The DUNE Grid Interface
An Introduction

Christian Engwer

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

March 7, 2017

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

The DUNE Framework

Design Principles

Flexibility: Seperation of data structures and algorithms.

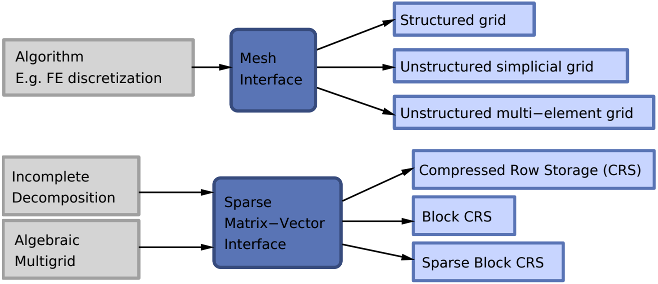
Efficiency: Generic programming techniques.

Legacy Code: Reuse existing finite element software.

Flexibility

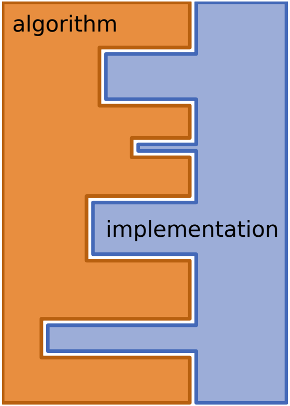
Seperate data structures and algorithms.

- 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
   - Engine Concept (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.

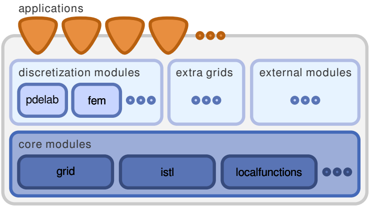
Contents

Design Principles

The DUNE Framework

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 their own modules.
- Portability
- Open Development Process
- Free Software Licence



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

DUNE Release 2.4.1

Current stable version is 2.4.1,
available since February 29th 2015.

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.

DUNE http://www.dune-project.org/

DUNE ecosystem

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

Discretization Modules:

dune-pdelab: discretization module based on dune-localfunctions.

dune-fem: Alternative implementation of finite element functions.

dune-functions: A new initiative to provide unified interfaces for functions and function spaces.

External Modules:

Kaskade 7: Simulation Suite – uses Dune for the grid and linear algebra infrastructure.

DuMu^x: simulations of flow and transport processes in porous media. Development is in an early state.

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-networkgrid: is a grid manager for a network of 1d entities in a 3d 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.

...

DUNE ecosystem

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

Discretization Modules:

dune-pdelab: discretization module based on dune-localfunctions.

dune-fem: Alternative implementation of finite element functions.

dune-functions: A new initiative to provide unified interfaces for functions and function spaces.

External Modules:

Kaskade 7: Simulation Suite – uses Dune for the grid and linear algebra infrastructure.

DuMu^x: simulations of flow and transport processes in porous media. Development is in an early state.

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-networkgrid: is a grid manager for a network of 1d entities in a 3d 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.

...

DUNE ecosystem

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

Discretization Modules:

dune-pdelab: discretization module based on dune-localfunctions.

dune-fem: Alternative implementation of finite element functions.

dune-functions: A new initiative to provide unified interfaces for functions and function spaces.

External Modules:

Kaskade 7: Simulation Suite – uses Dune for the grid and linear algebra infrastructure.

DuMu^x: simulations of flow and transport processes in porous media. Development is in an early state.

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-networkgrid: is a grid manager for a network of 1d entities in a 3d 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.

...

DUNE ecosystem

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

Discretization Modules:

dune-pdelab: discretization module based on dune-localfunctions.

dune-fem: Alternative implementation of finite element functions.

dune-functions: A new initiative to provide unified interfaces for functions and function spaces.

External Modules:

Kaskade 7: Simulation Suite – uses Dune for the grid and linear algebra infrastructure.

DuMu^x: simulations of flow and transport processes in porous media. Development is in an early state.

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-networkgrid: is a grid manager for a network of 1d entities in a 3d 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.

. . .

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 $\Omega \subset \mathbb{R}^d$.

How to evaluate the integrals?

- No analytic integrals available for $a(u,v)$ and $l(v)$.
- No analytic description for the shape of $\Omega \subset \mathbb{R}^d$.

⟶ Use a numerical quadrature scheme!

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 $\Omega \subset \mathbb{R}^d$.

