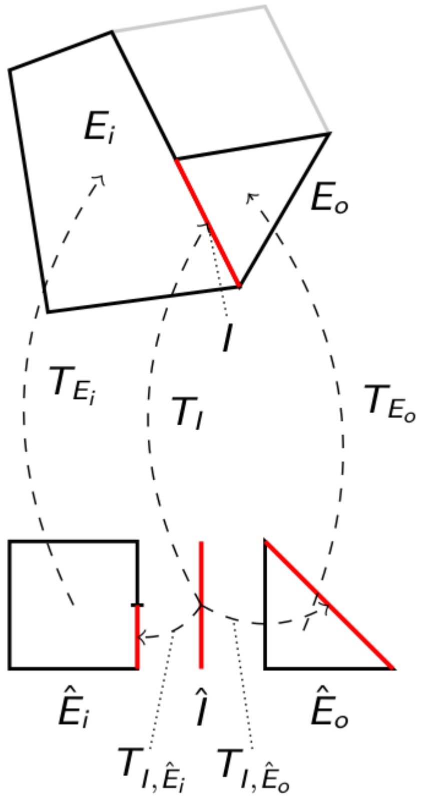
Intersections

Intersections

- Grids may be non conforming.
- Entities can intersect with neighbours and boundary.
- Represented by Intersection objects.
- Intersections hold topological and geometrical information.
- Intersections depend on the view.
- **Note:** Intersections are always of codimension 1!



Intersection Interface

- Is this an intersection with the domain boundary?

```
bool b = intersection.boundary();
```

- Is there an entity on the outside of the intersection?

```
bool b = intersection.neighbor();
```

- Get the cell on the inside

```
auto inside_cell = intersection.inside();
```

- Get the cell on the outside

```
// Do this only if intersection.neighbor() == true
auto outside_cell = intersection.outside();
```

Intersection: Geometries

- Get mapping from intersection reference element to global coordinates

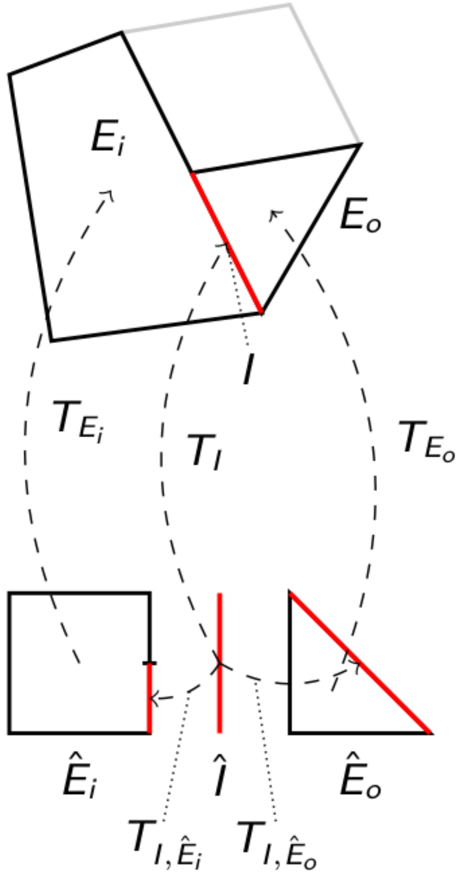
```
auto world_geo = intersection.geometry();
```

- Get mapping from intersection reference element to reference element of inside cell

```
auto inside_geo = intersection.geometryInInside();
```

- Get mapping from intersection reference element to reference element of outside cell

```
auto outside_geo = intersection.geometryInOutside();
```



Intersection: Normals

- Get unit outer normal for local coordinate.

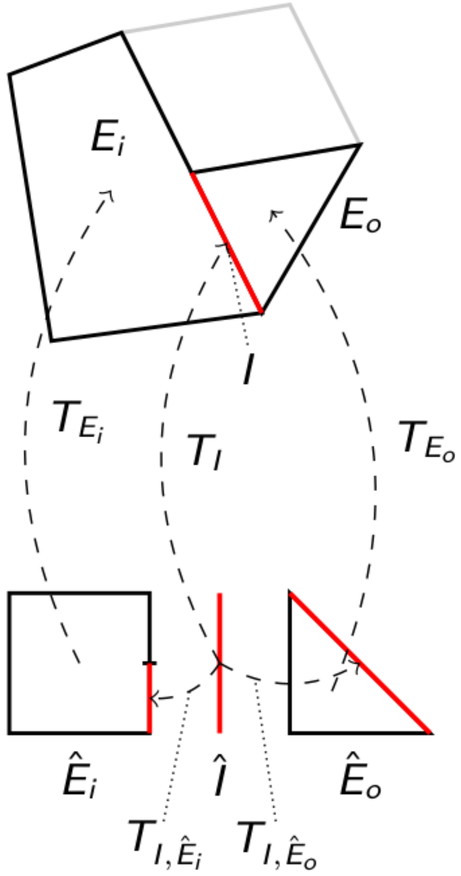
```
auto unit_outer_normal = intersection.unitOuterNormal(x_local);
```

- Get unit outer normal for center of intersection (good for affine geometries).

```
auto unit_outer_normal = intersection.centerUnitOuterNormal();
```

- Get unit outer normal scaled with integration element (convenient for numerical quadrature).

```
auto integration_outer_normal
= intersection.integrationOuterNormal(x_local);
```



Iterating over intersections

Example

In order to iterate over the intersections of a given grid cell with respect to some `GridView`, use a range-based for loop with the argument `intersections(gv,cell)`.

