The DUNE Grid Interface
An Introduction

Christian Engwer

Applied Mathematics, WWU Münster
Orleans-Ring 10, 48149 Münster

Part I

Dune Course: Design Principles

[...] *a modular toolbox for solving partial differential equations (PDEs) with grid-based methods* [...]

— `http://www.dune-project.org/`

Part I

Dune Course: Design Principles

[...] *a modular toolbox for solving partial differential equations (PDEs) with grid-based methods* [...]

— `http://www.dune-project.org/`

Contents

Design Principles

Flexibility: Seperation of data structures and algorithms.

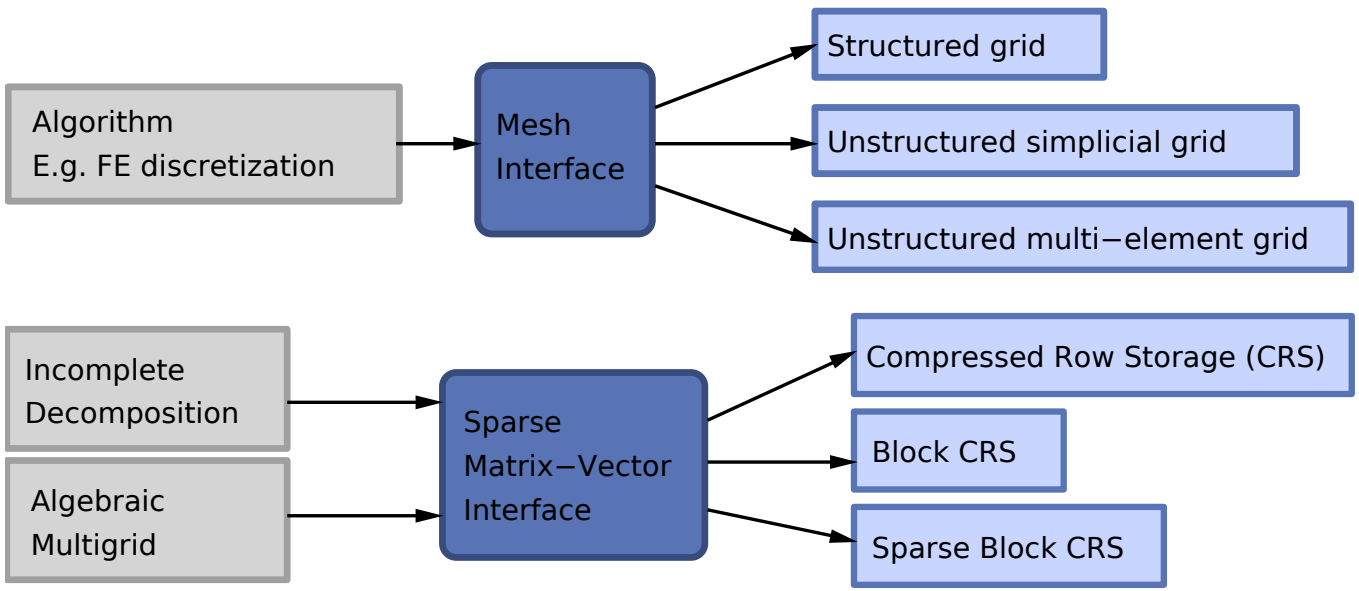
Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.

Flexibility

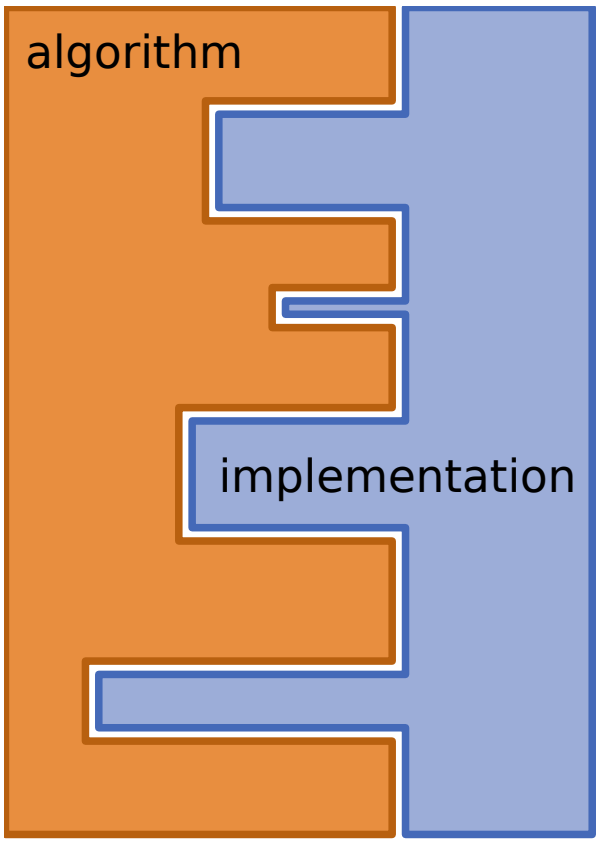
Seperate data structures and algorithms.

- The algorithm determins the data structure to operate on.
- Data structures are hidden under a common interface.
- Algorithms work only on that interface.
- Different implementations of the interface.



Efficiency

Implementation with generic programming techniques.

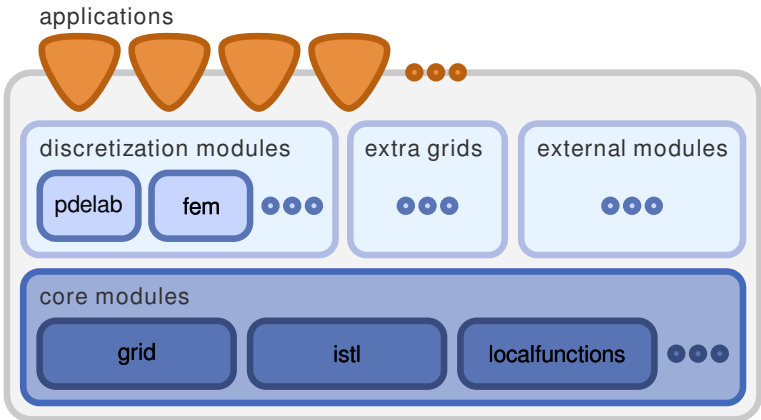


1. Static Polymorphism
   - Duck Typing (see STL)
   - Curiously Recurring Template Pattern (Barton and Nackman)
2. Grid Entity Ranges
   - Generic access to different data structures.
3. View Concept
   - Access to different partitions of one data set.

Contents

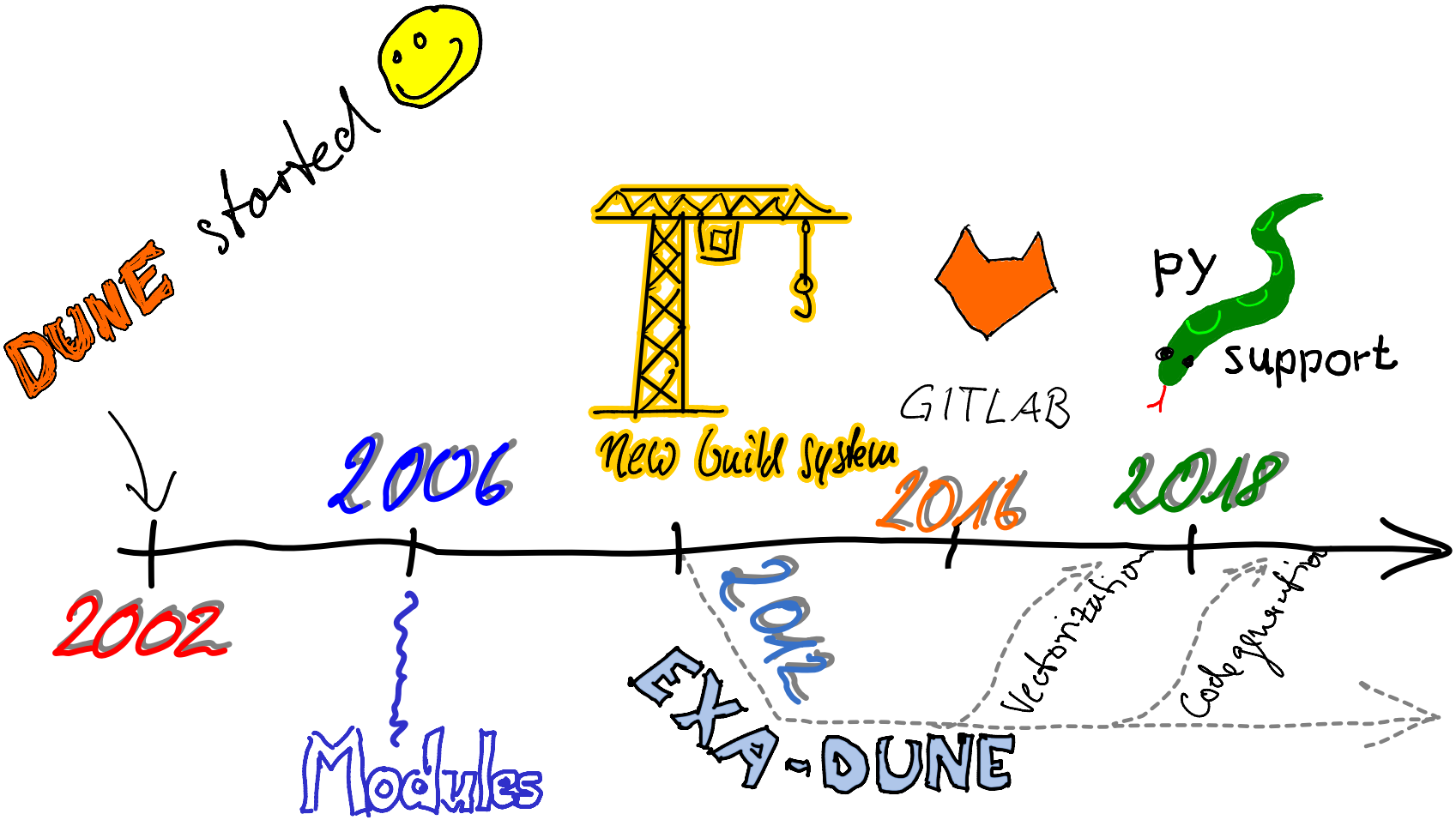
The DUNE Framework

- Modules
  - Code is split into separate modules.
  - Applications use only the modules they need.
  - Modules are sorted according to level of maturaty.
  - Everybody can provide her/his own modules.
- Portability
- Open Development Process
- Free Software Licence



[Bastian, Blatt, Dedner, Engwer, Klöfkorn, Kornhuber, Ohlberger, Sander 2008]

Some historic remarks



DUNE Release 2.6.0

dune-common: foundation classes, infrastructure

dune-geometry: geometric mappings, quadrature rules visualization

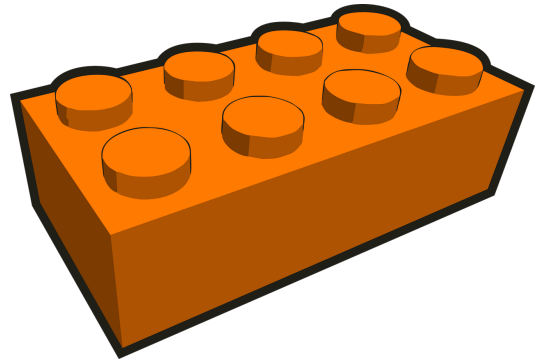
dune-grid: grid interface, visualization

dune-istl: *(Iterative Solver Template Library)*
generic sparse matrix/vector classes,
solvers (Krylov methods, AMG, etc.)

dune-localfunctions: generic interface for local finite element functions. Abstract definition following Ciarlet. Collection of different finite elements.

