marking elements manually, or one can insert and remove vertices directly using `removeVertex(vertex)` and `refineEdge(element, edgeIndex)`.

2. Adapt

After marking elements, the `adapt()` routine performs the actual adaptation process. The adaptation is performed by insertion and removal of points.

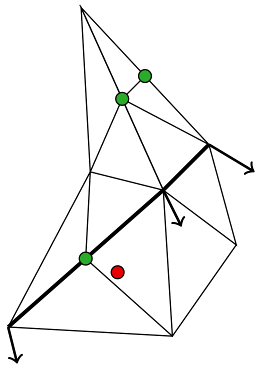


Fig. 6: Inserting and removing points.

- In each element that is marked for refinement, the center of the longest edge is inserted, i.e., refinement is done via bisection.
- In all elements marked for coarsening, one vertex is removed. Here, the vertex incident to the shortest edges of the cell is chosen, but we give priority to non-interface and non-boundary vertices.

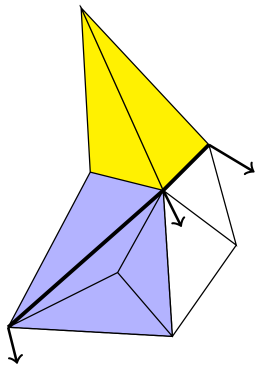


Fig. 7: Connected components.

When a vertex is removed, the resulting star-shaped hole is re-triangulated with respect to the interface. Here, for the purpose of projection, we introduce *connected components*. These are the minimal sets of cells from the triangulation before adaptation that cover the same area as a set of cells in the triangulation afterwards. The easiest representatives of these connected components are the incident cells when bisecting an edge and the incident cells to a vertex that is removed. Though, we have to combine overlapping sets of these representatives. For a conservative projection of discrete functions we compute a cut-set triangulation which enables evaluation with agglomerated quadrature rules on triangles. Here, we prolong from an old cell onto such a cut triangle and prolong onto the new cell. This whole projection is performed under the hood and just assumes that you use the callback adaptation in dune-fem. We use a similar concept on the interface grid that enables projection of discrete functions on the interface.

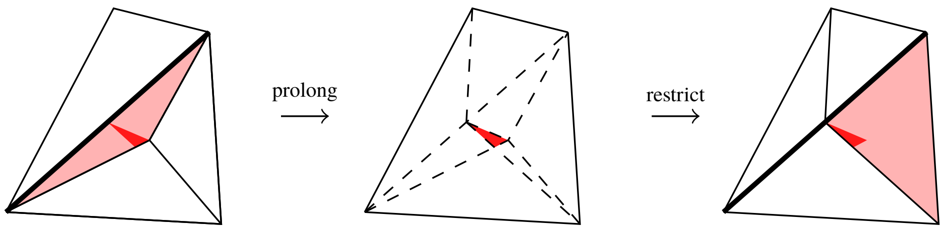


Fig. 8: Non-hierarchic projection with cut-set triangulation.

4 Examples

We implemented a few examples to display how Dune-MMesh can be used in different contexts.

4.1 Coupling to the interface

This is an example of how to use Dune-MMesh and solve coupled problems on the bulk and interface grid.

4.1.1 Grid creation

We use the *horizontal* grid file that contains an interface $\Gamma = [0.25, 0.75] \times 0.5$ embedded in a domain $\Omega = [0, 1]^2$. Grid creation from a mesh file works as follow.

[1]:
```
from dune.grid import reader
from dune.mmesh import mmesh
dim = 2
file = "grids/horizontal.msh"
gridView  = mmesh((reader.gmsh, file), dim)
igridView = gridView.hierarchicalGrid.interfaceGrid
```

4.1.2 Solve a problem on the bulk grid

Let us solve the Poisson equation

$$-\Delta u = f \qquad \text{in } \Omega \tag{1}$$

on the bulk grid. We use the manufactured solution $\hat{u} = \sin(4\pi xy)$ and therefore apply the source term $f = -\operatorname{div}(\nabla \hat{u})$. The weak form of the problem above reads

$$\int_\Omega \nabla u \cdot \nabla v \; dx = \int_\Omega fv \; dx \tag{2}$$

for all corresponding test functions v. This can be implemented as follows.

[2]:
```
from ufl import *
from dune.ufl import DirichletBC
from dune.fem.space import lagrange
from dune.fem.scheme import galerkin
from dune.fem.function import integrate
```

(continues on next page)

(continued from previous page)

```
space = lagrange(gridView, order=3)
u = TrialFunction(space)
v = TestFunction(space)

x = SpatialCoordinate(space)
exact = sin(x[0]*x[1]*4*pi)
f = -div(grad(exact))

a = inner(grad(u), grad(v)) * dx
b = f * v * dx

scheme = galerkin([a == b, DirichletBC(space, exact)], solver=("suitesparse", "umfpack"))
uh = space.interpolate(0, name="solution")
scheme.solve(target=uh)

def L2(u1, u2):
    return sqrt(integrate(u1.grid, (u1-u2)**2, order=5))

L2(uh, exact)
```

```
[2]: 5.628259763933402e-07
```

4.1.3 Solve a problem on the interface

We can solve similar problem on the interface Γ like

$$-\Delta u_\Gamma = f \qquad \text{in } \Gamma \tag{3}$$

with the weak form

$$\int_\Gamma \nabla u_\Gamma \cdot \nabla v_\Gamma \, dx = \int_\Gamma f v_\Gamma \, dx \tag{4}$$

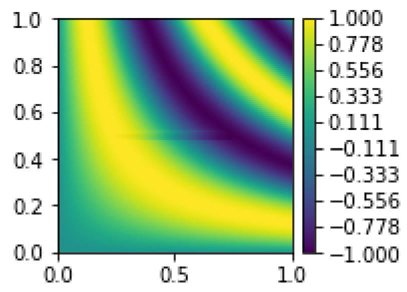
for all corresponding test functions v_Γ.

```
[3]: ispace = lagrange(igridView, order=3)
     iuh = ispace.interpolate(0, name="isolution")

     iu = TrialFunction(ispace)
     iv = TestFunction(ispace)

     ix = SpatialCoordinate(ispace)
     iexact = sin(0.5*ix[dim-2]*4*pi)
     iF = -div(grad(iexact))

     ia = inner(grad(iu), grad(iv)) * dx
     ib = iF * iv * dx

     ischeme = galerkin([ia == ib, DirichletBC(ispace, iexact)])
     ischeme.solve(target=iuh)
     L2(iuh, iexact)
```

```
[3]: 5.807742030532398e-08
```

We can use the `plotPointData` function to visualize the solution of both grids.

```
[4]: import matplotlib.pyplot as plt
     from dune.fem.plotting import plotPointData as plot

     figure = plt.figure(figsize=(3,3))
     plot(uh, figure=figure, gridLines=None)
     plot(iuh, figure=figure, linewidth=0.04, colorbar=None)
```



4.1.4 Couple bulk to surface

Dune-MMesh makes it possible to compute traces of discrete functions on Ω along Γ.

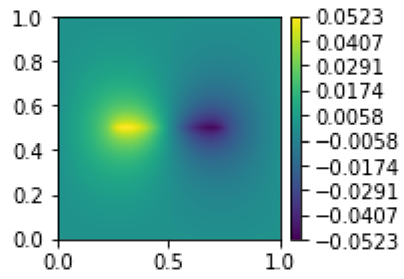
```
[5]: from dune.mmesh import trace
     tr = avg(trace(uh))
     ib = inner(grad(tr), grad(iv)) * dx

     iuh.interpolate(0)
     ischeme = galerkin([ia == ib, DirichletBC(ispace, avg(trace(uh)))])
     ischeme.solve(target=iuh)
     L2(iuh, iexact)
```

```
[5]: 4.266679479547976e-08
```

4.1.5 Couple surface to bulk

Similarly, we can evaluate a discrete function on Γ at the skeleton of the triangulation of Ω.

```
[6]: from dune.mmesh import skeleton

     sk = skeleton(iuh)
     b = avg(sk) * avg(v) * dS

     uh.interpolate(0)
     scheme = galerkin([a == b, DirichletBC(space, 0)])
     scheme.solve(target=uh)

     figure = plt.figure(figsize=(3,3))
     plot(uh, figure=figure, gridLines=None)
```



4.1.6 Compute jump of gradient of traces

We can use trace and skeleton within UFL expressions. This is useful when using jump and average operators, but also to compute gradients and more. Together with the utility function `normals`, which returns the positive normal of the underlying bulk facet, we can determine the orientation of the restricted values.

[7]:
```
from dune.mmesh import normals
inormal = normals(igridView)
jmp = jump(grad(trace(uh)), inormal)
```

4.2 Moving and adapting

This is an example of how to move the interface and adapt the mesh.

We implement the finite volume moving mesh method presented in [CMR+18].

4.2.1 Grid creation

We use the *vertical* grid file that contains an interface $\Gamma = 0.5 \times [0, 1]$ embedded in a domain $\Omega = [0, 1]^2$. For this example, we have to construct an *adaptive* leaf grid view and we will need to obtain the hierarchical grid object.