The following code iterates over all cells in a GridView and over all intersections of each cell:

```
for (const auto& cell : elements(gv)) {
  for (const auto& is : intersections(gv, cell)) {
    if (is.boundary()) {
      // handle potential Neumann boundary
    }
    if (is.neighbor()) {
      // code for Discontinuous Galerkin or Finite Volume
    }
  }
}
```

Sequential finite volume solver

Elementwise divergence of a vector field:

$$\int_T \nabla \cdot f(x)\,\mathrm{d}x = \int_{\partial T} f(x) \cdot \mathbf{n}_{\partial T}\,\mathrm{d}s$$

Consider the *first-order linear PDE* $\partial_t u + \nabla \cdot (vu) = 0$ *with given vector field* $v(x)$ *and unknown solution* $u(x,t)$. *The structure-explicit cell-centered finite volume method reads*

{(T,T')\in I(T)} \phi(v\cdot\mathbf{n}T, \bar{u}_T^k, \bar{u}{T'}^k) |I(T,T')| u¯T k+1 = u¯T k − |T|Δt (T,T')∈I(T)∑ ϕ(v · nT , u¯T k, u¯T'k)|I(T , T')| with the numerical flux function ϕ

chosen as upwind flux here.

> keep this in mind for exercise 3

Attaching Data to the Grid

Attaching Data to the Grid

For computations we need to associate data with grid entities:

- spatially varying parameters
- entries in the solution vector or the stiffness matrix
- polynomial degree for p-adaptivity
- status information during assembling
- ...

Attaching Data to the Grid

For computations we need to associate data with grid entities:

- spatially varying parameters
- entries in the solution vector or the stiffness matrix
- polynomial degree for p-adaptivity
- status information during assembling
- ...

... meaning we want to

- associate data with subsets of entities
- subsets could be "vertices of level l", "faces of leaf elements" ...
- data should be stored in arrays for efficiency
- associate index/id with each entity

Indices and Ids

Index Set: Provides a map $m : E \to \mathbb{N}_0$, where E is a subset of the entities of a grid view.

We define the subsets E_g^c of a grid view

$$E_g^c = \{e \in E | e \text{ has codimension } c \text{ and geometry type } g\}.$$

- unique within the subsets E_g^c.
- consecutive and zero-starting within the subsets E_g^c.
- distinct leaf and a level index.

Indices and Ids

Index Set: Provides a map $m : E \to \mathbb{N}_0$, where E is a subset of the entities of a grid view.

We define the subsets E_g^c of a grid view

$$E_g^c = \{e \in E | e \text{ has codimension } c \text{ and geometry type } g\}.$$

- unique within the subsets E_g^c.
- consecutive and zero-starting within the subsets E_g^c.
- distinct leaf and a level index.

Id Set: provides a map $m : E \to \mathbb{I}$, where $\mathbb{I}$ is a discrete set of ids.

- unique within E.
- ids need not to be consecutive nor positive.
- persistent with respect to grid modifications.

Example: Store the lengths of all edges

The following example demonstrates how to

- query an index set for the number of contained entities of a certain codimension (so that we can allocate a vector of correct size).
- obtain the index of a grid entity from an index set and use it to store associated data.

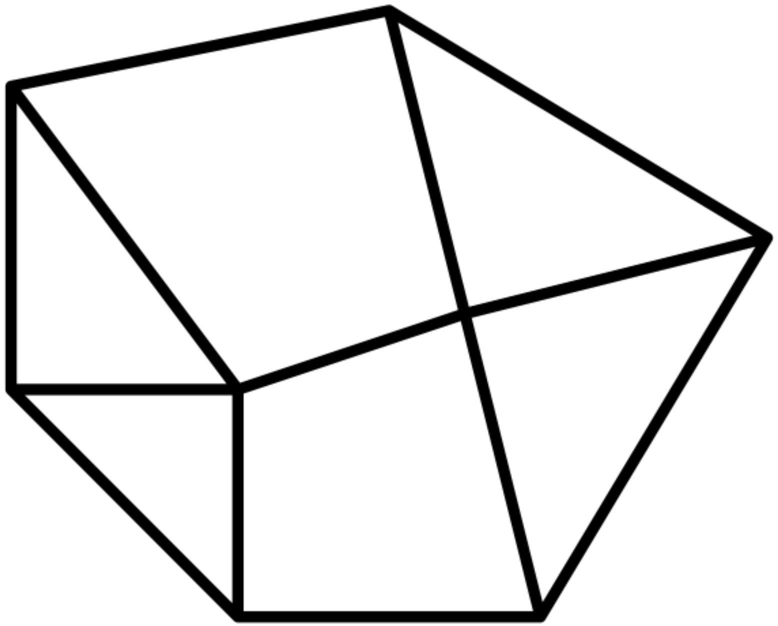
```
// Get the IndexSet associated to a GridView gv
const auto& index_set = gv.indexSet();

// Create a vector with one entry for each edge
auto edge_lengths = std::vector<double>(index_set.size(1));

// Loop over all edges and store their length
for (const auto& edge : edges(gv))
  edge_lengths[ index_set.index(edge) ] = edge.geometry().volume();
```