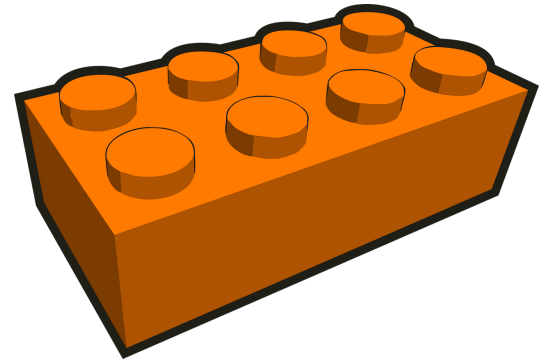
DUN(E)iverse

- modular structure
- write your own DUNE modules
- available under different licenses



- Discretization Modules
- Additional Grid Implementations
- Extension Modules

DUN(E)iverse

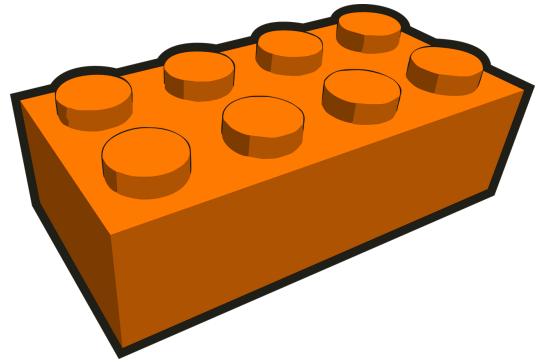


- modular structure
- write your own DUNE modules
- available under different licenses
- **Discretization Modules**

  dune-pdelab: Discretization module based on dune-localfunctions.
  dune-fem: Discretization module based on dune-localfunctions.
  dune-functions: A new initiative to provide unified interfaces for functions and function spaces.

- **Additional Grid Implementations**
- **Extension Modules**

DUN(E)iverse

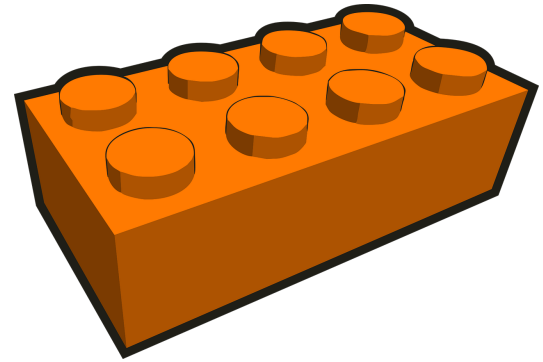


- modular structure
- write your own DUNE modules
- available under different licenses
- **Discretization Modules**

  dune-pdelab: Discretization module based on dune-localfunctions.
  dune-fem: Discretization module based on dune-localfunctions.
  dune-functions: A new initiative to provide unified interfaces for functions and function spaces.

- **Additional Grid Implementations**
- **Extension Modules**

DUN(E)iverse



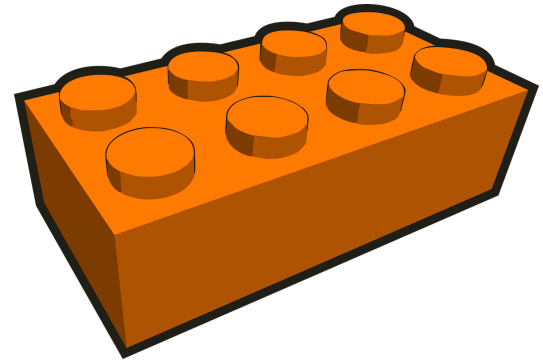
- modular structure
- write your own DUNE modules
- available under different licenses

- **Discretization Modules**
- **Additional Grid Implementations**

dune-grid-glue:	allows to compute overlapping and nonoverlapping couplings of Dune grids, as required for most domain decomposition algorithms.
dune-subgrid:	allows you to work on a subset of a given DUNE grid.
dune-foamgrid:	non-manifold grids of 1d or 2d entities in higher-dimensional world.
dune-prismgrid:	is a tensorgrid of a 2D simplex grid and a 1D grid.
dune-cornerpoint:	a cornerpoint mesh, compatible with the grid format of the ECLIPSE reservoir simulation software.
...	

- **Extension Modules**

DUN(E)iverse



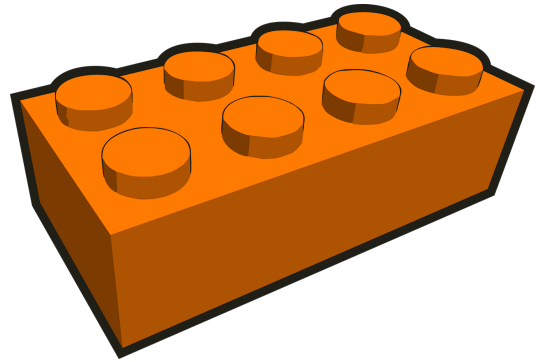
- modular structure
- write your own DUNE modules
- available under different licenses

- **Discretization Modules**
- **Additional Grid Implementations**
- **Extension Modules**

dune-python	python bindings for centrral DUNE components
dune-typetree	classes to organise types in trees
dune-dpg	construct optimal Discontinuous-Petrov-Galerkin test spaces
dune-tpmc	cut-cell construction using level-sets
...	

DUN(E)iverse

- modular structure
- write your own DUNE modules
- available under different licenses



- **Discretization Modules**
- **Additional Grid Implementations**
- **Extension Modules**

Goals: allow people to...

- get credit for their innvoations
- experiment without breaking the core
- develop at different speeds

A Package System

`dunecontrol`

- control of module-interplay
- suggestions & dependencies
- intergrates with cmake & git
- works with Linux, Mac and Mingw

Source: gnome

Note: Dependencies should form a DAG

`dunecontrol cmake`
: configure packages via cmake, include necessary path information

`dunecontrol make`
: build packages in correct order

`...` works without `make install`

A Package System

`dunecontrol`

- control of module-interplay
- suggestions & dependencies
- intergrates with cmake & git
- works with Linux, Mac and Mingw

Source: gnome

Note: Dependencies should form a DAG

`dunecontrol cmake`
configure packages via cmake, include necessary path information

`dunecontrol make`
build packages in correct order

`...` works without `make install`

A Package System

`dunecontrol`

- control of module-interplay
- suggestions & dependencies
- intergrates with cmake & git
- works with Linux, Mac and Mingw

Source: gnome

Note: Dependencies should form a DAG

`dunecontrol cmake`
: configure packages via cmake, include necessary path information

`dunecontrol make`
: build packages in correct order

`...` works without `make install`

Part II

Dune Course: Grid Module

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.

— *Donald E. Knuth*

Why Grids?

Weak formulation of boundary value problem:

$$\text{Find } u \in U \text{ s.t.} \qquad a(u, v) = l(v) \quad \forall\, v \in V.$$

$a(u, v)$ and $l(v)$ are (bi)linear forms, e.g.

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx,$$

with spatial domain $\Omega \subset \mathbb{R}^d$.

Grids are necessary for at least three reasons:

1. Piecewise description of the complicated domain Ω
2. Piecewise approximation of functions (by polynomials)
3. Piecewise computation of integrals (by numerical quadrature)

Why Grids?

Weak formulation of boundary value problem:

$$\text{Find } u \in U \text{ s.t.} \qquad a(u, v) = l(v) \quad \forall\, v \in V.$$

$a(u, v)$ and $l(v)$ are (bi)linear forms, e.g.

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx,$$

with spatial domain $\Omega \subset \mathbb{R}^d$.