[1]:
```
from dune.grid import reader
from dune.mmesh import mmesh
from dune.fem.view import adaptiveLeafGridView as adaptive
dim = 2
file = "grids/vertical.msh"
gridView = adaptive( mmesh((reader.gmsh, file), dim) )
hgrid = gridView.hierarchicalGrid
igridView = hgrid.interfaceGrid
```

4.2.2 Problem

Let us consider the following transport problem.

$$u_t + \operatorname{div} f(u) = 0, \qquad \text{in } \Omega \times [0,T], \tag{5}$$

$$u(\cdot, 0) = u_0, \qquad \text{in } \Omega \tag{6}$$

where

$$f(u) = [1,0]^T u, \tag{7}$$

$$u_0(x,y) = (0.5 + x)\chi_{x<0.5}. \tag{8}$$

Further, the interface is supposed to move with the transport speed in f, i.e. $m = [1,0]^T$.

[2]:
```
from ufl import *
from dune.ufl import Constant

tEnd = 0.4

def speed():
    return as_vector([1.0, 0.0])

def movement(x):
    return as_vector([1.0, 0.0])

def f(u):
    return speed() * u

def u0(x):
    return conditional(x[0] < 0.5, 0.5+x[0], 0.0)

def uexact(x, t):
    return u0( x - t * speed() )
```

4.2.3 Finite Volume Moving Mesh Method

We use a Finite Volume Moving Mesh method to keep the discontinuity sharp. It can be formulated by

$$\int_\Omega (u^{n+1}|det(\Psi)| - u^n) v \, dx + \Delta t \int_{\mathcal{F}} \big(g(u^n, n) - h(u^n, n)\big)[v] \, dS = 0 \tag{9}$$

where $\Psi := x + \Delta t s$ and s is a linear interpolation of the interface's vertex movement m on the bulk triangulation.

The numerical fluxes $g(u,n)$ and $h(u,n)$ are assumed to be consistent with the flux functions $f(u) \cdot n$ and $us \cdot n$, respectively.

[3]:
```
from dune.fem.space import finiteVolume

space = finiteVolume(gridView)
u = TrialFunction(space)
v = TestFunction(space)
```

(continues on next page)

(continued from previous page)

```
x = SpatialCoordinate(space)
n = FacetNormal(space)

uh = space.interpolate(u0(x), name="uh")
uh_old = uh.copy()
```

[4]:
```
import numpy as np
from dune.geometry import vertex
from dune.mmesh import edgeMovement

def getShifts():
    mapper = igridView.mapper({vertex: 1})
    shifts = np.zeros((mapper.size, dim))
    for v in igridView.vertices:
        shifts[ mapper.index(v) ] = as_vector(movement( v.geometry.center ))
    return shifts

em = edgeMovement(gridView, getShifts())
t = Constant(0, name="time")

def g(u, n):
    sgn = inner(speed(), n('+'))
    return inner( conditional( sgn > 0, f( u('+') ), f( u('-') ) ), n('+') )

def gBnd(u, n):
    sgn = inner(speed(), n)
    return inner( conditional( sgn > 0, f(u), f(uexact(x, t)) ), n )

def h(u, n):
    sgn = inner(em('+'), n('+'))
    return conditional( sgn > 0, sgn * u('+'), sgn * u('-') )
```

[5]:
```
from dune.fem.scheme import galerkin

tau = Constant(0, name="tau")
detPsi = abs(det(nabla_grad(x + tau * em)))

a = (u * detPsi - uh_old) * v * dx
a += tau * (g(uh_old, n) - h(uh_old, n)) * jump(v) * dS
a += tau * gBnd(uh_old, n) * v * ds

scheme = galerkin([a == 0], solver=("suitesparse","umfpack"))


# Compute CFL time step using cell edge lengths
l, ll = TrialFunction(space), TestFunction(space)
b = (l - MinCellEdgeLength(space)) * ll * dx
lscheme = galerkin([b == 0], solver=("suitesparse","umfpack"))
lh = space.interpolate(0, name="lh")
```

(continues on next page)

(continued from previous page)

```python
def dtCFL():
    lscheme.solve(lh)
    lMin = np.min(lh.as_numpy)
    return 0.5 * lMin / speed()[0]
```

4.2.4 Timeloop without adaptation

For comparison, we run the finite volume scheme once without adaptation of the mesh.

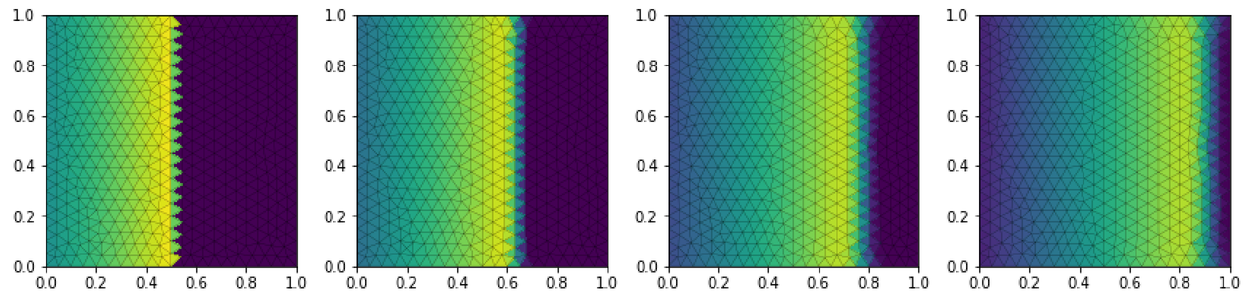
[6]:
```python
from time import time
from dune.fem.function import integrate
from dune.fem.plotting import plotPointData as plot
import matplotlib.pyplot as plt
plots = 4
fig, axs = plt.subplots(1, plots, figsize=(3*plots,3))
def shouldPlot(a):
    t_a = a * tEnd / (plots-1)
    return t.value > t_a and t.value - t_a <= tau.value

runtime = 0

uh.interpolate(u0(x))

def L2Error(uh):
    u = uexact(x, t)
    return sqrt(integrate(gridView, dot(uh-u, uh-u), order=5))

l2s = []

# Disable edge movement
em.interpolate([0,0])

tau.value = dtCFL()
print(f"dt = {tau.value:.4f}")

i = 0
t.assign(0)
while t.value < tEnd:
    runtime -= time()

    t.value += tau.value

    uh_old.assign(uh)
    scheme.solve(target=uh)
    l2s += [(t.value, L2Error(uh))]

    runtime += time()

    i += 1
    for a in range(plots):
        if shouldPlot(a):
```

(continues on next page)

(continued from previous page)

```
            plot(uh, figure=(fig, axs[a]), clim=[0,1], colorbar=None)

print(f"Runtime: {runtime:.3f}s")

L2 = L2Error(uh)
print(f"L2-Error: {L2:.3f}")
```

```
dt = 0.0169
Runtime: 0.905s
L2-Error: 0.107
```



4.2.5 Timeloop with adaptation

We need to set the "`fem.adaptation.method`" parameter to "`callback`" in order to use the non-hierarchical adaptation strategy of Dune-MMesh. Then, within the time loop, we can adapt the mesh according to the following strategy.

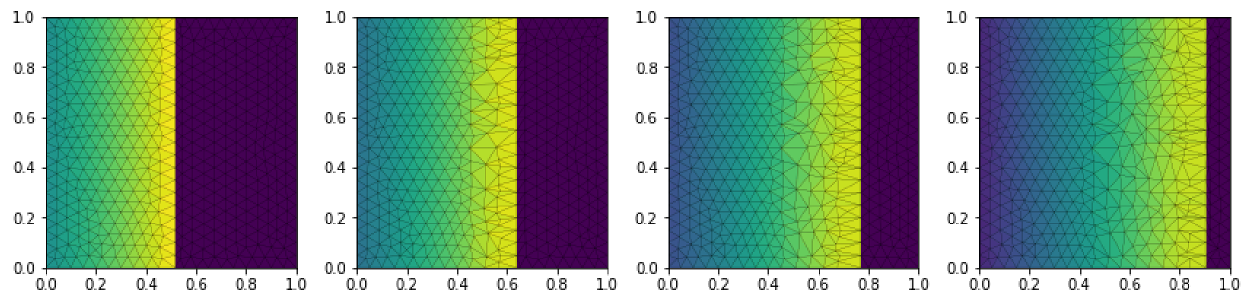
[7]:
```
from dune.fem import parameter, adapt
parameter.append( { "fem.adaptation.method": "callback" } )
fig.clear()
fig, axs = plt.subplots(1, plots, figsize=(3*plots,3))
runtimeAdapted = 0
dtmin = 1e3

uh.interpolate(u0(x))

l2sAdapted = []

i = 0
t.assign(0)
while t.value < tEnd:
    runtimeAdapted -= time()

    tau.value = dtCFL()
    dtmin = min(dtmin, tau.value)

    t.value += tau.value

    mark = hgrid.markElements()
    ensure = hgrid.ensureInterfaceMovement(getShifts()*tau.value)
    if mark or ensure:
        adapt([uh])
```

