Exercises for the introduction to the Grid Interface

Exercise 1 Iterating over a grid

First, you should start a fresh terminal and switch to the working directory of this exercise:

```
[user@dune-vm ~]$ cd dune-course
[user@dune-vm dune-course]$ cd release-build
[user@dune-vm release-build]$ cd dune-pdelab-tutorials
[user@dune-vm dune-pdelab-tutorials]$ cd gridinterface
[user@dune-vm gridinterface]$ cd exercise
[user@dune-vm exercise]$ cd task
```

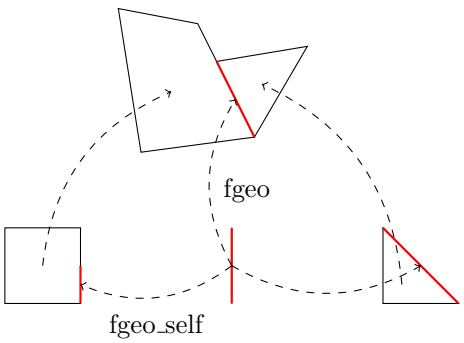
This is the build directory, so typing `make` will build three executables. To switch to the source directory, where the actual `.cc` files are located, type `cd src_dir`. To switch back to the build directory, type `cd ...`

Open the file `grid-exercise1.cc` in a text editor. It is an example code that creates a structured grid (using the DUNE class `YaspGrid`). The printgrid function then visualizes the grid as a png file with some useful information, like global and local indices, boundary intersections and such. You can have a look at the visualization after a succesful run of the executable `grid-exercise1` with:

```
[user@dune-vm task]$ xdg-open printgrid_0.png
```

By default, a 4x4 grid is generated. You can change this to some other number, recompile and rerun the executable and have a look at the new grid visualization.

The code then iterates over all elements of this grid and over all intersections of each element. The code is meant to print some information about the grid cells and the intersections (but it does not yet). The file is intermingled by suggestions what to print. You are invited to follow these suggestions or to try any of the member functions you learned about in the lectures.



Exercise 2 Integrating a function

Write a code that integrates a function using a quadrature formula of given order using the code in the file `integration.cc`. Dune provides you with quadrature formulas for many geometry types and integration orders with the following simple mechanism:

```
#include<dune/pdelab/common/quadraturerules.hh>

...
  auto rule = Dune::PDELab::quadratureRule(globalgeo,order);
  for (const auto& qp : rule)
  { ... }
...
```

You can now call the methods `weight()` and `position()` on the quadrature point object `qp`.

Exercise 3 The Finite Volume Method for the Transport Equation

The partial differential equation considered in this exercise is the *linear transport equation*,

$$\begin{aligned} \partial_t c(x,t) + \nabla \cdot \big(\mathbf{u}(x)\, c(x,t)\big) &= 0 && \text{in } \Omega, \\ c(x,t) &= c_{\text{in}}(x,t) && \text{on } \Gamma_{\text{in}}. \end{aligned} \tag{1}$$

The unknown solution is denoted by $c(x,t)$ and the velocity field by $\mathbf{u}(x)$. The domain Ω is some open subset of $\mathbb{R}^d$. For this exercise, we choose $d = 2$ where intersections are 1-D edges. For $d = 3$, intersections would be 2-D faces. The inflow boundary Γ_{in} is the set of points x on the boundary of Ω for which the velocity vector $\mathbf{u}(x)$ points inwards.

We want to numerically solve this equation by a cell-centered finite volume scheme. We discretize the domain Ω by a triangulation T_h and approximate the solution c by a function c_h that is constant on each cell $E \in T_h$. We denote the value of c_h on a cell E by c_E.

Using the explicit Euler time discretization, the scheme can be written as

$$c_E(t_{k+1}) = c_E(t_k) - \frac{t_{k+1} - t_k}{|E|} \sum_{e \subset \partial E} |e|\; c^e(t_k)\; \mathbf{n}_E^e \cdot \mathbf{u}^e. \tag{2}$$

The notation is as follows: $|E|$ is the area of the cell E, the sum runs over all intersections e of E with either the boundary or a neighboring cell, $|e|$ is the length of edge e, $\mathbf{n}_E^e$ is the outer normal of edge e and $\mathbf{u}^e$ is the velocity at the center of edge e. Finally, c^e denotes the upwind concentration. If $\mathbf{n}_E^e \cdot \mathbf{u}^e > 0$, this is c_E. Otherwise it is either the concentration in the neighboring cell or given by the boundary condition c_{in}, depending on the location of e.

Run the incomplete finite volume program and familiarize yourself with the code in the files ***`finitevolume.cc`*** *and* ***`fv.hh`****.*

Take a look at the code. You will recognize some of the methods and classes from the first part of the grid exercise. Don't bother with grid creation and the VTK output for now, that will be explained in the second part of the grid tutorial.

The program will already compile and run as-is, but it does not yet update the solution (i.e. the solution does not change over time). You can run the program `finitevolume` and take a look at its current output. The solution is written in the VTK data format, which you can visualize using ParaView. The data consists of one `.vtk` file per time step and an additional file `concentration.pvd` that contains information about the whole time series. Open that file by calling

```
[user@dune-vm task]$ paraview --data=concentration.pvd &
```

to load the complete output of the program. ParaView will be explained in more detail in the second part of the grid tutorial, but for now just click the "Apply" button in the middle left of the screen to load the data and the triangular "Play" button at the top to start playing the solution. You won't see anything move right now, but once you've finished the next part of the exercise, the red blob in the bottom left part of the solution should start moving to the top right, blurring out in the process.

Complete the implementation in the files ***`finitevolume.cc`*** *and* ***`fv.hh`****.*

An implementation of the scheme at hand has to store the values of the concentration on each cell for the current time step. In the example code, a simple `std::vector<double>` is used for this purpose, see the type `ScalarField` in `finitevolume.cc`. We use a mapper to get a consecutive numbering of the cells of the grid, even for hybrid grids that contain cells with different geometry types. If the variable `e` holds some entity of codimension 0, the concentration value in this entity is `c[mapper.index(e)]`.

At each time step, an update to the vector of concentrations has to be computed. This is done by the `update_concentration()` member function of the class `FiniteVolume` in the file `fv.hh`. This function iterates over all cells of the grid in order to compute an update for each cell. Your task is to implement the computation of the update `update[cell_index]`. To this end, the code has to iterate over all intersections of the current cell. For each intersection, the flux $|e|\mathbf{n}_E^e \cdot \mathbf{u}^e/|E|$ is to be calculated. Depending on the sign of this flux, the upwind decision can be made.

Further tasks

Once you have managed to implement the Finite Volume scheme, there are a few additional things you can try:

- Try varying the velocity field (maybe make it dependent on the coordinate) or add an inflow boundary.
- Increase the time step size. How far can you go before the scheme breaks down? It might be a good idea to pass the time step size as a command line parameter to the program in this case. You can convert a command line parameter to a `double` value like this:

```
#include <cstdlib>

int main(int argc, char** argv)
{
  double dt = atof(argv[1]);
  ...
}
```

- Right now, the program only works for 2D calculations. While you can replace the value for `dim` in the `main()` function, the program will not work correctly afterwards. Find what needs to be changed and fix it. Afterwards, try running your program in 3D by setting `dim = 3` and recompiling. Before running the program, you might want to reduce the grid size, which is given by the variable `N`.