Grids are necessary for at least three reasons:

1. Piecewise description of the complicated domain Ω
2. Piecewise approximation of functions (by polynomials)
3. Piecewise computation of integrals (by numerical quadrature)

Numerical Quadrature

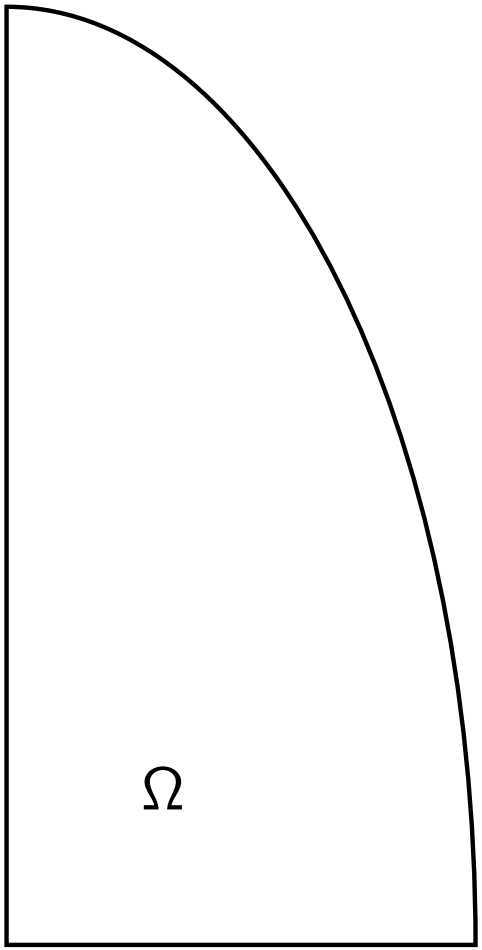
- Approximate integral by a weighted sum of function evaluations at sampling points:

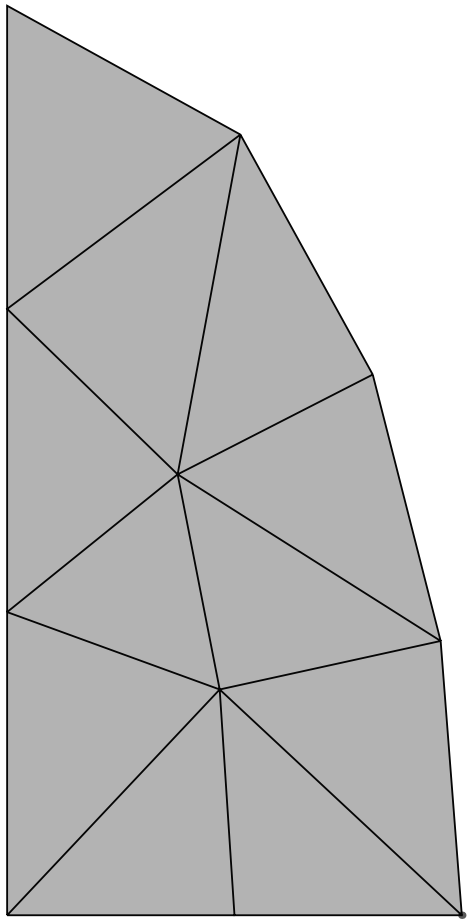
$$\int_{\hat{\Omega}} f(x)\,dx \approx \sum_{i=1}^{N} w_i f(x_i)$$

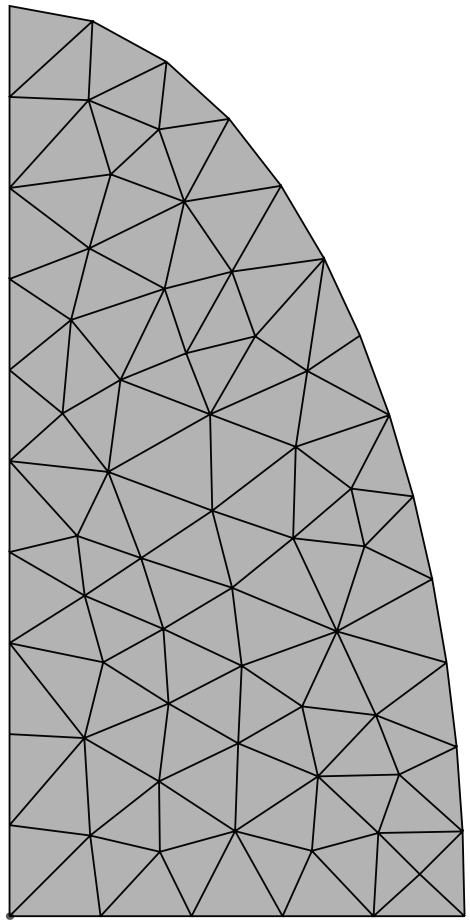
  with weights w_i and sampling points x_i, $i = 1, \ldots, N$.

- Different construction methods for w_i and x_i
  - Typically uses series of polynomials (Legendre, Lagrange, Lobatto, ...).
  - Exact for polynomial f up to a predefined order.
- Quadrature scheme depends on $\hat{\Omega}$!
  - Most schemes only available for simple shapes (triangle, square, tetrahedron, ...).
  - Quadrature on complicated shapes done by approximating Ω by small volumes of regular shape.

Computational Grid







The DUNE Grid Module

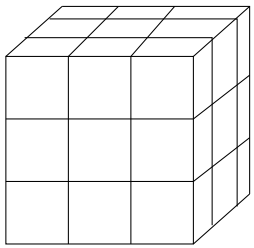
- The DUNE Grid Module is one of the five DUNE Core Modules.
- DUNE wants to provide an interfaces for grid-based methods. Therefore the concept of a *Grid* is the central part of DUNE.
- `dune-grid` provides the interfaces, following the concept of a *Grid*.
- Its implementation follows the three *design principles* of DUNE:

  Flexibility: Separation of data structures and algorithms.

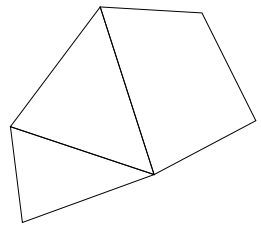
  Efficiency: Generic programming techniques.

  Legacy Code: Reuse existing finite element software.

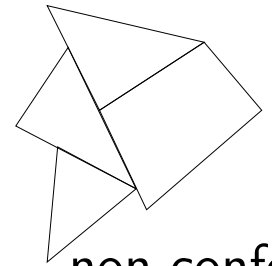
Designed to support a wide range of Grids



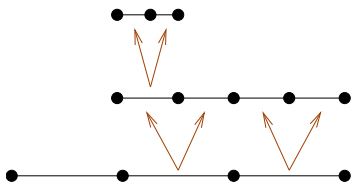
structured



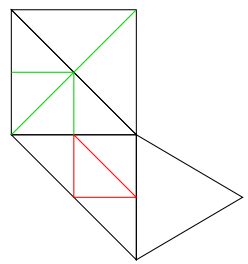
conforming



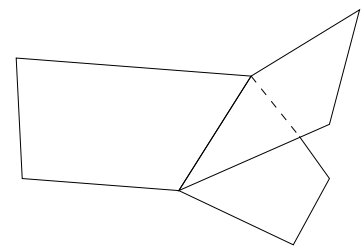
non conforming



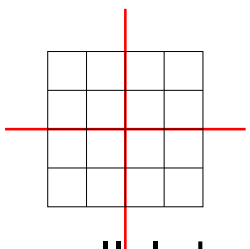
nested, 1D



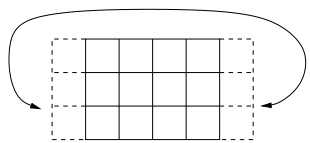
red-green, bisektion



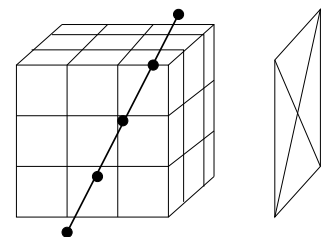
manifolds



parallel data decomposition



periodic



mixed dimensions

DUNE Grid Interface[1] Features

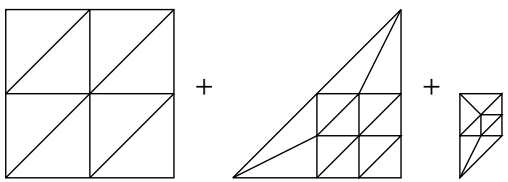
- Provide abstract interface to grids with:
  - Arbitrary dimension embedded in a world dimension,
  - multiple element types,
  - conforming or nonconforming,
  - hierarchical, local refinement,
  - arbitrary refinement rules (conforming or nonconforming),
  - parallel data distribution and communication,
  - dynamic load balancing.
- Reuse existing implementations (ALU, UG, Alberta) + special implementations (YaspGrid, FoamGrid).
- Meta-Grids built on-top of the interface (GeometryGrid, SubGrid, MultiDomainGrid)

[1] Bastian, Blatt, Dedner, Engwer, Klöfkorn, Kornhuber, Ohlberger, Sander: *A generic grid interface for parallel and adaptive scientific computing. Part I: Implementation and tests in DUNE*. Computing, 82(2-3):121–138, 2008.

Contents

The Grid

A formal specification of grids is required to enable an accurate description of the grid interface.



Hierarchic Grid

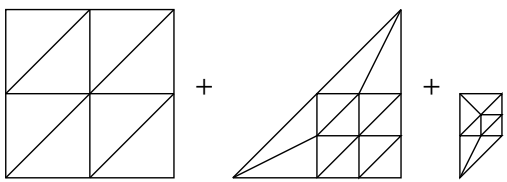
In DUNE a *Grid* is always a hierarchic grid of dimension d, existing in a w dimensional space.
The Grid is parametrised by

- the dimension d,
- the world dimension w
- and the maximum level J.

Within todays excercises we will always assume $d = w$ and we will ignore the hierarchic structure of the grids we deal with.

The Grid

A formal specification of grids is required to enable an accurate description of the grid interface.



Hierarchic Grid

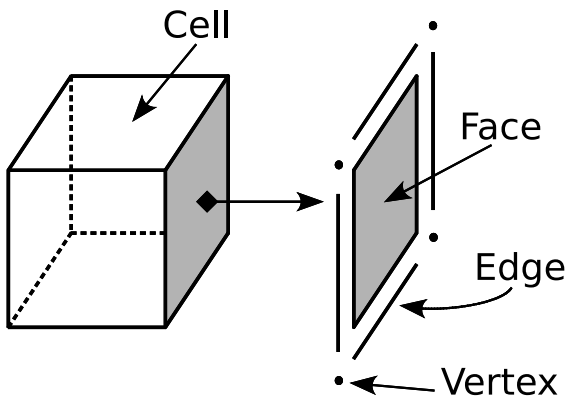
In DUNE a *Grid* is always a hierarchic grid of dimension d, existing in a w dimensional space.
The Grid is parametrised by

- the dimension d,
- the world dimension w
- and the maximum level J.

Within todays excercises we will always assume $d = w$ and we will ignore the hierarchic structure of the grids we deal with.

The Grid. . . A Container of Entities. . .