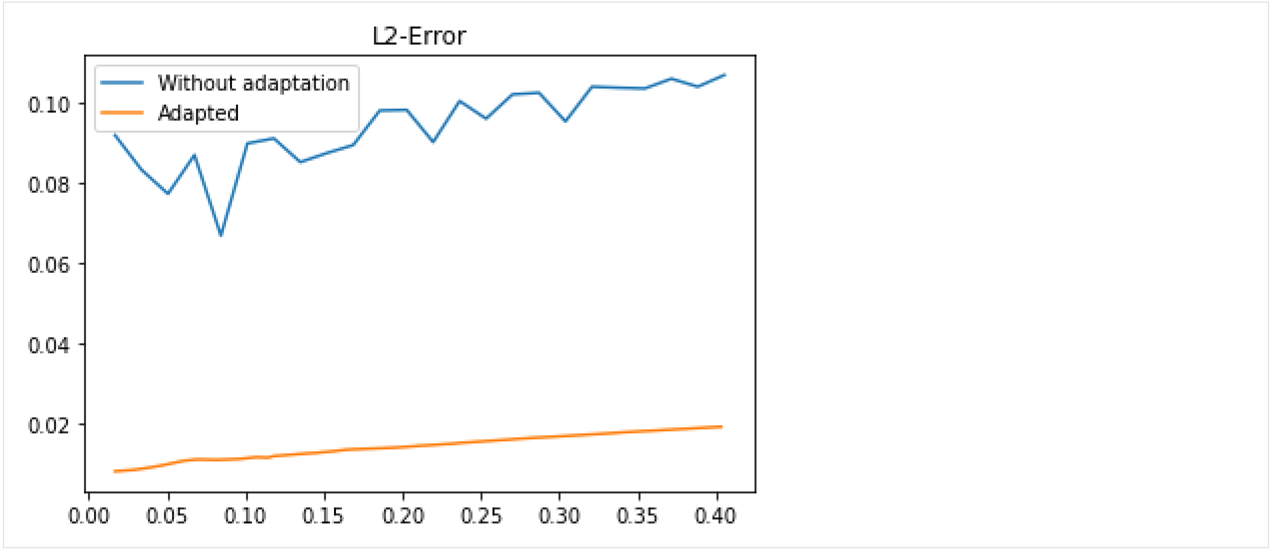
(continues on next page)

(continued from previous page)

```
    shifts = getShifts()
    em.assign(edgeMovement(gridView, shifts))

    uh_old.assign(uh)
    scheme.solve(target=uh)

    hgrid.moveInterface(shifts*tau.value)
    l2sAdapted += [(t.value, L2Error(uh))]

    runtimeAdapted += time()

    i += 1
    for a in range(plots):
        if shouldPlot(a):
            plot(uh, figure=(fig, axs[a]), clim=[0,1], colorbar=None)


print(f"dt_min = {dtmin:.4f}")
print(f"Runtime: {runtimeAdapted:.3f}s")

L2Adapted = L2Error(uh)
print(f"L2-Error: {L2Adapted:.3f}")

print(f"\nRuntime factor: {runtimeAdapted / runtime:.2f}x")
print(f"Error improvement: {L2 / L2Adapted:.2f}x")

import matplotlib.pyplot as plt
fig = plt.figure()
plt.plot([l2[0] for l2 in l2s], [l2[1] for l2 in l2s], label="Without adaptation")
plt.plot([l2[0] for l2 in l2sAdapted], [l2[1] for l2 in l2sAdapted], label="Adapted")
plt.legend(loc='best')
plt.title('L2-Error')
plt.show()
```

```
dt_min = 0.0034
Runtime: 2.235s
L2-Error: 0.019

Runtime factor: 2.47x
Error improvement: 5.59x
```





4.3 Manual adaptation

This is an example of how to manually remove vertices and refine edges. With this, we can also implement anisotropic refinement.

4.3.1 Grid creation

We use the *horizontal* grid file that contains an interface $\Gamma = [0.25, 0.75] \times 0.5$ embedded in a domain $\Omega = [0, 1]^2$.

[1]:
```
from dune.grid import reader
from dune.mmesh import mmesh

from dune.fem.view import adaptiveLeafGridView as adaptive
from dune.fem import parameter, adapt
parameter.append( { "fem.adaptation.method": "callback" } )

dim = 2
file = "grids/horizontal.msh"

gridView  = adaptive( mmesh((reader.gmsh, file), dim) )
hgrid = gridView.hierarchicalGrid
```

4.3.2 Interpolating some high order function

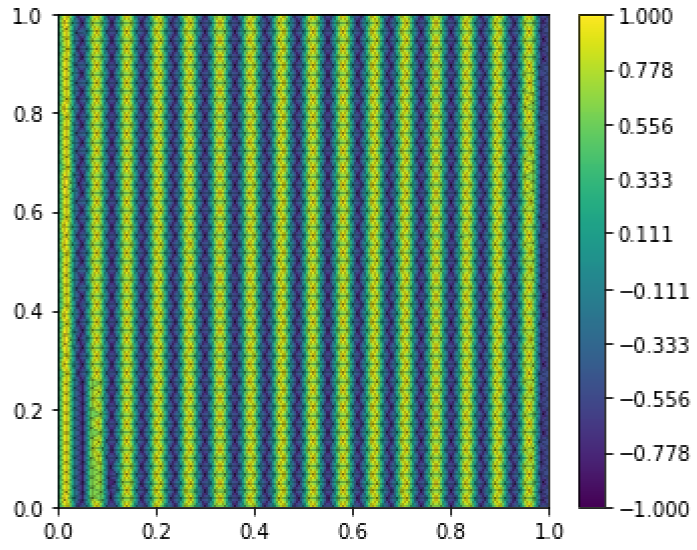
First, we interpolate some high order function.

[2]:
```
from ufl import *
from dune.fem.space import dglagrange as functionspace

space = functionspace(gridView, order=5)
x = SpatialCoordinate(space)
```

(continues on next page)

(continued from previous page)

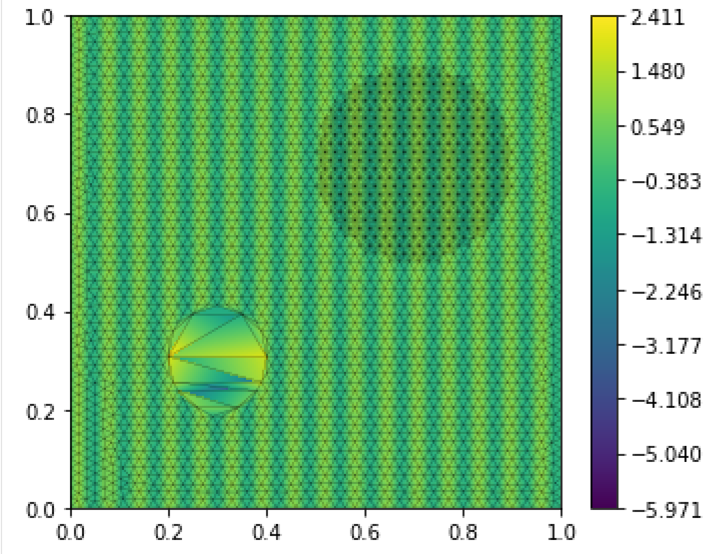
```
uh = space.interpolate( sin(100*x[0]), name="uh")
uh.plot()
```



4.3.3 Removing and inserting vertices

Then, we can insert and remove arbitrary vertices. We flag the vertices that should be removed and define points to be inserted within cells. Points inserted in cells will be connected to the all of the cell's vertices. Afterwards, we simply call adapt.

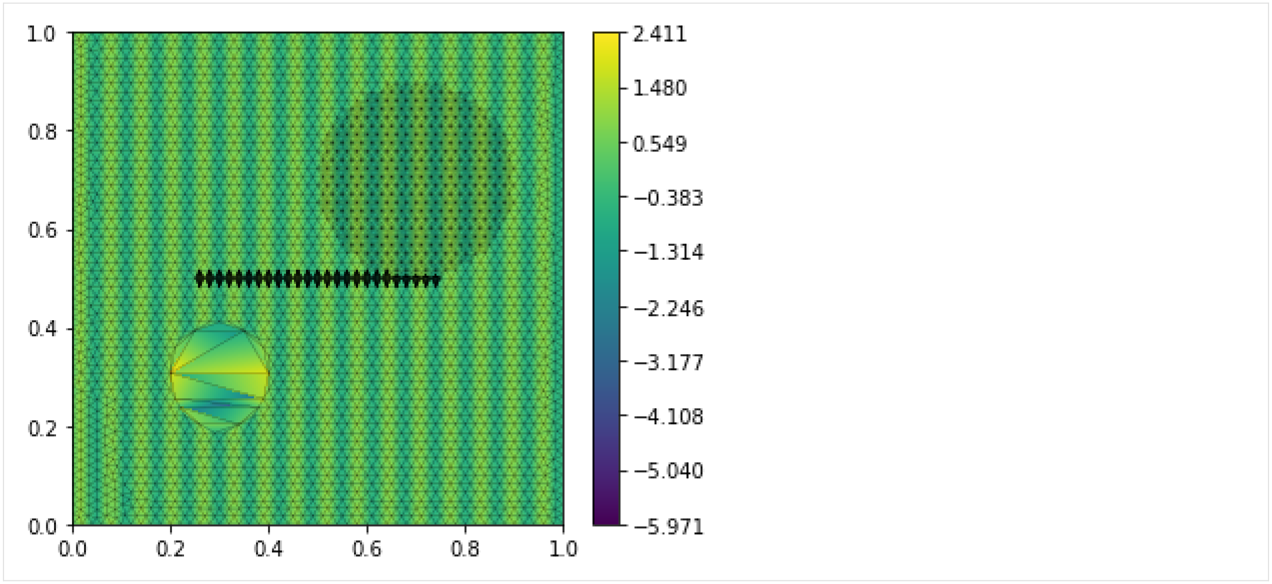
[3]:
```
from dune.common import FieldVector

for v in gridView.vertices:
    if (v.geometry.center - FieldVector([0.3, 0.3])).two_norm < 0.1:
        hgrid.removeVertex(v)

for e in gridView.elements:
    if (e.geometry.center - FieldVector([0.7, 0.7])).two_norm < 0.2:
        hgrid.insertVertexInCell( e.geometry.center )

adapt([uh])
uh.plot()
```



4.3.4 Refining edges

Another possibility to change the grid is to bisect specific edges. Hereby, we have to define an edge for bisection by a corresponding intersection's index.