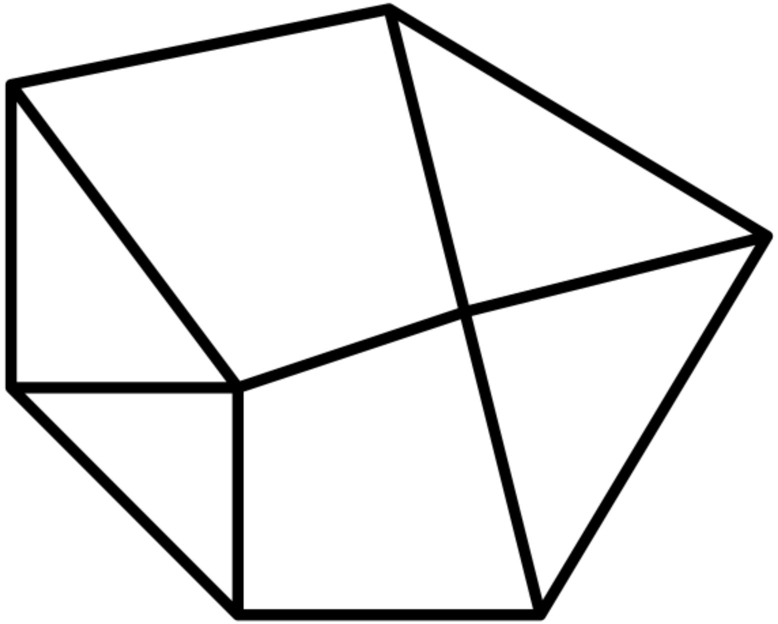
Example: 2D Multi-Element Grid – Indices

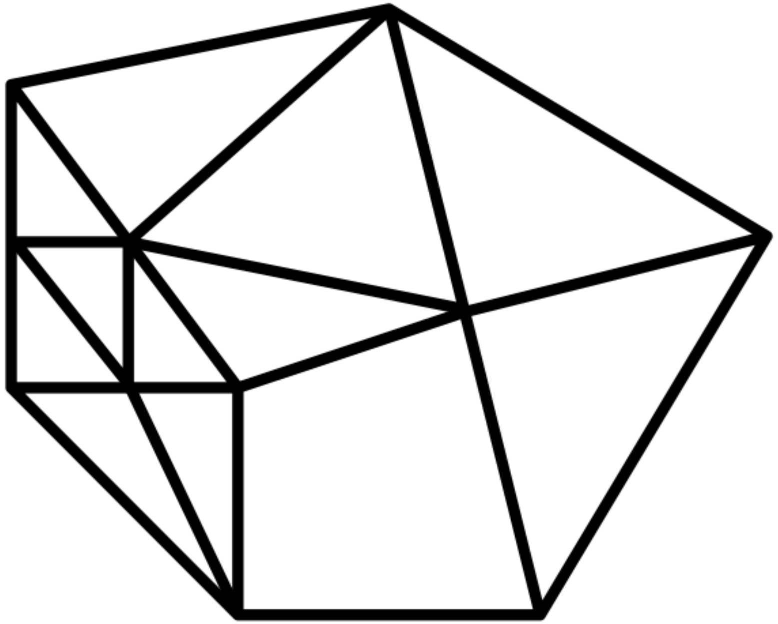
Locally refined grid:



Example: 2D Multi-Element Grid – Indices

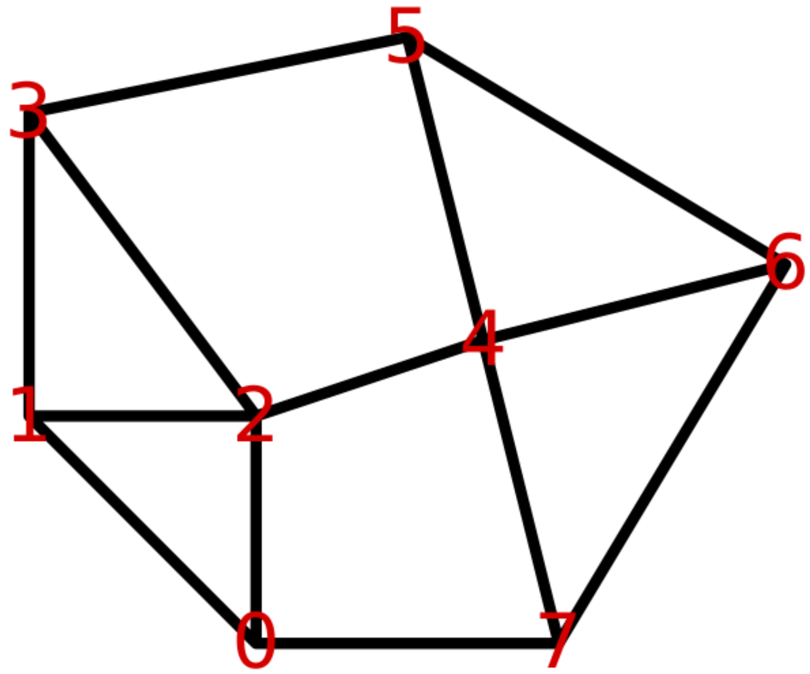
Locally refined grid:

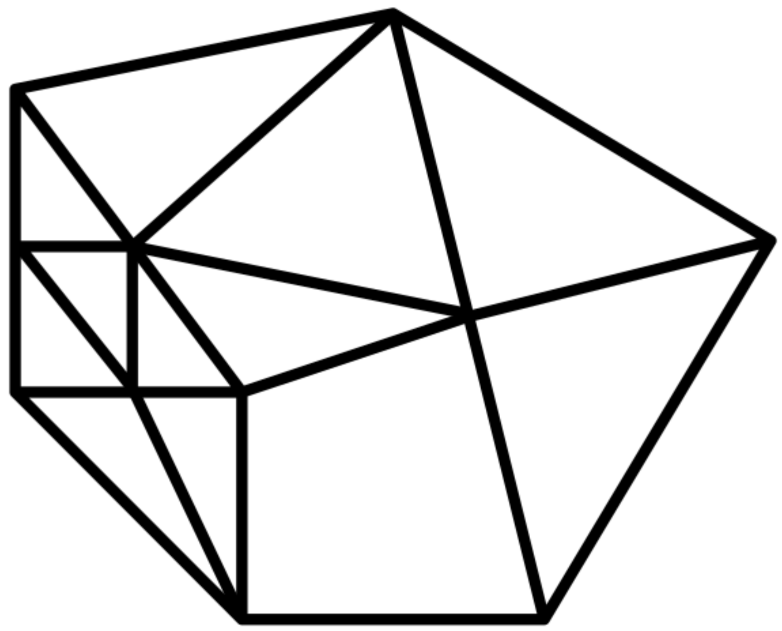




Example: 2D Multi-Element Grid – Indices

Locally refined grid: **Indices**

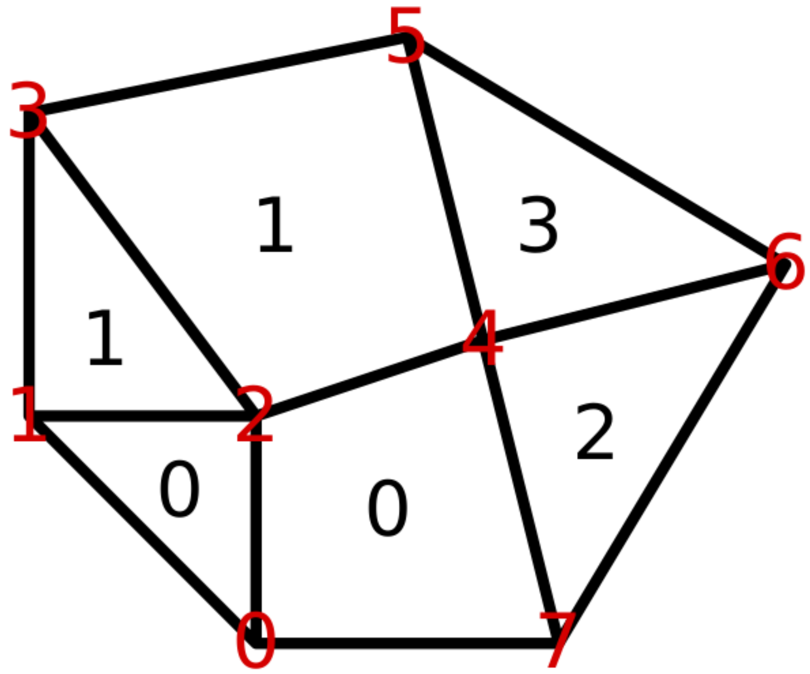


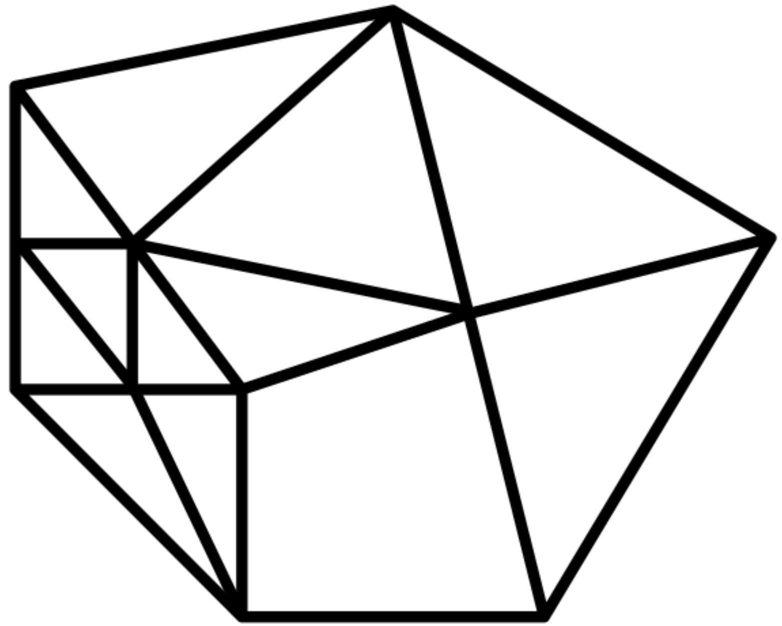


Consecutive index for vertices

Example: 2D Multi-Element Grid – Indices

Locally refined grid: **Indices**

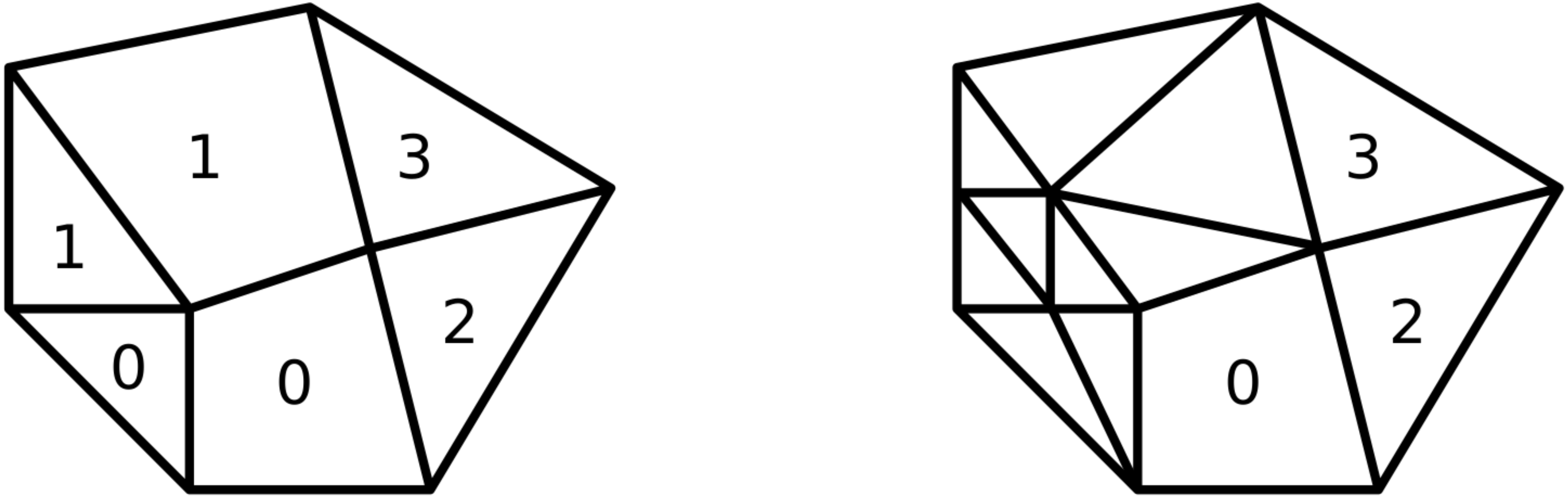




… and cells

Example: 2D Multi-Element Grid – Indices

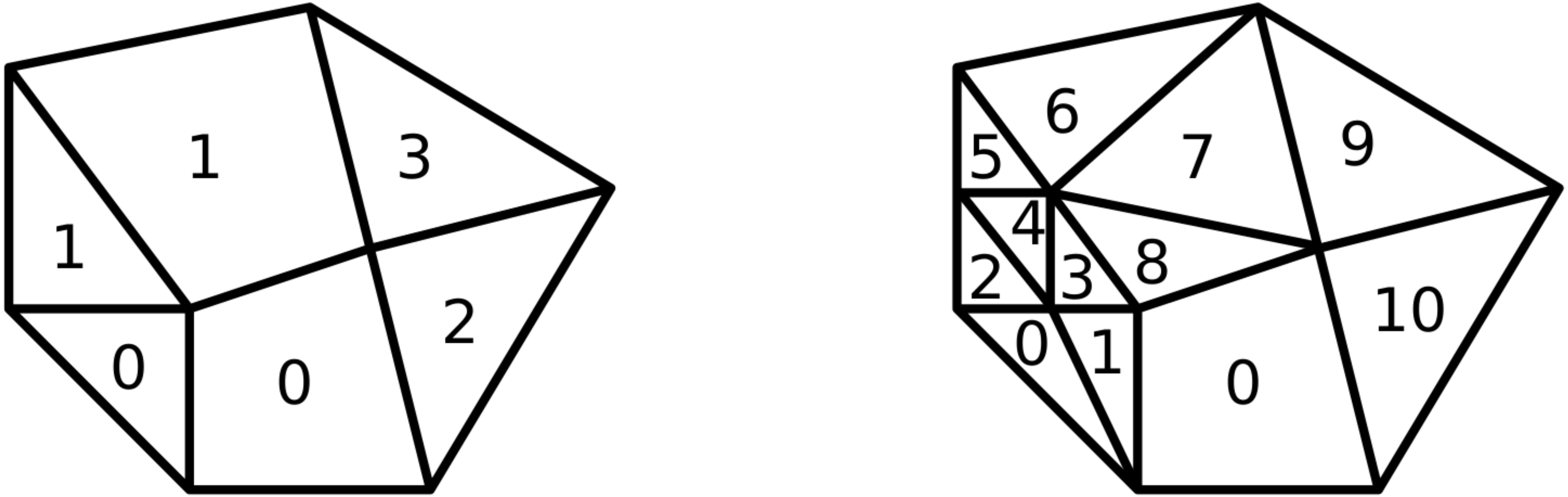
Locally refined grid: **Indices**



Old cell indices on coarse grid level

Example: 2D Multi-Element Grid – Indices

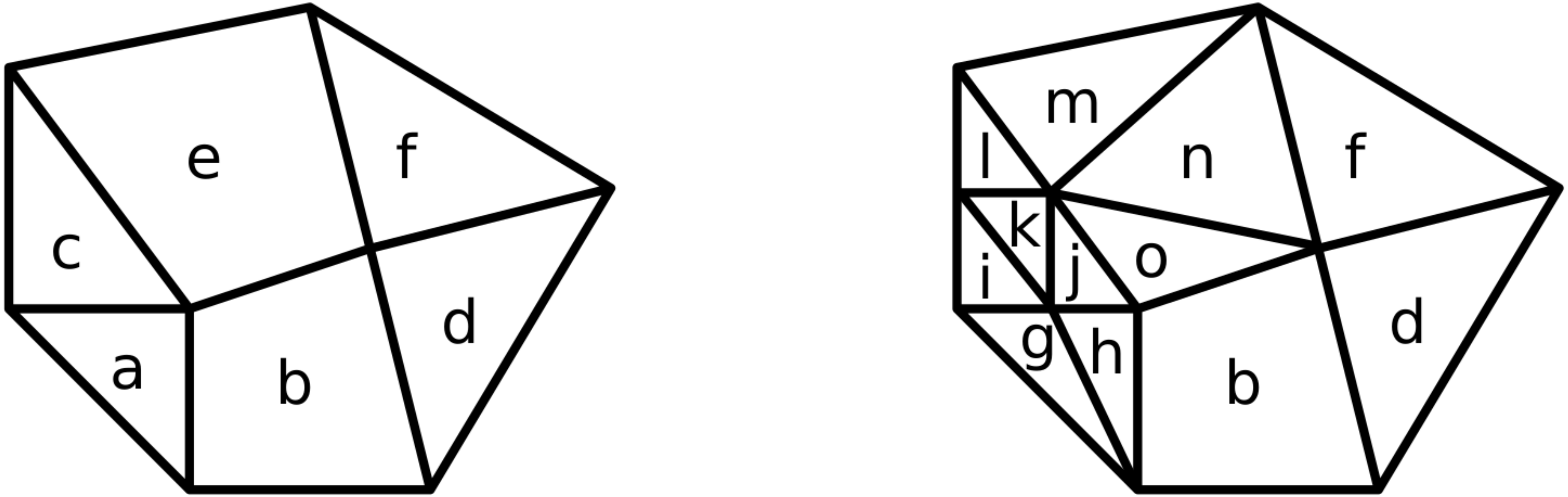
Locally refined grid: **Indices**



Consecutive cell indices on coarse and refined grid

Example: 2D Multi-Element Grid – Indices

Locally refined grid: **Ids**



Persistent Ids on coarse and refined grid

Mapper

Mappers extend the functionality of **Index Sets**.

- associate data with an arbitrary subsets $E' \subseteq E$ of the entities E of a grid.
- the data $D(E')$ associated with E' is stored in an array.
- map from the consecutive, zero-starting index $I_{E'} = \{0, \ldots, |E'| - 1\}$ to the data set $D(E')$.

Mappers can be easily implemented upon the Index Sets and Id Sets.

Mapper

Mappers extend the functionality of **Index Sets**.

- associate data with an arbitrary subsets $E' \subseteq E$ of the entities E of a grid.
- the data $D(E')$ associated with E' is stored in an array.
- map from the consecutive, zero-starting index $I_{E'} = \{0, \ldots, |E'| - 1\}$ to the data set $D(E')$.