In the DUNE sense a *Grid* is a container of entities:

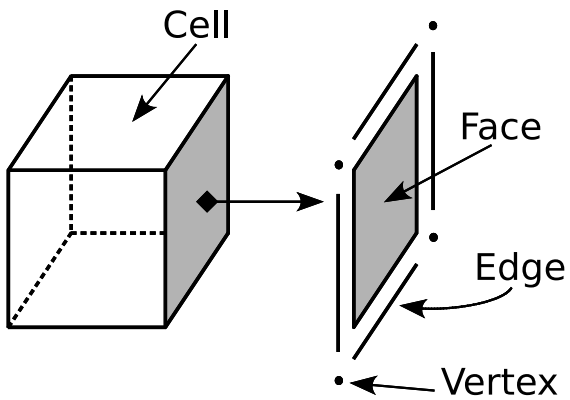


- vertices ,
- edges ,
- faces ,
- cells , . . .

In order to do dimension independent programming, we need a dimension independent naming for different entities.
We distinguish entities according to their codimension.
Entities of codim $= c$ contain subentities of codim $= c + 1$. This gives a recursive construction down to codim $= d$.

The Grid. . . A Container of Entities. . .

In the DUNE sense a *Grid* is a container of entities:

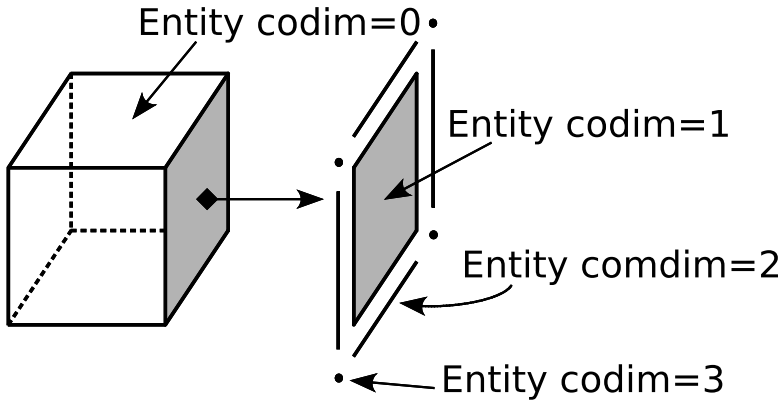


- vertices ,
- edges ,
- faces ,
- cells , . . .

In order to do dimension independent programming, we need a dimension independent naming for different entities.
We distinguish entities according to their codimension.
Entities of codim $= c$ contain subentities of codim $= c + 1$. This gives a recursive construction down to codim $= d$.

The Grid. . . A Container of Entities. . .

In the DUNE sense a *Grid* is a container of entities:



- vertices *(Entity codim* $= d$*)*,
- edges *(Entity codim* $= d - 1$*)*,
- faces *(Entity codim* $= 1$*)*,
- cells *(Entity codim* $= 0$*)*, . . .

In order to do dimension independent programming, we need a dimension independent naming for different entities.
We distinguish entities according to their codimension.
Entities of codim $= c$ contain subentities of codim $= c + 1$. This gives a recursive construction down to codim $= d$.

The DUNE Grid Interface

The DUNE Grid Interface is a collection of classes and methods

```
#include <dune/grid/yaspgrid.hh>

...

using Grid = Dune::YaspGrid<2>;
Grid grid({4,4},{1.0,1.0},{false,false});
auto gv = grid.leafGridView();
for (const auto& cell : elements(gv)) {
  // do something
}
```

We will now get to know the most important classes and see how they interact.

The DUNE Grid Interface

The DUNE Grid Interface is a collection of classes and methods

```
#include <dune/grid/yaspgrid.hh>

...

using Grid = Dune::YaspGrid<2>;
Grid grid({4,4},{1.0,1.0},{false,false});
auto gv = grid.leafGridView();
for (const auto& cell : elements(gv)) {
  // do something
}
```

We will now get to know the most important classes and see how they interact.

Modifying a Grid

The DUNE Grid interface follows the *View-only* Concept.

View-Only Concept

- Views offer (read-only) access to the data
  - Read-only access to grid entities allow the consequent use of `const`.
  - Access to entities is only through iterators for a certain view.

  → *This allows on-the-fly implementations.*
- Data can only be modified in the primary container *(the Grid)*

Modification Methods:

- Global Refinement
- Local Refinement & Adaption
- Load Balancing

Modifying a Grid

The DUNE Grid interface follows the *View-only* Concept.

View-Only Concept

- Views offer (read-only) access to the data
  - Read-only access to grid entities allow the consequent use of `const`.
  - Access to entities is only through iterators for a certain view.

  → *This allows on-the-fly implementations.*
- Data can only be modified in the primary container *(the Grid)*

Modification Methods:

- Global Refinement
- Local Refinement & Adaption
- Load Balancing

Modifying a Grid

The DUNE Grid interface follows the *View-only* Concept.

View-Only Concept

- Views offer (read-only) access to the data
  - Read-only access to grid entities allow the consequent use of `const`.
  - Access to entities is only through iterators for a certain view.

  → *This allows on-the-fly implementations.*
- Data can only be modified in the primary container *(the Grid)*

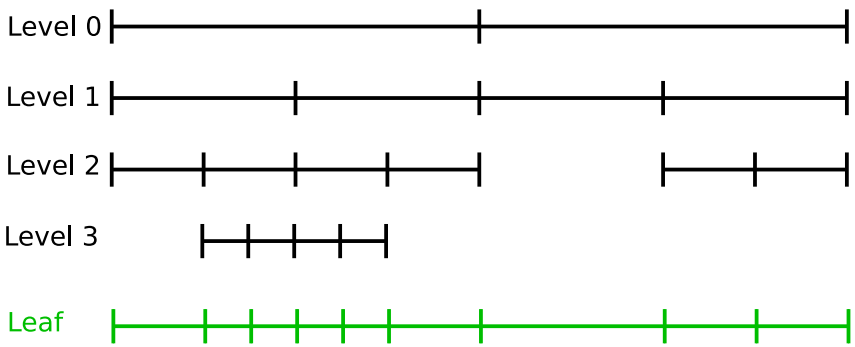
Modification Methods:

- Global Refinement
- Local Refinement & Adaption
- Load Balancing

Contents

Views to the Grid

A Grid offers two major views:

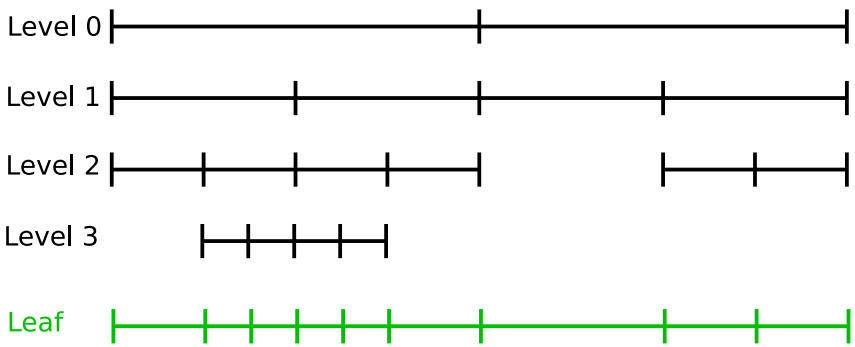


levelwise:
all entities associated with the same level.
Note: not all levels must cover the whole domain.

leafwise:
all leaf entities (entities which are not refined).
The leaf view can be seen as the projection of a levels onto a flat grid. It again covers the whole domain.

Views to the Grid

A Grid offers two major views:

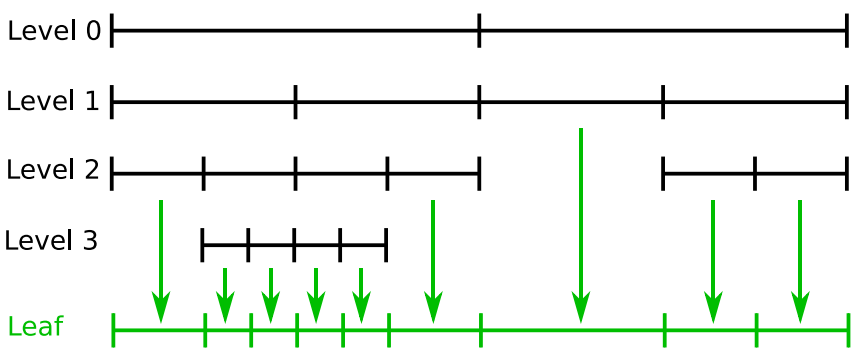


levelwise:
all entities associated with the same level.
Note: not all levels must cover the whole domain.

leafwise:
all leaf entities (entities which are not refined).
The leaf view can be seen as the projection of a levels onto a flat grid. It again covers the whole domain.

Views to the Grid

A Grid offers two major views:



levelwise:
all entities associated with the same level.
Note: not all levels must cover the whole domain.

leafwise:
all leaf entities (entities which are not refined).
The leaf view can be seen as the projection of a levels onto a flat grid. It again covers the whole domain.

Views to the Grid

Dune::GridView

- The `Dune::GridView` class consolidates all information depending on the current View.
- Every Grid must provide
  - `Grid::LeafGridView` and
  - `Grid::LevelGridView`.
- The `Grid` creates a new view every time you ask it for one, so you need to store a copy of it.
- Accessing the Views:
  - `Grid::leafGridView()` and
  - `Grid::levelGridView(int level)`.

Views to the Grid

Dune::GridView

- The `Dune::GridView` class consolidates all information depending on the current View.
- Every Grid must provide
  - `Grid::LeafGridView` and
  - `Grid::LevelGridView`.
- The `Grid` creates a new view every time you ask it for one, so you need to store a copy of it.
- Accessing the Views:
  - `Grid::leafGridView()` and
  - `Grid::levelGridView(int level)`.

Views to the Grid

Dune::GridView

- The `Dune::GridView` class consolidates all information depending on the current View.
- Every Grid must provide
  - `Grid::LeafGridView` and
  - `Grid::LevelGridView`.
- The `Grid` creates a new view every time you ask it for one, so you need to store a copy of it.
- Accessing the Views:
  - `Grid::leafGridView()` and
  - `Grid::levelGridView(int level)`.

Iterating over grid entities

Typically, most code uses the grid to iterate over some of its entities (e.g. cells) and perform some calculations with each of those entities.

- `GridView` supports iteration over all entities of one codimension.
- Iteration uses C++11 range-based for loops:

```
for (const auto& cell : elements(gv)) {
  // do some work with cell
}
```

- The type in front of `cell` is important:
  - If you create an entity in a range-based for loop, use `const auto&`.
  - In *all* other cases, use plain `auto`!