[4]:
```
def myRefine():
    for e in gridView.elements:
        for i in gridView.intersections(e):
            if hgrid.isInterface(i):
                hgrid.refineEdge(e, i.indexInInside)

for i in range(5):
    myRefine()
    adapt([uh])

uh.plot()
```



4.4 Mixed-dimensional problem

A strength of Dune-MMesh is its application to mixed-dimensional problems. So, let us consider a mixed-dimensional Burger's equation on a domain with a lower-dimensional interface Γ that represents a physical domain with aperture $d > 0$.

Find u, u_Γ s.t

$$u_t + \operatorname{div} \mathbf{F}(u) = 0, \qquad \text{in } \Omega, \tag{10}$$

$$(du_\Gamma)_t + \operatorname{div} c\mathbf{F}(u_\Gamma) = [\mathbf{F}(u) \cdot \mathbf{n}], \qquad \text{in } \Gamma, \tag{11}$$

where $\mathbf{F}(u) = \frac{1}{2}u^2\mathbf{a}$ with $\mathbf{a} = [1, 0]^T$ and $c > 0$ is a speed scaling for the interface problem.

We impose the interior boundary condition for the bulk problem

$$F(u) = F(u_\Gamma), \quad \text{on } \Gamma, \tag{12}$$

$$\tag{13}$$

and consider inflow from left on the *T-junction* grid.

[1]:
```
from dune.grid import reader
from dune.mmesh import mmesh

file = "grids/tjunction.msh"

gridView = mmesh((reader.gmsh, file), 2)
igridView = gridView.hierarchicalGrid.interfaceGrid
```

[2]:
```
from ufl import *
from dune.ufl import Constant

d = Constant(0.01, name="d")
a = as_vector([1, 0])
c = Constant(100, name="c")
```

(continues on next page)

(continued from previous page)

```
def F(u):
    return 0.5 * u**2 * a

def upwind(u_l, u_r, n):
    return inner( conditional( inner(a, n) > 0, F(u_l), F(u_r) ), n )

dt = 0.25
tau = Constant(dt, name="tau")
```

4.4.1 Bulk problem

We use a finite volume space and define the UFL form of the bulk problem. Hereby, we used the interface indicator I to distinguish between interface and non-interface facets.

[3]:
```
from dune.fem.space import finiteVolume
space = finiteVolume(gridView)

u  = TrialFunction(space)
uu = TestFunction(space)

x = SpatialCoordinate(space)
n = FacetNormal(space)

left = conditional(x[0] < 1e-6, 1, 0)
uD = 1

uh = space.interpolate(0, name="uh")
uhOld = uh.copy()

from dune.mmesh import interfaceIndicator
I = interfaceIndicator(igridView)

A  = (u - uhOld) * uu * dx
A += tau * upwind(u('+'), u('-'), n('+')) * jump(uu) * (1-I)*dS
A += tau * upwind(u, uD, n) * uu * left * ds
```

4.4.2 Interface problem

Similarly, we define the UFL form of the interface problem.

[4]:
```
ispace = finiteVolume(igridView)

u_g  = TrialFunction(ispace)
uu_g = TestFunction(ispace)

n_g = FacetNormal(ispace)

uh_g = ispace.interpolate(0, name="uh_g")
uhOld_g = uh_g.copy()
```

(continues on next page)

(continued from previous page)

```
A_g  = d * (u_g - uhOld_g) * uu_g * dx
A_g += tau * c * upwind(u_g('+'), u_g('-'), n_g('+')) * jump(uu_g) * dS
```

4.4.3 Coupling condition

The coupling condition can be incorporated using the trace and skeleton functionality.

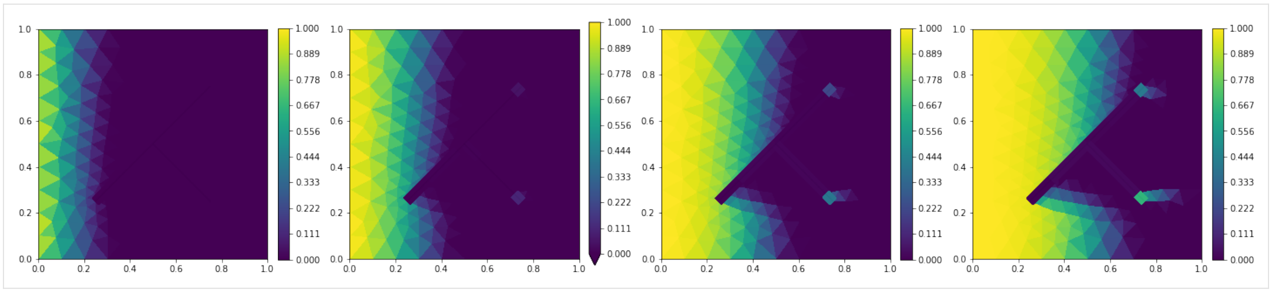
[5]:
```
from dune.mmesh import skeleton, trace, normals

u_gamma = avg(skeleton(uh_g))

A += tau * upwind(u('+'), u_gamma, n('+')) * uu('+') * I*dS
A += tau * upwind(u('-'), u_gamma, n('-')) * uu('-') * I*dS


traceu = trace(uh)
inormal = normals(igridView)

A_g -= tau * upwind(traceu('+'), u_g, -inormal) * uu_g * dx
A_g -= tau * upwind(traceu('-'), u_g, -inormal) * uu_g * dx
```

4.4.4 Monolithic solution strategy

Dune-MMesh provides coupled solution strategies to solve bulk and interface schemes together, either iteratively or monolithically. Here, we choose the monolithicSolve method that implements a mixed-dimensional Newton method.

[6]:
```
from dune.fem.scheme import galerkin
scheme   = galerkin([A   == 0])
scheme_g = galerkin([A_g == 0])

from dune.mmesh import monolithicSolve
def solve():
    uhOld.assign(uh)
    uhOld_g.assign(uh_g)
    monolithicSolve(schemes=(scheme, scheme_g), targets=(uh, uh_g), verbose=False)
```

[7]:
```
import matplotlib.pyplot as plt
from dune.fem.plotting import plotPointData as plot

fig, axs = plt.subplots(1, 4, figsize=(20,5))

for i in range(4):
    solve()

    plot(uh, figure=(fig, axs[i]), gridLines=None, clim=[0,1])
    plot(uh_g, figure=(fig, axs[i]), linewidth=0.04, clim=[0,1], colorbar=None)
```



4.5 Poroelasticity

We consider quasi-static linear Biot-equations with an interface Γ considered as thin heterogeneity with different permeability.

Find $\mathbf{u}, p, p_\Gamma$ s.t

$$\begin{aligned} -\operatorname{div}(\mathbf{K}\nabla p) &= 0, \quad \text{in } \Omega, && (14)\\ -\operatorname{div}\left(\sigma(\mathbf{u}) - \alpha p I\right) &= 0, \quad \text{in } \Omega, && (15)\\ -\operatorname{div}(\mathbf{K}_\Gamma \nabla p_\Gamma) &= q_\Gamma, \quad \text{in } \Gamma, && (16)\\ & && (17)\end{aligned}$$

where

$$\begin{aligned} \sigma(\mathbf{u}) &= \lambda \operatorname{tr} \epsilon(\mathbf{u}) I + 2\mu\epsilon(\mathbf{u}), && (18)\\ \epsilon(\mathbf{u}) &= \frac{1}{2}\left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\right) && (19)\\ & && (20)\end{aligned}$$

and $\lambda, \mu > 0$ are the Lamé constants, α the Biot-Willis constant, $\mathbf{K}$, $\mathbf{K}_\Gamma$ bulk and interface permeability and q_Γ a source term.

We impose interior boundary conditions

$$\begin{aligned} p &= p_\Gamma, && (21)\\ \left(\sigma(\mathbf{u}) - \alpha p I\right) \cdot \mathbf{n} &= -p_\Gamma \mathbf{n}. && (22)\end{aligned}$$

and consider a *T-junction* as grid geometry.