Mappers can be easily implemented upon the Index Sets and Id Sets.

Example

You will be using the

```
Dune::MultipleCodimMultipleGeomTypeMapper<GridView>
```

```cpp
#include <dune/grid/common/mcmgmapper.hh>
...
using GridView = SomeGrid::LeafGridView;

// Layout description
Dune::MCMGLayout layout = [](Dune::GeometryType gt, int griddim) {
  return gt.dim() == griddim;
};

// mapper for elements (codim==0) on leaf
using Mapper = Dune::MultipleCodimMultipleGeomTypeMapper<GridView>;
Mapper mapper{gridview, layout};

// iterate over the leaf
for (const auto& entity : elements(gridview))
{
  int index = mapper.index(entity);
  // iterate over all intersections of this cell
  for (const auto& i : intersections(gridview, entity))
  {
    // neighbor intersection
    if (i.neighbor()) {
      int nindex = mapper.index(i.outside());
      sparsityPattern[index].insert(nindex);
    }
  }
}
```

Input and Output

Input and Output

- We need to provide a grid either by geometric description or loaded from file
- Powerful grid generators exist, exporting grids in multiple formats
- For the visualization of grid we also need to write the geometry and maybe attached data in a common file format

In this section:

1. Grid factories
2. Structured grid generation
3. Reading grids from file
4. Writing output to a file

Grid Factories

- A grid is a combination of elements, described by their vertex coordinates + a connectivity of the element nodes.
- All Dune grids provide a **Grid Factory** that allows to create a grid by these two information

```
Dune::GridFactor<SomeGrid> factory;
```

- The `factory` then provides two main methods:

```
void insertVertex (const FieldVector<ct,dimworld>& pos);
void insertElement (const GeometryType& type, const std::vector<unsigned int>& vertices)
```

- Construct the grid after inserting vertices and element connectivity:

```
std::unique_ptr<SomeGrid> createGrid ();
```

Grid Factories

Example

```
Dune::GridFactory<SomeGrid> factory;

factory.insertVertex({0, 0, 0});
factory.insertVertex({1, 0, 0});
factory.insertVertex({0, 1, 0});
factory.insertVertex({1, 1, 0});
factory.insertVertex({0, 0, 1});
factory.insertVertex({1, 0, 1});
factory.insertVertex({0, 1, 1});
factory.insertVertex({1, 1, 1});

namespace GT = Dune::GeometryTypes;
factory.insertElement(GT::tetrahedron, {0, 1, 3, 7});  // insertion-index 0
factory.insertElement(GT::tetrahedron, {0, 5, 1, 7});  // insertion-index 1
factory.insertElement(GT::tetrahedron, {0, 4, 5, 7});  // insertion-index 2
factory.insertElement(GT::tetrahedron, {0, 6, 4, 7});  // insertion-index 3
factory.insertElement(GT::tetrahedron, {0, 2, 6, 7});  // insertion-index 4
factory.insertElement(GT::tetrahedron, {0, 3, 2, 7});  // insertion-index 5

std::unique_ptr<SomeGrid> gridPtr = factory.createGrid();
```

Grid Factories

- Identify elements in the constructed grid, by **insertion-index**

```
unsigned int insertionIndex (const typename Codim<0>::Entity& entity);
```

- **Note:** The insertion-index might be different from the entity index in the Index Set

Example

```
auto const& indexSet = gridPtr->leafIndexSet();
for (auto const& cell : elements(*gridPtr))
{
  auto index = indexSet.index(cell);
  auto insertion_index = factory.insertionIndex(cell);

  // in general: index != insertion_index
}
```

Structured Grid Generation

- Some standard Grid Factories exist for rectangular domains

```
#include <dune/grid/utility/structuredgridfactory.hh>
...
Dune::StructuredGridFactory<SomeGrid>
```

- These structured factories provide static methods to construct cube/simplex grids of a prescribed domain:

```
std::unique_ptr<SomeGrid> createCubeGrid (const FieldVector<ct,dimworld>& lowerLeft,
                                          const FieldVector<ct,dimworld>& upperRight,
                                          const std::array<unsigned int,dim>& elements);

std::unique_ptr<SomeGrid> createSimplexGrid (const FieldVector<ct,dimworld>& lowerLeft,
                                             const FieldVector<ct,dimworld>& upperRight,
                                             const std::array<unsigned int,dim>& elements);
```

Structured Grid Generation

Example

A domain $\Omega = [0, 2] \times [0, 4]$ with 100 elements in x direction and 200 elements in y direction

```
using GridType = Dune::UGGrid<2>;
using Factory = Dune::StructuredGridFactory<GridType>;

auto gridPtr = Factory::createCubeGrid({0.0,0.0}, {2.0,4.0}, {100u,200u});
```

But also simplex elements can be used:

```
using GridType = Dune::UGGrid<2>;
using Factory = Dune::StructuredGridFactory<GridType>;

auto gridPtr = Factory::createSimplexGrid({0.0,0.0}, {2.0,4.0}, {100u,200u});
```

Reading Grids from File

External tools like **GMsh** or **ParaView** or CAD tools export their grids in a specific file format. In order to create a Dune Grid from these file, the content needs to be parsed and a `GridFactory` can be filled with the vertices and element connectivity.