How to evaluate the integrals?

- No analytic integrals available for $a(u, v)$ and $l(v)$.
- No analytic description for the shape of $\Omega \subset \mathbb{R}^d$.

→ Use a numerical quadrature scheme!

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 $\Omega \subset \mathbb{R}^d$.

How to evaluate the integrals?

- No analytic integrals available for $a(u,v)$ and $l(v)$.
- No analytic description for the shape of $\Omega \subset \mathbb{R}^d$.

→ Use a numerical quadrature scheme!

Numerical Quadrature

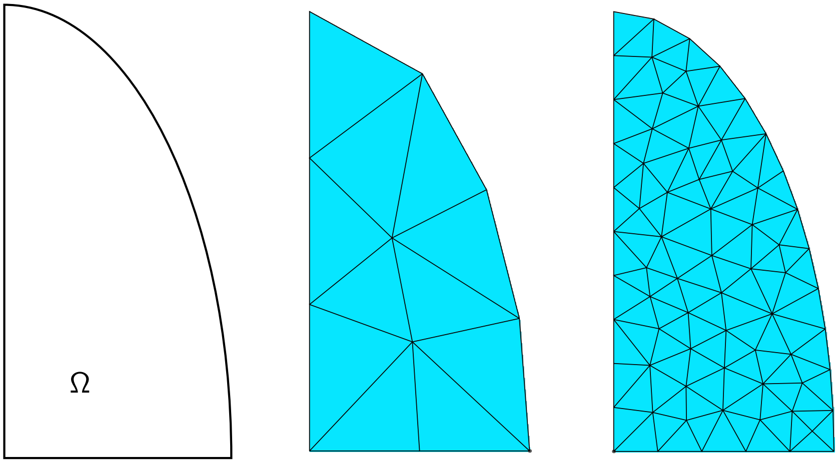
- Approximate integral by a weighted sum of function evaluations at sampling points:

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

  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 Ω!
  - 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

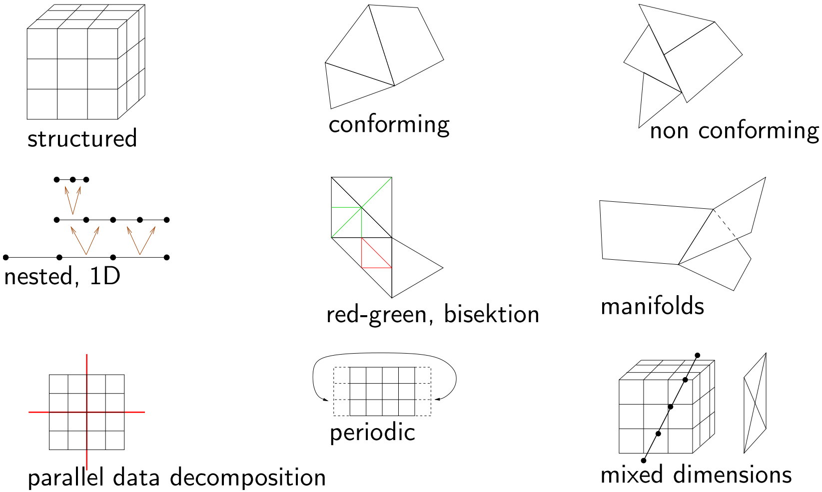
- The DUNE Grid Module is one of the five DUNE Core Modules.
- DUNE wants to provide an interfaces for grid-based methods. Therefore the concept of a *Grid* is the central part of DUNE.
- `dune-grid` provides the interfaces, following the concept of a *Grid*.
- Is implementation follows the three *design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



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

Views to the Grid

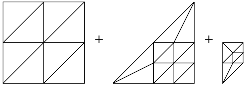
Entities

Attaching Data to the Grid

Further Reading

The Grid

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



Hierarchic Grid

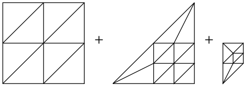
In DUNE a *Grid* is always a hierarchic grid of dimension d, existing in a w dimensional space.
The Grid is parametrised by

- 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

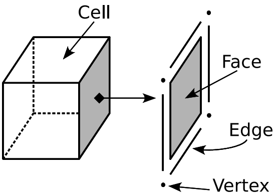
In DUNE a *Grid* is always a hierarchic grid of dimension d, existing in a w dimensional space.
The Grid is parametrised by

- 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:

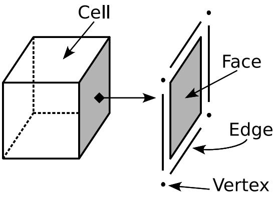


- vertices ,
- edges ,
- faces ,
- cells , . . .