[1]:
```python
from dune.grid import reader
from dune.mmesh import mmesh

file = "grids/tjunction.msh"

gridView = mmesh((reader.gmsh, file), 2)
igridView = gridView.hierarchicalGrid.interfaceGrid
```

[2]:
```python
from ufl import *
from dune.ufl import Constant

lamb    = Constant( 1.2, name="lambda")
mu      = Constant( 0.8, name="mu")
```

(continues on next page)

(continued from previous page)

```
alpha   = Constant(   1, name="alpha")
K       = as_matrix([[1e-4, 0], [0, 1e-6]])
K_g     = Constant(   1, name="K_g")
q       = Constant(1e-6, name="q")

epsilon = lambda u: 0.5 * (nabla_grad(u) + nabla_grad(u).T)
sigma   = lambda u: lamb * div(u) * Identity(2) + 2 * mu * epsilon(u)
```

[3]:
```
from dune.fem.space import dglagrange

# Bulk
space = dglagrange(gridView, dimRange=3, order=1)

trial = TrialFunction(space)
test = TestFunction(space)

p,  ux,  uy  = split(trial)
pp, uux, uuy = split(test)

u  = as_vector([ux,  uy ])
uu = as_vector([uux, uuy])

x = SpatialCoordinate(space)
n = FacetNormal(space)
h = FacetArea(space)

left = conditional(x[0] < 1e-6, 1, 0)
right = conditional(x[0] > 1-1e-6, 1, 0)
bottom = conditional(x[1] < 1e-6, 1, 0)
top = conditional(x[1] > 1-1e-6, 1, 0)

solution = space.interpolate([0,0,0], name="solution")

# Interface
space_g = dglagrange(igridView, order=1)

p_g  = TrialFunction(space_g)
pp_g = TestFunction(space_g)

x_g = SpatialCoordinate(space_g)
n_g = FacetNormal(space_g)
h_g = FacetArea(space_g)

solution_g = space_g.interpolate(0, name="solution_g")
```

4.5.1 Interior Penalty Discontinuous Galerkin (IPDG) Scheme

We implement a mixed-dimensional Interior Penalty Discontinuous Galerkin (IPDG) scheme which directly allows for a discontinuity along the interface. Including the penalty and consistency terms, we have the following weak form.

$$\begin{aligned}
A(\mathbf{u}, p; \mathbf{v}, \varphi) := &\int_\Omega (\mathbf{K}\nabla p) \cdot \nabla\varphi \, dx - \int_\Omega q\varphi \, dx && (23)\\
&+ \int_{\mathcal{F}\setminus\mathcal{F}_\Gamma} \frac{\beta}{h}[p][\varphi] - \{\{\mathbf{K}\nabla p \cdot \mathbf{n}\}\}[\varphi] \, dS + \int_{\mathcal{F}_D^p} \frac{\beta}{h}(p - p_D)\varphi - \mathbf{K}\nabla p \cdot \mathbf{n}\varphi \, ds && (24)\\
&+ \int_{\mathcal{F}_\Gamma} \frac{\beta}{h}(p^+ - p_\Gamma)\varphi^+ - \mathbf{K}\nabla p^+ \cdot \mathbf{n}^+\varphi^+ \, dS && (25)\\
&+ \int_{\mathcal{F}_\Gamma} \frac{\beta}{h}(p^- - p_\Gamma)\varphi^- - \mathbf{K}\nabla p^- \cdot \mathbf{n}^-\varphi^- \, dS && (26)\\
&+ \int_\Omega (\sigma(\mathbf{u}) - \alpha p I) : \epsilon(\mathbf{v}) \, dx + \int_{\mathcal{F}\setminus\mathcal{F}_\Gamma} \frac{\beta}{h}[\mathbf{u}][\mathbf{v}] - (\{\{\sigma(\mathbf{u}) - \alpha p I\}\} \cdot \mathbf{n}) \cdot [\mathbf{v}] \, dS && (27)\\
&+ \int_{\mathcal{F}_D^u} \frac{\beta}{h}(\mathbf{u} - \mathbf{u}_D) \cdot \mathbf{v} - ((\sigma(\mathbf{u}) - \alpha p I) \cdot \mathbf{n}) \cdot \mathbf{v} \, ds && (28)\\
&- \int_{\mathcal{F}_\Gamma} -p_\Gamma(\mathbf{v}^+ \cdot \mathbf{n}^+) - p_\Gamma(\mathbf{v}^- \cdot \mathbf{n}^-) \, dS && (29)\\
A_\Gamma(p_\Gamma; \varphi_\Gamma) := &\int_\Gamma (\mathbf{K}_\Gamma \nabla p_\Gamma) \cdot \nabla\varphi_\Gamma - q\varphi_\Gamma + \beta(p_\Gamma - \{\{p\}\})\varphi_\Gamma \, dx && (30)\\
&+ \int_{\mathcal{F}^\Gamma} \frac{\beta}{h_\Gamma}[p_\Gamma][\varphi_\Gamma] - \{\{\mathbf{K}_\Gamma \nabla p_\Gamma \cdot \mathbf{n}_\Gamma\}\}[\varphi_\Gamma] \, dS && (31)
\end{aligned}$$

This weak form is implemented using UFL and Dune-MMesh's trace and skeleton functionality as follows.

```
[4]: from dune.mmesh import skeleton, trace, interfaceIndicator

     I = interfaceIndicator(igridView)
     beta = Constant(1e2, name="beta")

     p_gamma = avg(skeleton(solution_g))
     tracep = trace(solution)[0]


     # Pressure
     a  = inner(K * grad(p), grad(pp)) * dx
     a -= q * pp * dx

     a += beta / h * inner(jump(p), jump(pp)) * (1-I)*dS
     a -= inner(avg(K * grad(p)), n('+')) * jump(pp) * (1-I)*dS

     a += beta / h * (p - 0) * pp * (left+right) * ds
     a -= inner(K * grad(p), n) * pp * (left+right) * ds
```

(continues on next page)

(continued from previous page)

```
# Interface pressure
a_g  = inner(K_g * grad(p_g), grad(pp_g)) * dx
a_g -= q * pp_g * dx

a_g += beta / h_g * inner(jump(p_g), jump(pp_g)) * dS
a_g -= inner(avg(K_g * grad(p_g)), n_g('+')) * jump(pp_g) * dS


# Pressure is continuous at the interface
a += beta * (p('+') - p_gamma) * pp('+') * I*dS
a -= inner(K * grad(p('+')), n('+')) * pp('+') * I*dS

a += beta * (p('-') - p_gamma) * pp('-') * I*dS
a -= inner(K * grad(p('-')), n('-')) * pp('-') * I*dS

a_g += beta * (p_g - avg(tracep)) * pp_g * dx


# Displacement
sigma_p = lambda u, p: sigma(u) - alpha * p * Identity(2)
a += inner(sigma_p(u, p), epsilon(uu)) * dx

a += beta / h * inner(jump(u), jump(uu)) * (1-I)*dS
a -= dot(dot(avg(sigma_p(u, p)), n('+')), jump(uu)) * (1-I)*dS

a += beta / h * inner(u - as_vector([0,0]), uu) * (top+bottom) * ds
a -= dot(dot(sigma_p(u, p), n), uu) * (top+bottom) * ds


# Normal stress is -p at the interface
a -= -p_gamma * inner(uu('+'), n('+')) * I*dS
a -= -p_gamma * inner(uu('-'), n('-')) * I*dS
```

4.5.2 Monolithic solution strategy

We use the `monolithicSolve` to obtain the solution of the strongly coupled mixed-dimensional system.

```
[5]: from dune.fem.scheme import galerkin
scheme   = galerkin([a   == 0], solver=("suitesparse", "umfpack"))
scheme_g = galerkin([a_g == 0], solver=("suitesparse", "umfpack"))

from dune.mmesh import monolithicSolve
converged = monolithicSolve(schemes=(scheme, scheme_g), targets=(solution, solution_g),␣
↪verbose=True)
```

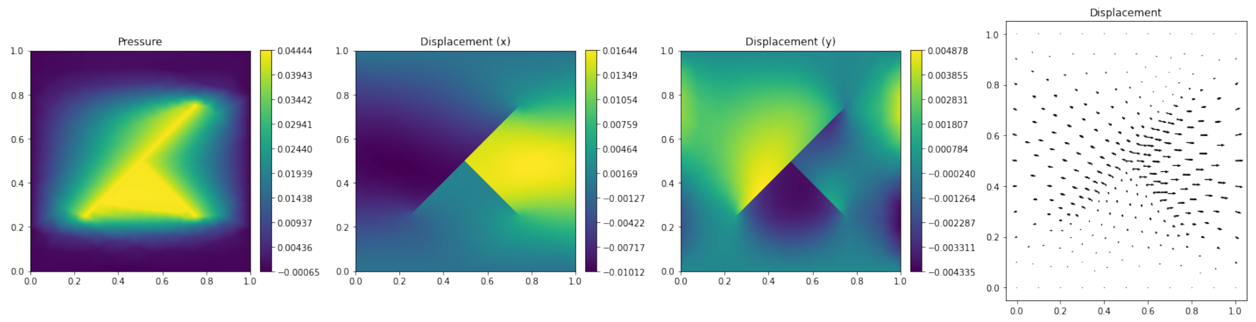
```
 i: 1  |x| = 1.19366078e+00  |f| = 1.72475757e-14
 i: 2  |x| = 5.35261373e-09  |f| = 9.79676418e-15
```

```
[6]: import matplotlib.pyplot as plt
from dune.fem.plotting import plotPointData as plot
```

(continues on next page)

(continued from previous page)

```
fig, axs = plt.subplots(1, 4, figsize=(20,5))
axs[0].set_title('Pressure')
plot(solution[0], figure=(fig, axs[0]), gridLines=None)
axs[1].set_title('Displacement (x)')
plot(solution[1], figure=(fig, axs[1]), gridLines=None)
axs[2].set_title('Displacement (y)')
plot(solution[2], figure=(fig, axs[2]), gridLines=None)
axs[3].set_title('Displacement')
plot(solution, vectors=[1,2], figure=(fig, axs[3]), gridLines=None)
```



4.6 Two-phase Navier-Stokes with surface tension

Let's consider the incompressible Navier-Stokes equations with two immiscible phases and surface tension to demonstrate more complex usage of Dune-MMesh's capabilities.