  If you do not follow this advice, your program may crash in unpredictable ways.

Iteration functions

```
for (const auto& cell : elements(gv)) {
  // do some work with cell
}
```

Depending on the entities you are interested in, you can use one of the following functions:

```
// Iterates over cells   (codim = 0)
for (const auto& c : elements(gv))
// Iterates over vertices  (dim = 0)
for (const auto& v : vertices(gv))
// Iterates over facets  (codim = 1)
for (const auto& f : facets(gv))
// Iterates over edges     (dim = 1)
for (const auto& e : edges(gv))

// Iterates over entities with a given codimension (here: 2)
for (const auto& e : entities(gv,Dune::Codim<2>{}))
// Iterates over entities with a given dimension (here: 2)
for (const auto& e : entities(gv,Dune::Dim<2>{}))
```

Contents

Entities

Iterating over a grid view, we get access to the entities.

```
for (const auto& cell : elements(gv)) {
  cell.?????(); // what can we do here?
}
```

- Entities cannot be modified.
- Entities can be copied and stored
  (but copies might be expensive!).
- Entities provide topological and geometrical information.

Entities

Iterating over a grid view, we get access to the entities.

```
for (const auto& cell : elements(gv)) {
  cell.?????(); // what can we do here?
}
```

- Entities cannot be modified.
- Entities can be copied and stored
  (but copies might be expensive!).
- Entities provide topological and geometrical information.

Entities

Iterating over a grid view, we get access to the entities.

```
for (const auto& cell : elements(gv)) {
  cell.?????(); // what can we do here?
}
```

- Entities cannot be modified.
- Entities can be copied and stored
  (but copies might be expensive!).
- Entities provide topological and geometrical information.

Entities

An Entity E provides both topological information

- Type of the entity (triangle, quadrilateral, etc.).
- Relations to other entities.

and geometrical information

- Position of the entity in the grid.

Entity E is defined by...

- Reference Element $\hat{\Omega}$
- Transformation T_E

Mapping from $\hat{\Omega}$ into global coordinates.

`GridView::Codim<c>::Entity` implements the entity concept.

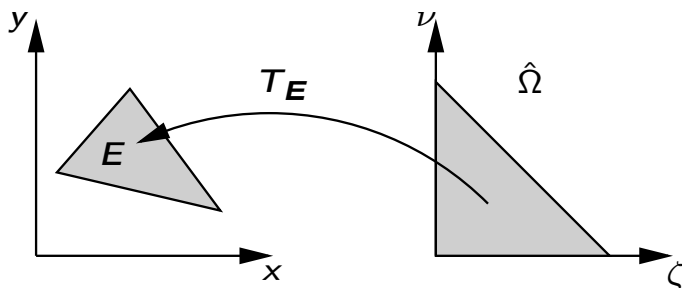
Entities

An Entity E provides both topological information

- Type of the entity (triangle, quadrilateral, etc.).
- Relations to other entities.

and geometrical information

- Position of the entity in the grid.



Mapping from $\hat{\Omega}$ into global coordinates.

Entity E is defined by...

- Reference Element $\hat{\Omega}$
- Transformation T_E

`GridView::Codim<c>::Entity` implements the entity concept.

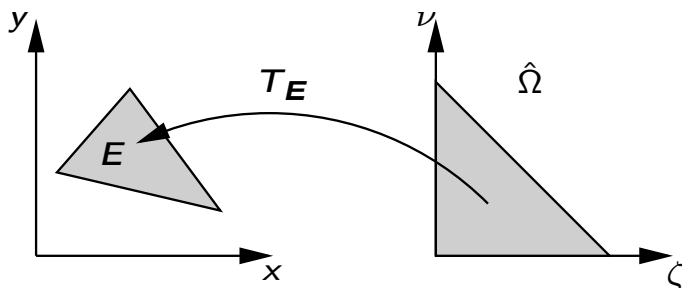
Entities

An Entity E provides both topological information

- Type of the entity (triangle, quadrilateral, etc.).
- Relations to other entities.

and geometrical information

- Position of the entity in the grid.



Mapping from $\hat{\Omega}$ into global coordinates.

Entity E is defined by...

- Reference Element $\hat{\Omega}$
- Transformation T_E

`GridView::Codim<c>::Entity` implements the entity concept.

Dimension and Codimension

Each entity has a **dimension**:

- `dim(vertex) == 0`
- `dim(line) == 1`
- `dim(triangle) == 2`
- `...`

When writing dimension-independent code, it is often easier to instead use the **codimension**.

The codimension of an entity e is always relative to the dimension of the grid and is given by:

$$\mathrm{codim}(e) = \dim(\mathit{grid}) - \dim(e)$$

- `codim(cell) == 0`
- `codim(face) == 1`
- `...`
- `codim(vertex) == dim(grid)`

Dimension and Codimension

Each entity has a **dimension**:

- `dim(vertex) == 0`
- `dim(line) == 1`
- `dim(triangle) == 2`
- `...`

When writing dimension-independent code, it is often easier to instead use the **codimension**.

The codimension of an entity e is always relative to the dimension of the grid and is given by:

$$\mathrm{codim}(e) = \mathrm{dim}(\mathit{grid}) - \mathrm{dim}(e)$$

- `codim(cell) == 0`
- `codim(face) == 1`
- `...`
- `codim(vertex) == dim(grid)`

Storing Entities

`GridView::Codim<c>::Entity`

- Entities can be copied and stored like any normal object.
- Important: There can be *multiple* entity objects for a single logical grid entity (because they can be copied)
- *Memory expensive, but fast.*

`GridView::Codim<c>::EntitySeed`

- Store minimal information to find an entity again.
- Create like this:

```
auto entity_seed = entity.seed();
```

- The grid can create a new `Entity` object from an `EntitySeed`:

```
auto entity = grid.entity(entity_seed);
```

- *Memory efficient, but run-time overhead to recreate entity.*

Storing Entities

`GridView::Codim<c>::Entity`

- Entities can be copied and stored like any normal object.
- Important: There can be *multiple* entity objects for a single logical grid entity (because they can be copied)
- *Memory expensive, but fast.*

`GridView::Codim<c>::EntitySeed`

- Store minimal information to find an entity again.
- Create like this:

```
auto entity_seed = entity.seed();
```

- The grid can create a new `Entity` object from an `EntitySeed`:

```
auto entity = grid.entity(entity_seed);
```

- *Memory efficient, but run-time overhead to recreate entity.*

Storing Entities

`GridView::Codim<c>::Entity`

- Entities can be copied and stored like any normal object.
- Important: There can be *multiple* entity objects for a single logical grid entity (because they can be copied)
- *Memory expensive, but fast.*

`GridView::Codim<c>::EntitySeed`

- Store minimal information to find an entity again.
- Create like this:

```
auto entity_seed = entity.seed();
```

- The grid can create a new `Entity` object from an `EntitySeed`:

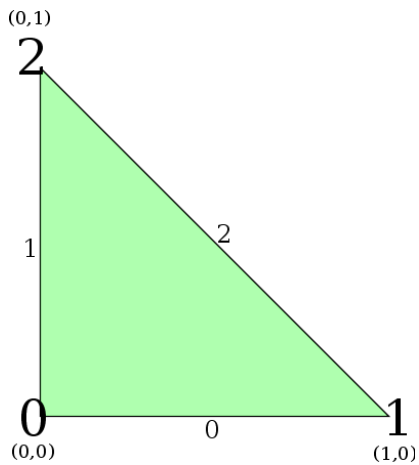
```
auto entity = grid.entity(entity_seed);
```

- *Memory efficient, but run-time overhead to recreate entity.*

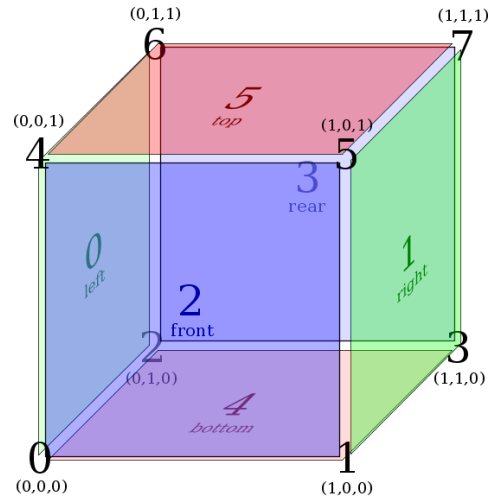
Reference Elements

`Dune::GeometryType` identifies the type of the entity's reference element.

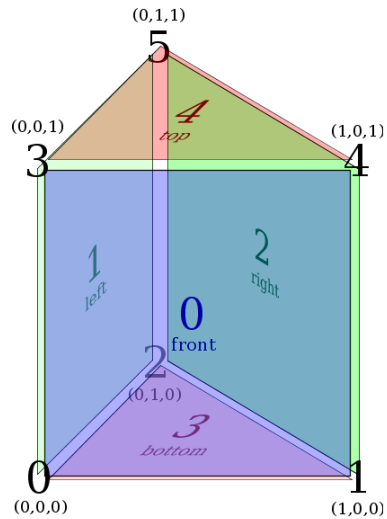
`Grid::Codim<c>::Entity::type()` returns the `GeometryType` of an entity.



simplex 2D



cube 3D



prism

Geometry Types

`GeometryType` is a simple identifier for a reference element

- Obtain from entity or geometry object using `.type()`
- `GeometryType` for specific reference elements in namespace `Dune::GeometryTypes`:

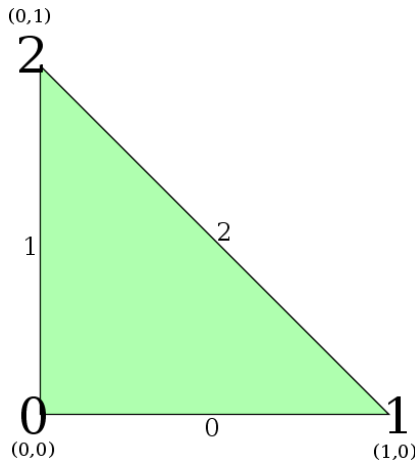
```
namespace GeometryTypes = Dune::GeometryTypes;
Dune::GeometryType gt;

gt = GeometryTypes::vertex;
gt = GeometryTypes::line;
gt = GeometryTypes::triangle;
gt = GeometryTypes::square;
gt = GeometryTypes::hexahedron;
gt = GeometryTypes::cube(dim);
gt = GeometryTypes::simplex(dim);
```