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:

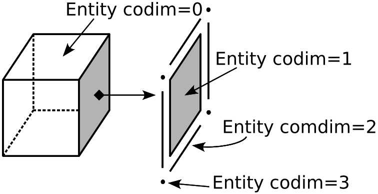


- vertices ,
- edges ,
- faces ,
- cells , . . .

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

The Grid

Views to the Grid

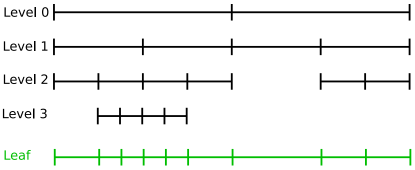
Entities

Attaching Data to the Grid

Further Reading

Views to the Grid

A Grid offers two major views:

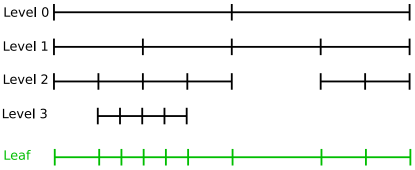


levelwise:
all entities associated with the same level.
Note: not all levels must cover the whole domain.

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:

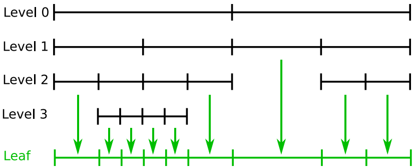


levelwise:
all entities associated with the same level.
Note: not all levels must cover the whole domain.

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

The Grid

Views to the Grid

Entities

Attaching Data to the Grid

Further Reading

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.

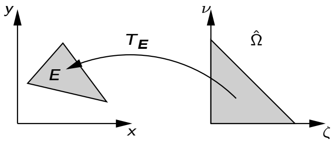
Entities

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.

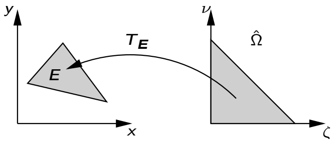
Entities

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.

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

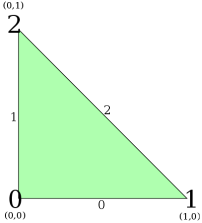
`Dune::GeometryType` identifies the type of the entities

Referenceelement.

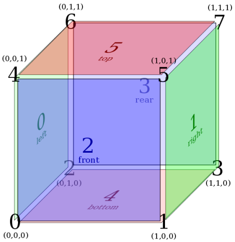
It bundles a *topology ID* and the dimension.

`Grid::Codim<c>::Entity::type()`

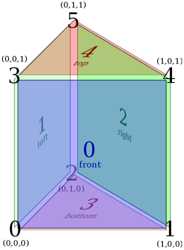
returns the GeometryType of the entity.



simplex 2D



cube 3D

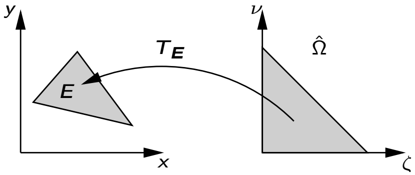


prism

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. $f\big(T_E(\hat{x})\big)$.

The gradient of f is then given by

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

```
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);
```

Quadrature Rules

- guarantees exact integration of polynomial functions of order k.
- Part of dune-geometry
- Given `Geometry` and quadrature order, we obtain the `QuadratureRule`.
- A `QuadratureRule` is a range of `QuadraturePoint`.
- `QuadraturePoint` gives weight an position:
  `QuadraturePoint::weight()` `QuadraturePoint::position()`

Note: Simple access to `QuadratureRule` provided by dune-pdelab

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

...

auto quad = Dune::PDELab::quadratureRule(geometry,order);
for (const auto& qp : quad)
{
    auto x_local = qp.position();
    auto w = qp.weight();
}
```

Quadrature Rules

- guarantees exact integration of polynomial functions of order k.
- Part of dune-geometry
- Given `Geometry` and quadrature order, we obtain the `QuadratureRule`.
- A `QuadratureRule` is a range of `QuadraturePoint`.
- `QuadraturePoint` gives weight an position:
  `QuadraturePoint::weight()` `QuadraturePoint::position()`

Note: Simple access to `QuadratureRule` provided by dune-pdelab

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

...

auto quad = Dune::PDELab::quadratureRule(geometry,order);
for (const auto& qp : quad)
{
    auto x_local = qp.position();
    auto w = qp.weight();
}
```