A domain $\Omega \subset \mathbb{R}^2$ is assumed to be separeted into two phases $\Omega_i(t), i = 1, 2$, by a sharp interface $\Gamma(t)$.

Find (u, p) and $\Gamma(t)$ s.t.

$$ \tag{32} $$

$$\rho u_t + \nabla \cdot (\rho u \otimes u) + \nabla \cdot T(u,p) = 0, \qquad \text{in } \Omega_i(t), \qquad i = 1,2, \tag{33}$$

$$\nabla \cdot u = 0, \qquad \text{in } \Omega_i(t), \qquad i = 1,2, \tag{34}$$

$$[p] = \sigma\kappa \cdot n, \qquad \text{on } \Gamma(t), \tag{35}$$

$$[u] = 0, \qquad \text{on } \Gamma(t), \tag{36}$$

$$\dot{x} = u, \qquad x \in \Gamma(t), \tag{37}$$

$$u(0) = u_0, \qquad \text{in } \Omega_i(0), \qquad i = 1,2, \tag{38}$$

$$\Gamma(0) = \Gamma_0. \tag{39}$$

Here, $T(u,p) := pI - \mu(\nabla u + (\nabla u)^T))$ is the stress tensor, $\mu_i, i = 1, 2$, are the dynamic viscositities and $\rho_i, i = 1, 2$, the densities of the two phases, σ is the surface tension and κ is the signed mean curvature of the interface times its normal.

Now, we use Dune-MMesh to compute the dynamics of a droplet in a channel. Let us consider the *circle* grid.

[1]:
```
from dune.grid import reader
from dune.mmesh import mmesh, trace, skeleton, domainMarker
from dune.fem.view import adaptiveLeafGridView as adaptive

dim = 2
```

(continues on next page)

(continued from previous page)

```
file = "grids/circle.msh"

gridView = adaptive( mmesh((reader.gmsh, file), dim) )
hgrid = gridView.hierarchicalGrid
igridView = adaptive( hgrid.interfaceGrid )
```

[2]:
```
from ufl import *
from dune.ufl import Constant

mu0   = Constant(   1, name="mu0")
mu1   = Constant(   1, name="mu1")
rho0  = Constant(   1, name="rho0")
rho1  = Constant(   2, name="rho1")
sigma = Constant(0.03, name="sigma")

dt = 0.05
T  = 12.0
tau  = Constant(dt, name="tau")
```

4.6.1 Domain markers

The domain markers are initialized by the physical identifiers passed from the mesh file. We use them to interpolate the phase's parameters.

[3]:
```
from dune.mmesh import domainMarker

dm = domainMarker(gridView)
mu  = (1-dm) * mu0  + dm * mu1
rho = (1-dm) * rho0 + dm * rho1
```

[4]:
```
from dune.fem.space import lagrange, dglagrange

pspace = dglagrange(gridView, order=1)
p = TrialFunction(pspace)
q = TestFunction(pspace)

uspace = dglagrange(gridView, dimRange=dim, order=2)
u = TrialFunction(uspace)
v = TestFunction(uspace)

ph = pspace.interpolate(0, name="ph")
uh = uspace.interpolate([0,0], name="uh")
uh1 = uspace.interpolate([0,0], name="uh1")
```

4.6.2 Curvature

The mean curvature κ of the interface times its normal can be computed by solving

$$\int_\Gamma \kappa \cdot \phi + \nabla x \cdot \nabla \phi \, dS = 0, \qquad \text{in } \Gamma(t).$$

[5]:
```
from dune.fem.space import lagrange
from dune.fem.scheme import galerkin

kspace = lagrange(igridView, dimRange=dim, order=1)
k  = TrialFunction(kspace)
kk = TestFunction(kspace)

curvature = kspace.interpolate([0]*dim, name="curvature")

ix = SpatialCoordinate(kspace)
C  = inner(k, kk) * dx
C -= inner(grad(ix), grad(kk)) * dx

kscheme = galerkin([C == 0])
res = kscheme.solve(curvature)
```

4.6.3 Moving

We will move the interface movement by evaluating the trace of the bulk velocity u.

[6]:
```
import numpy as np

x = SpatialCoordinate(pspace)
n = FacetNormal(pspace)
h = FacetArea(pspace)

def getShifts():
    mapper = hgrid.interfaceGrid.indexSet
    shifts = np.zeros((igridView.size(dim-1), dim))
    for e in igridView.elements:
        for v in e.subEntities(dim-1):
            x = e.geometry.toLocal(v.geometry.center)
            shifts[ mapper.index(v) ] = trace(uh)(e, x)
    return shifts
```

4.6.4 Navier-Stokes equations

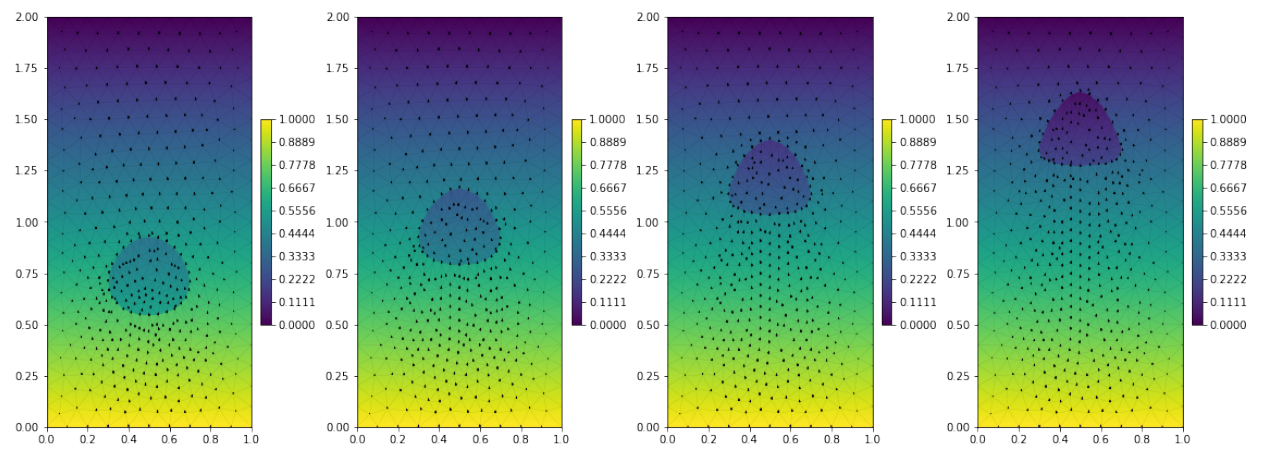
We implement a splitting scheme similar to the one presented in [GBK20].

[7]:
```
from dune.mmesh import skeleton, interfaceIndicator
I = interfaceIndicator(igridView)

penu = Constant(1e6, name="penaltyu")
penp = Constant(1e6, name="penaltyp")

noslip = conditional(x[0] < 1e-6, 1, 0) + conditional(x[0] > 1-1e-6, 1, 0)

a1  = rho * inner(u - uh, v) / tau * dx
a1 += inner(grad(uh) * uh, v) * dx
a1 += inner(mu * grad(u), grad(v)) * dx

a1 += penu / h * inner(jump(u), jump(v)) * dS
a1 += dot(dot(avg(mu * grad(u)), n('+')), jump(v)) * dS
a1 += penu / h * inner(u - zero(dim), v) * noslip * ds
a1 += dot(dot(mu * grad(u), n), v) * noslip * ds

A1 = galerkin([a1 == 0], solver=("suitesparse", "umfpack"))


dirichlet = conditional(x[1] < 1e-6, 1, 0) + conditional(x[1] > 2-1e-6, 1, 0)
pD = conditional(x[1] < 1e-6, 1, 0)

a2  = inner(grad(p), grad(q)) * dx
a2 += penp / h * jump(p) * jump(q) * dS
a2 += dot(dot(avg(grad(p)), n('+')), jump(q)) * dS
a2 += penp / h * (p - pD) * q * dirichlet * ds
a2 += dot(dot(grad(p), n), q) * ds
a2 += inner(rho * div(uh1), q) / tau * dx

kappa = avg(skeleton(curvature))
a2 += penp / h * inner(sigma * kappa, n('+')) * jump(q) * I*dS

A2 = galerkin([a2 == 0], solver=("suitesparse", "umfpack"))


a3  = rho * inner(u - uh1, v) / tau * dx
a3 += inner(grad(ph), v) * dx

a3 += penu / h * inner(jump(u), jump(v)) * dS
a3 += penu / h * inner(u - zero(dim), v) * noslip * ds

A3 = galerkin([a3 == 0], solver=("suitesparse", "umfpack"))
```

4.6.5 Timeloop

Finally, we specify the time iteration where we adapt the mesh and solve the schemes.

```
[8]: from dune.fem import parameter, adapt
     parameter.append( { "fem.adaptation.method": "callback" } )
     from dune.fem.plotting import plotPointData as plot
     import matplotlib.pyplot as plt

     N = 4
     i = 0
     fig, axs = plt.subplots(1, N, figsize=(16,6))

     ph.interpolate(0)
     uh.interpolate([0,0])
     uh1.interpolate([0,0])

     step = 0
     t = 0
     while t < T+dt:

         hgrid.moveInterface( dt*getShifts() )

         hgrid.markElements()
         adapt([ph, uh, uh1, dm])
         adapt([curvature])

         A1.solve(uh1)
         A2.solve(ph)
         A3.solve(uh)

         if int(N * t/T) > i:
             plot(ph, figure=(fig, axs[i]), gridLines='black', linewidth=0.02)
             plot(uh, figure=(fig, axs[i]), gridLines=None, vectors=[0,1])
             i += 1

         t += dt
```



All examples can be found in the directory *doc/examples* as IPython notebooks.