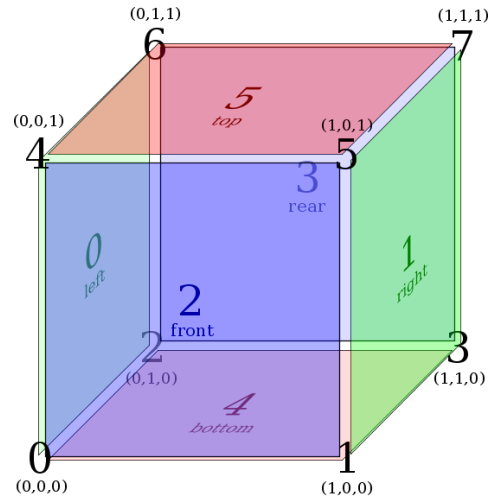
- `GeometryTypes` are cheap, always store and pass around copies (don't use references)

ReferenceElement (I)

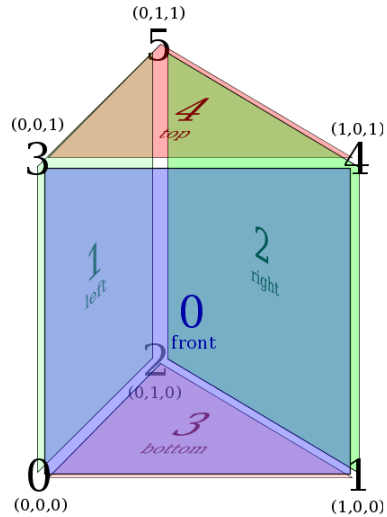
A reference element provides topological and geometrical information about the embedding of subentities:



simplex 2D



cube 3D



prism

- Numbering of subentities within the reference element
- Geometrical mappings from reference elements of subentities to the current reference element

ReferenceElement (II)

- Reference elements are templated on the dimension and the coordinate field type

```
Dune::ReferenceElement<double,dim> ref_el = ...;
```

- The function `Dune::referenceElement()` will extract the reference element from most objects that have one:

```
auto ref_el = Dune::referenceElement(entity.geometry());
```

  When using this function, you don't have to figure out the template parameters.

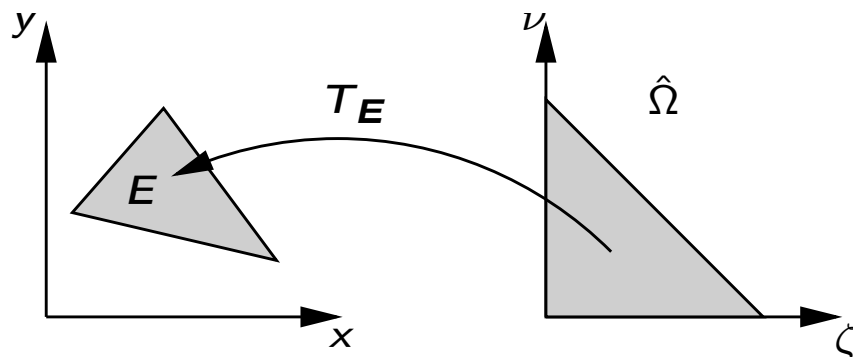
- `ReferenceElements` are cheap, always store and pass around copies (don't use references)

For more information see the documentation on reference elements (`https://dune-project.org/doxygen/master/group__GeometryReferenceElements.html`)

Geometry

Transformation T_E

- Maps from one space to an other.
- Main purpose is to map from the reference element to global coordinates.
- Provides transposed inverse of the Jacobian ($J^{-T}(T_E)$).



Geometry Interface (I)

- Obtain Geometry from entity

```
auto geo = entity.geometry();
```

- Convert local coordinate to global coordinate

```
auto x_global = geo.global(x_local);
```

- Convert global coordinate to local coordinate

```
auto x_local = geo.local(x_global);
```

Geometry Interface (II)

- Get center of geometry in global coordinates

```
auto center = geo.center();
```

- Get number of corners of the geometry (e.g. 3 for a triangle)

```
auto num_corners = geo.corners();
```

- Get global coordinates of i-th geometry corner ($0 \leq i <$`geo.corners()`)

```
auto corner_global = geo.corner(i);
```

Geometry Interface (III)

- Get type of reference element

```
auto geometry_type = geo.type(); // square, triangle, ...
```

- Find out whether geometry is affine

```
if (geo.affine()) {
  // do something optimized
}
```

- Get volume of geometry in global coordinate system

```
auto volume = geo.volume();
```

- Get integration element for a local coordinate (required for numerical integration)

```
auto mu = geo.integrationElement(x_local);
```

Gradient Transformation

Assume

$$f : \Omega \to \mathbb{R}$$

evaluated on a cell E, i.e. $f\left(T_E(\hat{x})\right)$.

The gradient of f is then given by

$$J_T^{-T}(\hat{x})\hat{\nabla} f\left(T_E(\hat{x})\right) :$$

```
auto x_global = geo.global(x_local);
auto J_inv = geo.jacobianInverseTransposed(x_local);
auto tmp = gradient(f)(x_global); // gradient(f) supplied by user
auto gradient = tmp;
J_inv.mv(tmp,gradient);
```

Obtaining Quadrature Rules

Recall: Numerical quadrature rules given by

$$\int_{\hat{\Omega}} f(x)\,dx \approx \sum_{i=1}^{N} w_i f(x_i)$$

- `dune-geometry` provides pre-defined quadrature rules for common geometry types:

```
int order = ...;
Dune::GeometryType gt = ...;
auto& qr = Dune::QuadratureRules<double,dim>::rule(gt,order);
```

- The rule factory is parameterized by the number type (typically use `Grid::ctype`) and the dimension of the integration domain
- The rule is exact for polynomials up to the given `order`
- Use `auto&` for the type of the rule to avoid expensive copies
- Optional third parameter to select type of rule (Jacobi, Legendre, Lobatto)

Using Quadrature Rules

- A `QuadratureRule` is a range of `QuadraturePoint`.
- `QuadraturePoint` provides weight and position:
  - `QuadraturePoint::weight()`
  - `QuadraturePoint::position()`

Example

```
auto f = some_function_to_integrate(...);
double integral = 0.0;
for (const auto& qp : rule)
{
  integral += f(qp.position()) * qp.weight();
}
```

Attention: When integrating over cells in a grid, keep in mind that the quadrature point coordinates are local to the reference element and need to be transformed when integrating an analytical function!

Using Quadrature Rules

- A `QuadratureRule` is a range of `QuadraturePoint`.
- `QuadraturePoint` provides weight and position:
  - `QuadraturePoint::weight()`
  - `QuadraturePoint::position()`

Example

```
auto f = some_function_to_integrate(...);
double integral = 0.0;
for (const auto& qp : rule)
{
  integral += f(qp.position()) * qp.weight();
}
```

Attention: When integrating over cells in a grid, keep in mind that the quadrature point coordinates are local to the reference element and need to be transformed when integrating an analytical function!

Using Quadrature Rules

- A `QuadratureRule` is a range of `QuadraturePoint`.
- `QuadraturePoint` provides weight and position:
  - `QuadraturePoint::weight()`
  - `QuadraturePoint::position()`

Example

```
auto f = some_function_to_integrate(...);
double integral = 0.0;
for (const auto& qp : rule)
{
  integral += f(qp.position()) * qp.weight();
}
```

Attention: When integrating over cells in a grid, keep in mind that the quadrature point coordinates are local to the reference element and need to be transformed when integrating an analytical function!

Quadrature Rule Access in PDELab

Most of the time, you want a quadrature rule for an entity geometry

$\Longrightarrow$ no need to specify template types

PDELab extension to simplify access:

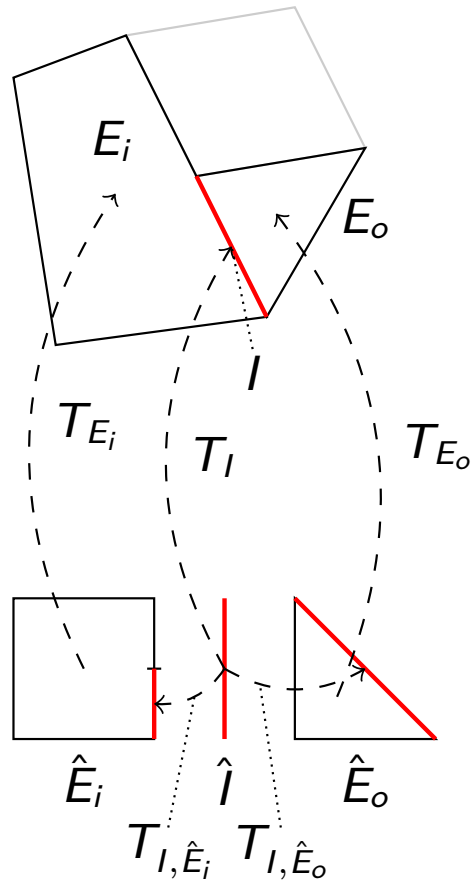
```
#include <dune/pdelab/common/quadraturerules.hh>

...

// PDELab quadrature rules wrap dune-geometry rules and are
// cheap to copy, so use "auto" instead of "auto&"
auto quad = Dune::PDELab::quadratureRule(geometry,order);
for (const auto& qp : quad)
{
    auto x_local = qp.position();
    auto w = qp.weight();
}
```

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

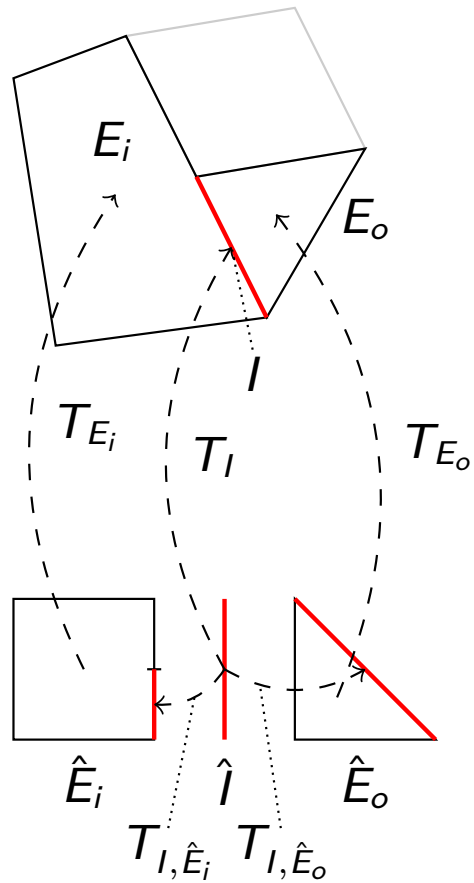
```
auto world_geo =
    intersection.geometry();
```

- 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.