For some file formats this is already implemented in `dune-grid`:

- GMsh (2.x)
- StarCD
- DGF
- AmiraMesh
- AlbertaGrid

With external modules also other formats are supported, e.g., GMsh (4.x) with `dune-gmsh4` and VTK with `dune-vtk`.

Reading Grids from File

Example

Reading a grid from a GMsh 2.x file

```
#include <dune/grid/io/file/gmshreader>
...
using Reader = Dune::GmshReader<SomeGrid>;
std::unique_ptr<SomeGrid> gridPtr = Reader::read(<filename>);
```

Reading Grids from File

Example

Reading a grid from a GMsh 2.x file

```
#include <dune/grid/io/file/gmshreader>
...
using Reader = Dune::GmshReader<SomeGrid>;
std::unique_ptr<SomeGrid> gridPtr = Reader::read(<filename>);
```

Some readers provide interfaces to also read data, e.g. with the `GMshReader`

```
Dune::GridFactory<SomeGrid> factory;
std::vector<int> boundaryTags;
std::vector<int> elementTags;
Reader::read(factory, <filename>, boundaryTags, elementTags);

std::unique_ptr<SomeGrid> gridPtr = factory.createGrid();
```

Writing Output to a File

For the visualization of simulation results, you need to write the grid and attached data to a file. A common output format is VTK.

Dune provides a `VTKWriter` and also sequence writers for animation files:

```
#include <dune/grid/io/file/vtk.hh>
...
Dune::VTKWriter<GridView>;
Dune::VTKSequenceWriter<GridView>;
```

Writing Output to a File

For the visualization of simulation results, you need to write the grid and attached data to a file. A common output format is VTK.

Dune provides a `VTKWriter` and also sequence writers for animation files:

```
#include <dune/grid/io/file/vtk.hh>
...
Dune::VTKWriter<GridView>;
Dune::VTKSequenceWriter<GridView>;
```

A writer is constructed from a `GridView` and allows to attach data on the cells or on the vertices:

```
void addCellData (...);
void addVertexData (...);
```

where the argument is something that can be evaluated on the corresponding grid entities:

- A container of values with contiguous indexing of the entity values
- A localizable function (see later)

Writing Output to a File

Example

Writing a vector of values attached to the grid cells to a file

```
Dune::VTKWriter writer{gv, Dune::VTK::nonconforming};  // write a non-conforming grid

auto u = std::vector<double>(gv.size(0));
writer.addCellData(u, "c", 1);                          // vector contains 1 value per cell

writer.write("concentration", "output", "");            // write into the directory output/
```

Writing Output to a File

Example

Writing a vector of values attached to the grid cells to a file

```
Dune::VTKWriter writer{gv, Dune::VTK::nonconforming};  // write a non-conforming grid

auto u = std::vector<double>(gv.size(0));
writer.addCellData(u, "c", 1);                         // vector contains 1 value per cell

writer.write("concentration", "output", "");           // write into the directory output/
```

A `VTKSequenceWriter` is constructed with the output location:

```
Dune::VTKSequenceWriter writer{gv, "concentration", "output", "", Dune::VTK::nonconforming};
writer.addCellData(u, "c", 1);
writer.write(time);
```

Exercise 3

Exercise 3

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

$$\partial_t u(x,t) + \nabla \cdot (v(x)u(x,t)) = 0 \quad \text{in } \Omega$$
$$u(x,t) = u_{in}(x,t) \quad \text{on } \Gamma_{in}$$

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

We want to numerically solve this equation by a cell-centered finite volume scheme. We discretize the domain Ω by a triangulation $\mathcal{T}_h$ and approximate the solution u by a function u_h that is constant on each cell $T \in \mathcal{T}_h$. We denote the value of u_h on a cell T by u_T.

Exercise 3

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

$$u_T(t_{k+1}) = u_T(t_k) - \frac{t_{k+1} - t_k}{|T|} \sum_{e \in \partial T} |e| u^e(t_k) n_T^e \cdot v^e \,.$$

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

Exercise 3

For the concrete setup, consider:

- Grid $\Omega = [0,1] \times [0,1]$ with 100 elements in each direction
- $u_T(t_0) = u_0(center(T))$ with

$$u_0(x) = \begin{cases} 1 & \text{if } \|x - (0.15, 0.15)\|_\infty \leq 0.05 \\ 0 & \text{otherwise} \end{cases}$$

- Velocity $v = (1,1)$
- Boundary condition $u_{\text{in}} = 0$
- Time interval $t \in [0,1]$ with timestep size $t_{k+1} - t_k = 0.5 \cdot h$ with h the grid size.