Some examples for the creation of grid files can be found in *Grid files*.

4.7 Grid files

Some examples for the creation of grid files:

4.7.1 circle.geo

```
lc = 0.1;
lcf = 0.05;

// Domain
Point(1) = {0, 0, 0, lc};
Point(2) = {1, 0, 0, lc};
Point(3) = {1, 2, 0, lc};
Point(4) = {0, 2, 0, lc};

// Points of circle
Point(5)  = {0.5, 0.5, 0, lcf};
Point(6)  = {0.5, 0.7, 0, lcf};
Point(7)  = {0.3, 0.5, 0, lcf};
Point(8)  = {0.5, 0.3, 0, lcf};
Point(9)  = {0.7, 0.5, 0, lcf};

// Domain outline
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};

// Interface outline
Ellipse(5) = {6, 5, 7};
Ellipse(6) = {7, 5, 8};
Ellipse(7) = {8, 5, 9};
Ellipse(8) = {9, 5, 6};

Curve Loop(1) = {5:8};
Curve Loop(2) = {1:4};

Plane Surface(0) = {2,1};
Plane Surface(1) = {1};
Physical Surface(0) = {0};
Physical Surface(1) = {1};

Physical Curve(10) = {5:8};
```

This file can be meshed using `gmsh -2 -format msh2 circle.geo`.

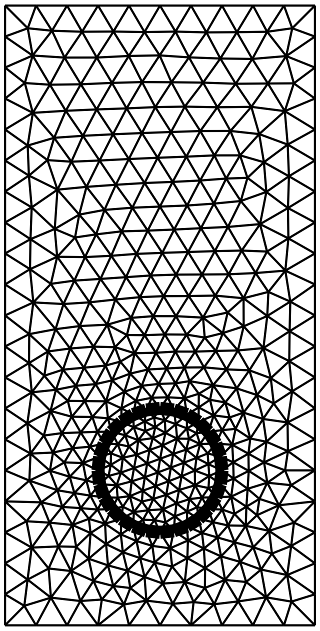


Fig. 9: circle.msh

4.7.2 horizontal.py

```
# A rectangle grid with a horizontal centered interface

name = "horizontal.msh"
h = 0.02
hf = h

import gmsh
gmsh.initialize()
gmsh.option.setNumber("General.Verbosity", 0)
gmsh.option.setNumber("Mesh.MshFileVersion", 2.2)

gmsh.model.add(name)

p1 = gmsh.model.geo.addPoint(0, 0, 0, h, 1)
p2 = gmsh.model.geo.addPoint(1, 0, 0, h, 2)
p3 = gmsh.model.geo.addPoint(1, 1, 0, h, 3)
p4 = gmsh.model.geo.addPoint(0, 1, 0, h, 4)

l1 = gmsh.model.geo.addLine(p1, p2, 1)
l2 = gmsh.model.geo.addLine(p2, p3, 2)
l3 = gmsh.model.geo.addLine(p3, p4, 3)
l4 = gmsh.model.geo.addLine(p4, p1, 4)

p5 = gmsh.model.geo.addPoint(0.25, 0.5, 0, hf, 5)
p6 = gmsh.model.geo.addPoint(0.75, 0.5, 0, hf, 6)
lf = gmsh.model.geo.addLine(p5, p6, 10)

gmsh.model.geo.addCurveLoop([l1, l2, l3, l4], 1)
```

(continues on next page)

(continued from previous page)

```
gmsh.model.geo.addPlaneSurface([1], 0)

gmsh.model.geo.synchronize()
gmsh.model.mesh.embed(1, [lf], 2, 0)

gmsh.model.mesh.generate(dim=2)
gmsh.write(name)
gmsh.finalize()
```

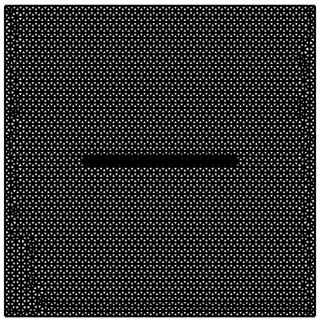


Fig. 10: horizontal.msh

4.7.3 tjunction.py

```
# A rectangle grid with a T-shaped junction

name = "tjunction.msh"
h = 0.1
hf = 0.05

import gmsh
gmsh.initialize()
gmsh.option.setNumber("General.Verbosity", 0)
gmsh.option.setNumber("Mesh.MshFileVersion", 2.2)

gmsh.model.add(name)

p1 = gmsh.model.geo.addPoint(0, 0, 0, h, 1)
p2 = gmsh.model.geo.addPoint(1, 0, 0, h, 2)
p3 = gmsh.model.geo.addPoint(1, 1, 0, h, 3)
p4 = gmsh.model.geo.addPoint(0, 1, 0, h, 4)

l1 = gmsh.model.geo.addLine(p1, p2, 1)
l2 = gmsh.model.geo.addLine(p2, p3, 2)
l3 = gmsh.model.geo.addLine(p3, p4, 3)
l4 = gmsh.model.geo.addLine(p4, p1, 4)

p5 = gmsh.model.geo.addPoint(0.25, 0.25, 0, hf, 5)
p6 = gmsh.model.geo.addPoint( 0.5,  0.5, 0, hf, 6)
p7 = gmsh.model.geo.addPoint(0.75, 0.75, 0, hf, 7)
```

(continues on next page)

(continued from previous page)

```
p8 = gmsh.model.geo.addPoint(0.75, 0.25, 0, hf, 8)

lf1 = gmsh.model.geo.addLine(p5, p6, 10)
lf2 = gmsh.model.geo.addLine(p6, p7, 11)
lf3 = gmsh.model.geo.addLine(p6, p8, 12)

gmsh.model.geo.addCurveLoop([l1, l2, l3, l4], 1)
gmsh.model.geo.addPlaneSurface([1], 0)

gmsh.model.geo.synchronize()
gmsh.model.mesh.embed(1, [lf1, lf2, lf3], 2, 0)

gmsh.model.mesh.generate(dim=2)
gmsh.write(name)
gmsh.finalize()
```

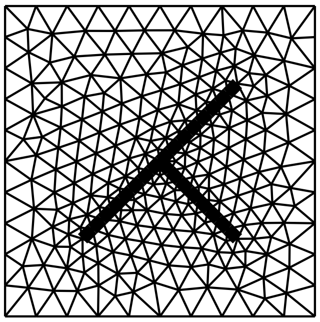


Fig. 11: tjunction.msh

4.7.4 vertical.py

```
# A rectangle grid with a vertical interface

name = "vertical.msh"
h = 0.05
hf = 0.05

import gmsh
gmsh.initialize()
gmsh.option.setNumber("General.Verbosity", 0)
gmsh.option.setNumber("Mesh.MshFileVersion", 2.2)

gmsh.model.add(name)

p1 = gmsh.model.geo.addPoint(   0, 0, 0, h, 1)
p2 = gmsh.model.geo.addPoint( 0.5, 0, 0, hf, 2)
p3 = gmsh.model.geo.addPoint(   1, 0, 0, h, 3)
p4 = gmsh.model.geo.addPoint(   1, 1, 0, h, 4)
p5 = gmsh.model.geo.addPoint( 0.5, 1, 0, hf, 5)
p6 = gmsh.model.geo.addPoint(   0, 1, 0, h, 6)
```

(continues on next page)

(continued from previous page)

```
l1 = gmsh.model.geo.addLine(p1, p2, 1)
l2 = gmsh.model.geo.addLine(p2, p3, 2)
l3 = gmsh.model.geo.addLine(p3, p4, 3)
l4 = gmsh.model.geo.addLine(p4, p5, 4)
l5 = gmsh.model.geo.addLine(p5, p6, 5)
l6 = gmsh.model.geo.addLine(p6, p1, 6)
lf = gmsh.model.geo.addLine(p2, p5, 10)

gmsh.model.geo.addCurveLoop([1, 10, 5, 6], 1)
gmsh.model.geo.addPlaneSurface([1], 0)
gmsh.model.geo.addCurveLoop([2, 3, 4, -10], 2)
gmsh.model.geo.addPlaneSurface([2], 1)

gmsh.model.geo.synchronize()

gmsh.model.mesh.generate(dim=2)
gmsh.write(name)
gmsh.finalize()
```

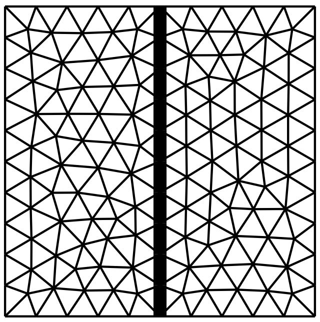


Fig. 12: vertical.msh

5 Python

In the following, we describe the additional python functionality that is not already part of Dune or dune-fem, and is exclusively available for Dune-MMesh.

5.1 Grid construction

5.2 Trace and skeleton

5.3 Utility functions

5.4 Coupled solve

6 C++

In the following, we describe the additional C++ class member functions that are not already part of the DUNE grid interface.

6.1 Grid

template<class **`HostGrid`**, int **`dim`**>

class **`MMesh`**

The *MMesh* class templatized by the CGAL host grid type and the dimension.