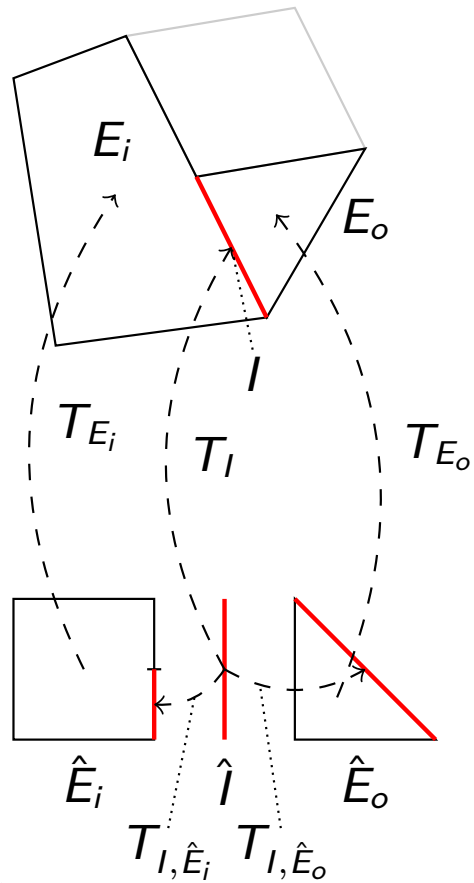
```
auto unit_outer_normal =
    intersection.unitOuterNormal(x_local);
```

- 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);
```



Example: Iterating over intersections

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
    }
  }
```

Example: Elementwise divergence of a vector field

$$\int_{\Omega_e} \nabla \cdot f(x)\, dx = \int_{\partial\Omega_e} f \cdot n_e\, ds$$

Go to code example 3 ...

Contents

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:

- Allow association of FE computations 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

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

- 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.

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

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

- 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.

```
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))
  lengths[index_set.index(edge)] = edge.geometry().volume();
```

Sequential finite volume solver

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 explicit cell-centered finite volume method reads

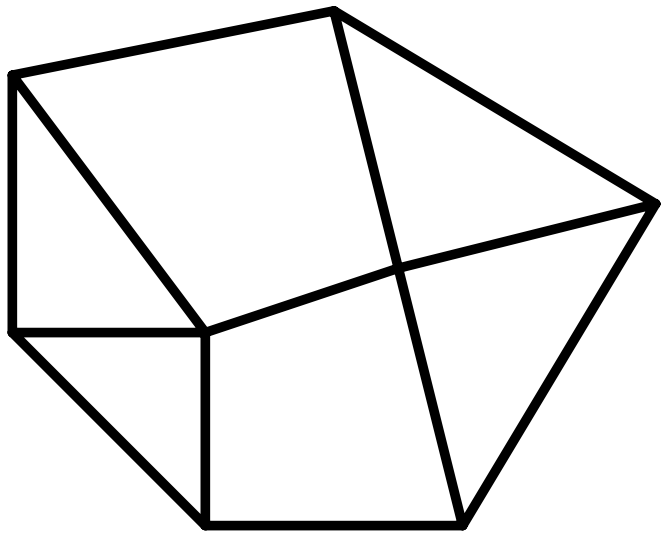
$$\bar{u}_e^{k+1} = \bar{u}_e^k - \frac{\Delta t}{|\Omega_e|} \sum_{(e,e') \in I(e)} \Phi(v \cdot n_e, \bar{u}_e^k, \bar{u}_{e'}^k) |I(e, e')|$$

with the numerical flux function Φ chosen as upwind flux here.

Go to code example 4 ...

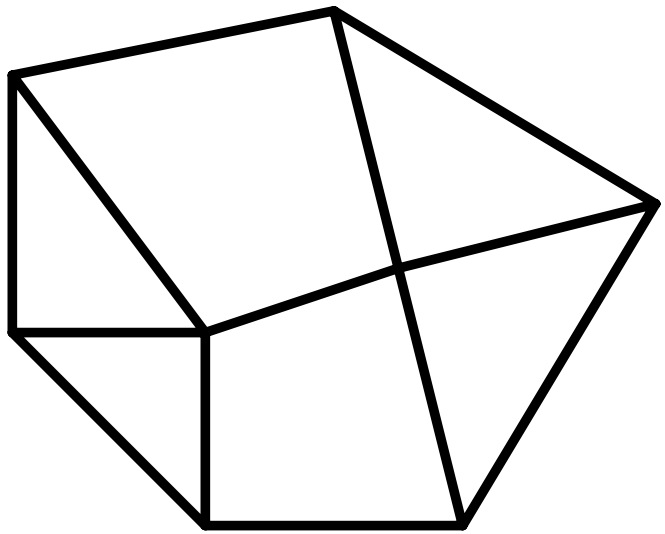
Example: 2D Multi-Element Grid – Indices

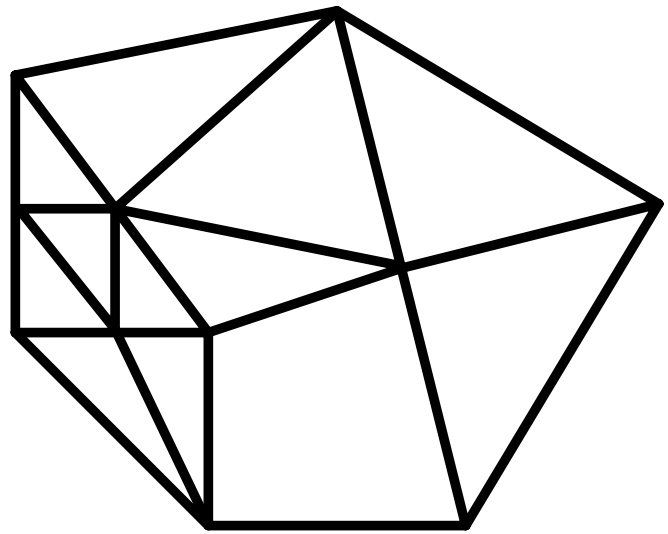
Locally refined grid:



Example: 2D Multi-Element Grid – Indices

Locally refined grid:

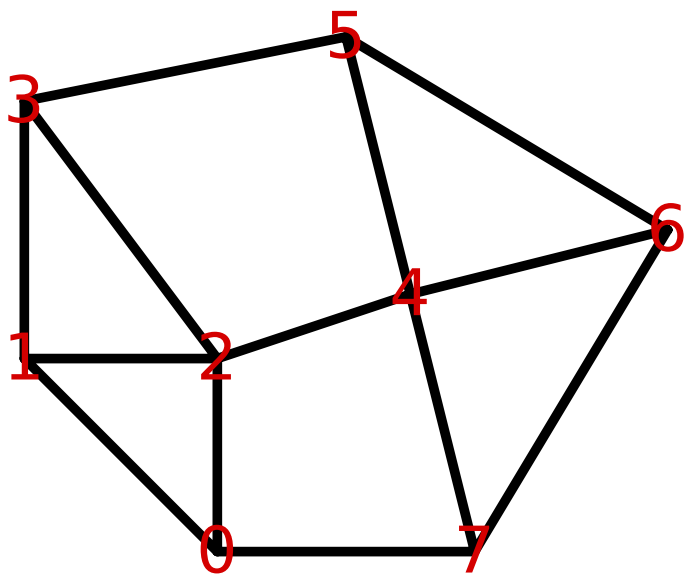


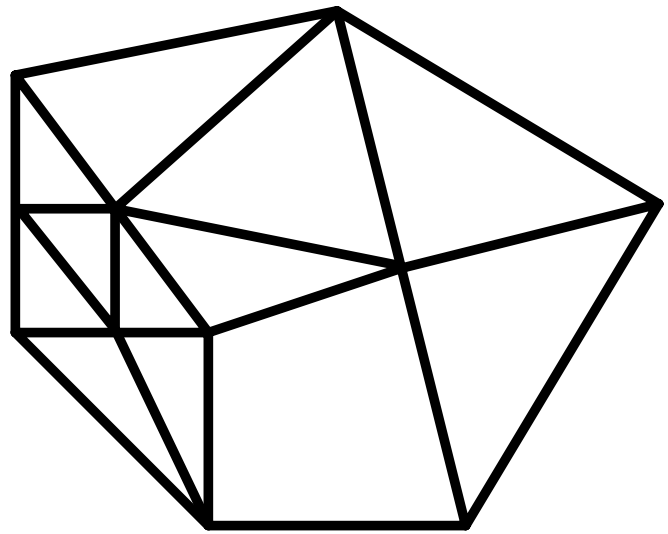


Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:



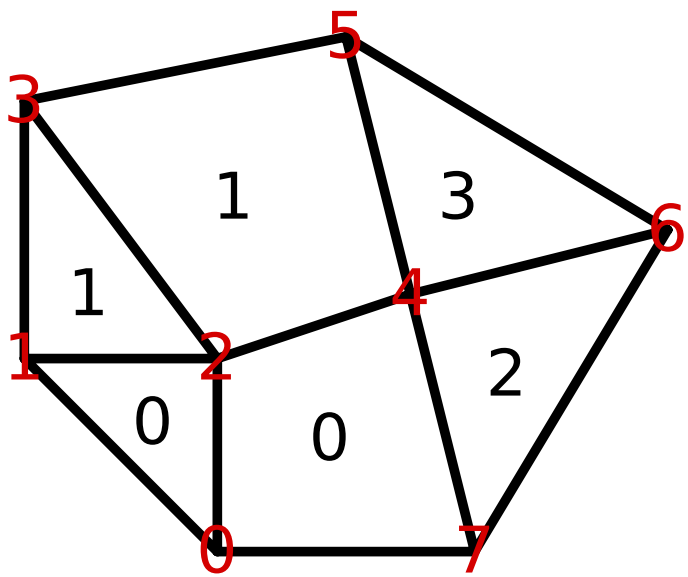


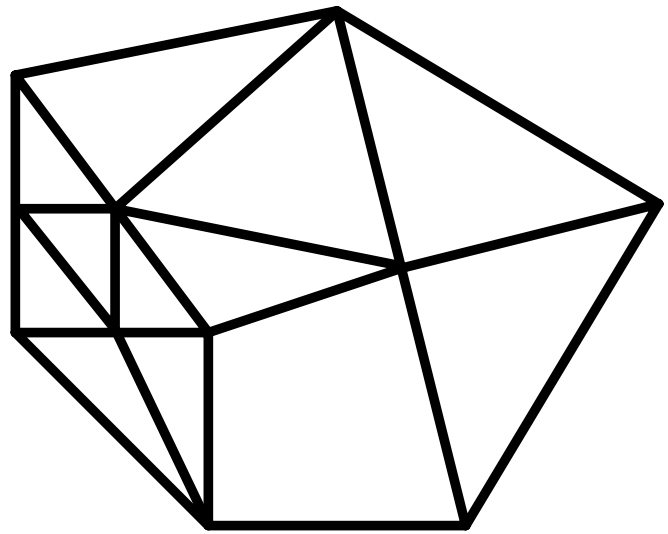
Consecutive index for vertices

Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:



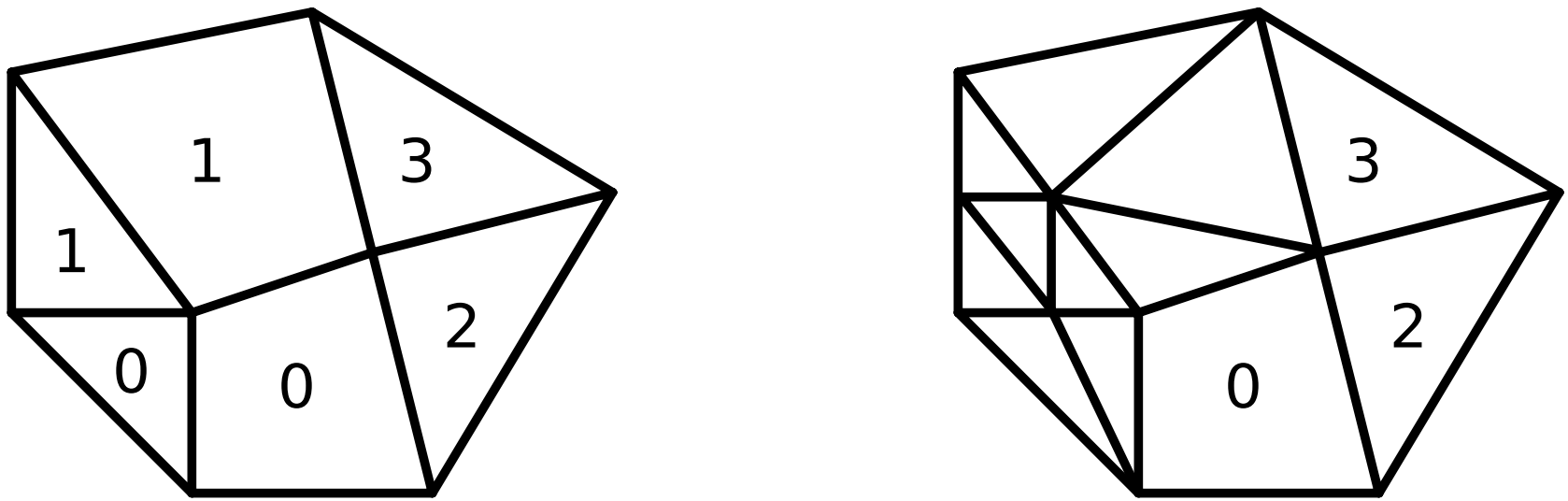


. . . and cells

Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:

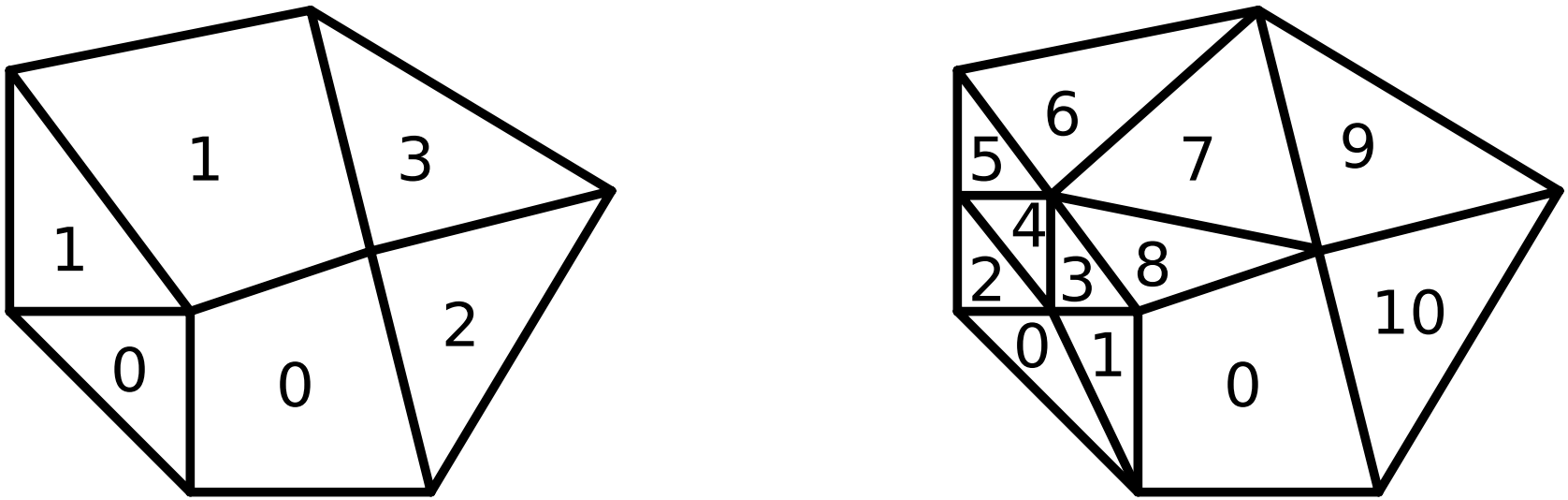


Old cell indices on refined grid

Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:

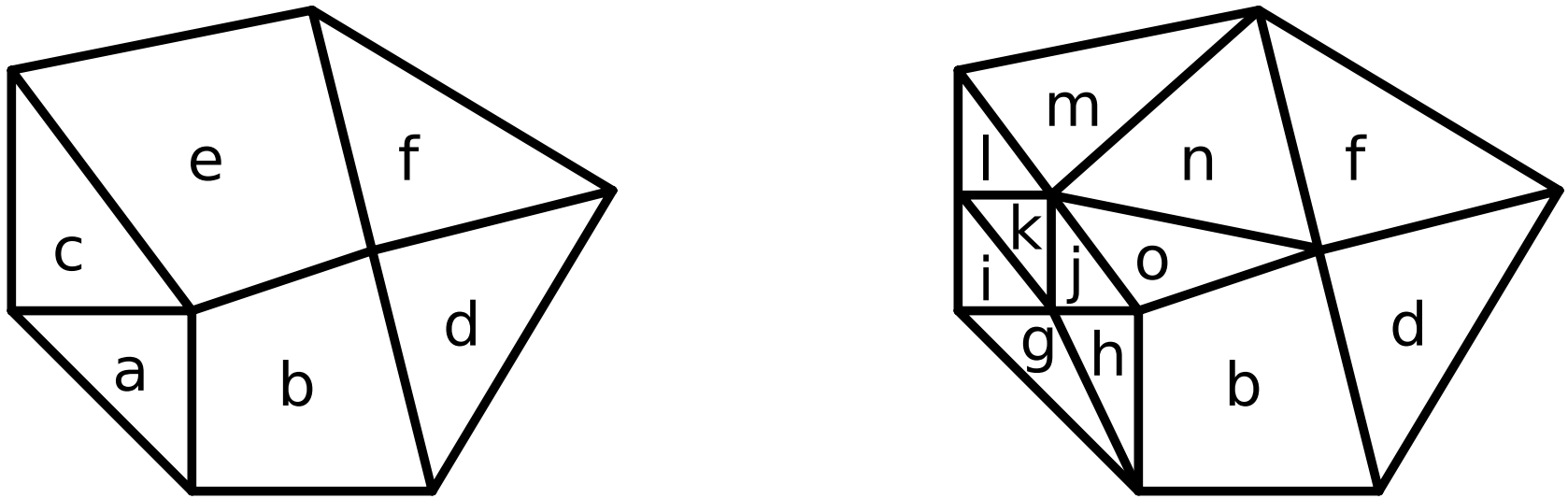


Consecutive cell indices on coarse and refined grid

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.

You will be using the

`Dune::MultipleCodimMultipleGeomTypeMapper<GridView>`.

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.

You will be using the

`Dune::MultipleCodimMultipleGeomTypeMapper<GridView>`.

Example: Mapper (I)

```
#include <dune/grid/common/mcmgmapper.hh>
...

typedef Dune::SomeGrid::LeafGridView GridView;
...

/* create a mapper*/
// 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>;
```

Example: Mapper (II)

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

/* setup sparsity pattern */
// 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());
            matrix[index].insert(nindex);
        }
    }
}
```

Contents

Further Reading

What we didn't discuss...

- grid adaptation
- parallelization, load balancing
- further specialized methods (e.g. related to grid hierarchy)

Further Reading

Literature

Cite when using the DUNE grid interface...

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander.
A Generic Grid Interface for Parallel and Adaptive Scientific Computing. *Part I: Abstract Framework.*
Computing, 82(2–3), 2008, pp. 103–119.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander.
A Generic Grid Interface for Parallel and Adaptive Scientific Computing. *Part II: Implementation and Tests in DUNE.*
Computing, 82(2–3), 2008, pp. 121–138.