\

Public Functions

inline void **`addInterface`**(const Intersection &intersection, const std::size_t marker = 1)

Add an intersection to the interface.

template<class **`I`**>
inline void **`addInterface`**(const *I* &intersection, const std::size_t marker = 1)

Add wrapped intersection to the interface.

inline bool **`isOnInterface`**(const Entity &entity) const

Return if entity shares a facet with the interface.

inline Intersection **`asIntersection`**(const InterfaceEntity &interfaceEntity) const

Return an interface entity as intersection of a *MMesh* entity.

inline Intersection **`asIntersection`**(const Facet &facet) const

Return a facet as intersection.

inline Intersection **`asIntersection`**(const FacetHandle &host) const

Return a host facet as intersection.

inline const Entity **`locate`**(const GlobalCoordinate &p, const Entity &element = {}) const

Locate an entity by coordinate using CGAL's locate.

inline bool **`markElements`**()

Mark elements for adaption using the default remeshing indicator.

Returns
if elements have been marked.

inline bool **`ensureVertexMovement`**`(`std::vector<GlobalCoordinate> shifts`)`

Mark elements such that after movement of vertices no cell degenerates.

Parameters
`shifts` – Vector that maps vertex index to GlobalCoordinate

Returns
If elements have been marked.

inline void **`moveVertices`**`(`const std::vector<GlobalCoordinate> &shifts`)`

Move vertices.

Parameters
`shifts` – Vector that maps interface vertex indices to GlobalCoordinate

template<typename **`Vertex`**>
inline void **`addToInterface`**`(`const *Vertex* &vertex, const GlobalCoordinate &p`)`

Insert p into the triangulation and add a new interface segment between p and vertex

inline const *HostGrid* &**`getHostGrid`**`()` const

Get reference to the underlying CGAL triangulation.

inline *HostGrid* &**`getHostGrid`**`()`

Get non-const reference to the underlying CGAL triangulation.

inline const InterfaceGrid &**`interfaceGrid`**`()` const

Get reference to the interface grid.

inline InterfaceGrid &**`interfaceGrid`**`()`

Get a non-const reference to the interface grid.

6.2 Entity

template<int **`dim`**, class **`GridImp`**>

class **`MMeshEntity`**<0, *dim*, *GridImp*>

The implementation of entities in *MMesh*.

Subclassed by Dune::MMeshCachingEntity< 0, dim, GridImp >

Public Functions

inline std::size_t **`domainMarker`**`()` const

Return domain marker of entity.

inline const HostGridEntity &**`hostEntity`**`()` const

Return the host entity.

inline HostGridEntity &**`hostEntity`**`()`

Return the host entity.

inline const *GridImp* &**`grid`**`()` const

Return the host grid.

inline IdType **`id`**`()` const

Return id computed by vertex ids.

6.3 InterfaceGrid

template<class **`MMesh`**>

class **`MMeshInterfaceGrid`**

Provides a DUNE grid interface class for the interface of a *MMesh* interface grid.

\

Public Functions

inline const *MMesh* &**`getMMesh`**`()` const

Return reference to *MMesh*.

inline const HostGridType &**`getHostGrid`**`()` const

Return reference to underlying CGAL triangulation.

6.4 InterfaceEntity

template<int **`codim`**, int **`dim`**, class **`GridImp`**>

class **`MMeshInterfaceGridEntity`**

The implementation of entities in a *MMesh* interface grid.

Public Functions

inline bool **`isTip`**`()` const

Return if this vertex is a tip.

inline const MMeshInterfaceEntity &**`hostEntity`**`()` const

Return reference to the host entity.

6.5 Iterator Ranges

template<typename **`Entity`**>
inline auto `Dune::`**`incidentElements`**`(`const *Entity* &entity`)`

Elements incident to a given entity.

template<typename **`Vertex`**>
inline auto `Dune::`**`incidentFacets`**`(`const *Vertex* &vertex`)`

Facets incident to a given vertex.

template<typename **`Vertex`**>
inline auto `Dune::`**`incidentVertices`**`(`const *Vertex* &vertex, bool includeInfinite = false`)`

Vertices incident to a given vertex.

template<typename **`GridView`**, int **`codim`** = 1>
inline auto `Dune::`**`interfaceElements`**`(`const *GridView* &gv, bool includeBoundary = false`)`

All interface elements.

template<typename **`GridView`**>

inline auto Dune::**interfaceVertices**(const *GridView* &gv, bool includeBoundary = false)

All interface vertices.

template<typename **Vertex**>
inline auto Dune::**incidentInterfaceVertices**(const *Vertex* &vertex)

Incident interface vertices.

template<typename **Entity**>
inline auto Dune::**incidentInterfaceElements**(const *Entity* &entity)

Incident interface elements.

6.6 Indicator

template<class **Grid**>

class **RatioIndicator**

Class defining an indicator for grid remeshing regarding the edge length ratio. By default, we take 2x length of the longest edge contained in the interface as maximal edge length and 0.5x length of the shortest edge as minimal edge length.

Public Functions

inline **RatioIndicator**(ctype h = 0.0, ctype distProportion = 1.0, ctype factor = 1.0)

Calculates the indicator for each grid cell.

Parameters

- **h** – The objective edge length (aims at edge length in [h/4, 2*h]).
- **distProportion** – Cells with distance to interface of value greater than distProportion * max(dist) are refined to …
- **factor** – … edge length in [factor * minH, factor * maxH].

inline void **init**(const *Grid* &grid)

Calculates minH_ and maxH_ for the current interface edge length and sets factor_ to maxh / minh.

inline void **update**()

Update the distances of all vertices.

template<class **Element**>
inline int **operator()**(const *Element* &element) const

Function call operator to return mark.

Returns

1 if an element should be refined, -1 if an element should be coarsened, 0 otherwise.

inline ctype **maxH**() const

Returns maxH.

inline ctype &**maxH**()

Returns reference to maxH.

inline ctype **minH**() const

Returns minH.

inline ctype &**minH**()

Returns reference to minH.

inline ctype &**distProportion**()

Returns reference to distProportion.

inline ctype &**factor**()

Returns reference to factor.

6.7 Connected Component

template<class **GridImp**>

class **MMeshConnectedComponent**

The implementation of a connected component of entities in *MMesh*.

The connected component stores a list of connected entities providing geometrical information for the remeshing step.

Public Functions

inline const std::list<CachingEntity> &**children**() const

Return list of caching entities in this component.

inline const std::size_t **size**() const

Return number of caching entities in this component.

template<class **GridImp**>

class **MMeshInterfaceConnectedComponent**

The implementation of connected components in a *MMeshInterfaceGrid*.

The connected component copies the vertex coordinates and ids.

Public Functions

inline const std::vector<CachingEntity> &**children**() const

Return list of caching entities in this component.

inline const std::size_t **size**() const

Return number of caching entities in this component.

The complete class documentation generated by Doxygen can be found here.

7 Known Issues

There are a few known issues that either just haven't been implemented so far or need further clarification:

- Adaptation in spatial dimension three and for parallel computations
- Topology change of interface (e.g., merging)

The re-meshing feature is not (yet) supported in spatial dimension three because the removal of a vertex is not offered by the underlying CGAL triangulation class. In fact, it could appear that the region formed by its adjacent tetrahedrons is an instance of the untetrahedralizable Schönhardt's polyhedron. In this case, the removal of the vertex might be impossible without rebuilding the whole triangulation. The Dune-MMesh grid implementation supports solvers that are parallelized with MPI, but the adaptation is not yet generalized to the parallel case. For parallelization, one can also rely on multi-threading features like the threaded Galerkin operator in Dune-Fem.

We do not handle topology changes of interface (e.g., merging of two interfaces) yet. A simple intersection algorithm for a tip crossing a second interface has been implemented, but other kinds of interaction of two interfaces need further clarification on how to handle this interaction.

8 Acknowledgements

Dune-MMesh has been developed by Samuel Burbulla during his PhD supervised by Christian Rohde at IANS at the University of Stuttgart. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 327154368 – SFB 1313.

We gratefully thank Andreas Dedner for all his support, in particular in providing the Python bindings. Thanks to all contributors that improved Dune-MMesh via the GitLab repository, especially Timo Koch.

9 Contribute

If you want to contribute, you can do this using our GitLab or GitHub.

References

[CMR+18] C. Chalons, J. Magiera, C. Rohde, M. Wiebe. A Finite-Volume Tracking Scheme for Two-Phase Compressible Flow. Theory, Numerics and Applications of Hyperbolic Problems I, pp. 309–322, 2018.

[GBK20] J. Gerstenberger, S. Burbulla, D. Kröner. Discontinuous Galerkin method for incompressible two-phase flows. Finite Volumes for Complex Applications IX - Methods, Theoretical Aspects, Examples, pp. 675–683, 2020.

[BBD+21] P. Bastian, M. Blatt, A. Dedner, N.-A. Dreier, C. Engwer, R. Fritze, C. Gräser, C. Grüninger, D. Kempf, R. Klöfkorn, M. Ohlberger, O. Sander. The DUNE framework: Basic concepts and recent developments. Computers & Mathematics with Applications 81, 2021, pp. 75-112.

[TCP20] The CGAL Project. CGAL User and Reference Manual. CGAL Editorial Board, 5.2 edition, 2020.

[DNK+20] A. Dedner, M. Nolte, and R. Klöfkorn. Python Bindings for the DUNE-FEM module. Zenodoo, 2020, DOI 10.5281/zenodo.3706994.