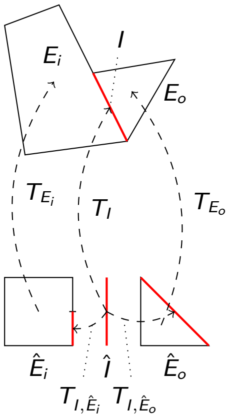
Example: Average of a function f on a GridView

$$\frac{1}{|\Omega|}\int_\Omega f(x)\,dx \approx \frac{1}{\sum_{E\in GV}|e|}\sum_{E\in GV}\sum_{i\in QR} f(T_e(x_i))w_i|\det J_E^T(x_i)|^{1/2}$$

```
double value = 0.0, volume = 0.0;
for (const auto& cell : elements(gv)) {
  auto geo = cell.geometry();
  // integrate with numerical quadrature
  for (auto& qp : Dune::PDELab::quadratureRule(geo,2)) {
    auto x_local = qp.position();
    auto x_global = geo.global(x_local);
    // accumulate integral contribution
    value += f(x_global) *
             qp.weight() * geo.integrationElement(x_local);
  }
  volume += geo.volume();
}
std::cout << "Average: " << value / volume << std::endl;
```

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

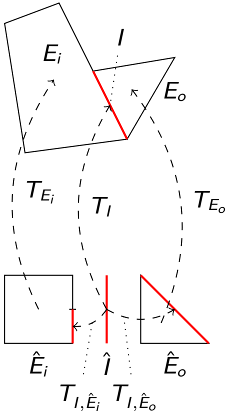
```
auto world_geo =
    intersection.geometry();
```

- 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.

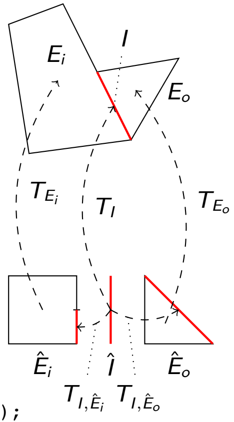
```
auto unit_outer_normal =
    intersection.unitOuterNormal(x_local);
```

- 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
    }
  }
```

Contents

The Grid

Views to the Grid

Entities

Attaching Data to the Grid

Further Reading

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

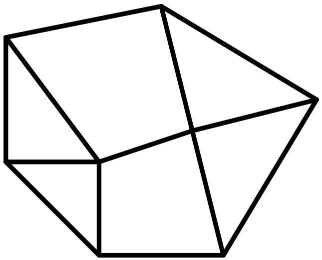
The following example demonstrates how to

- 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();
```

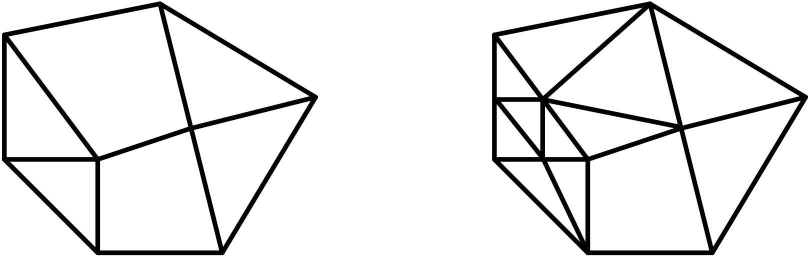
Example: 2D Multi-Element Grid – Indices

Locally refined grid:



Example: 2D Multi-Element Grid – Indices

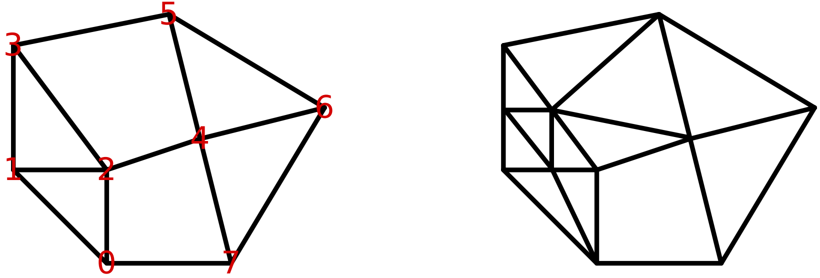
Locally refined grid:



Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:

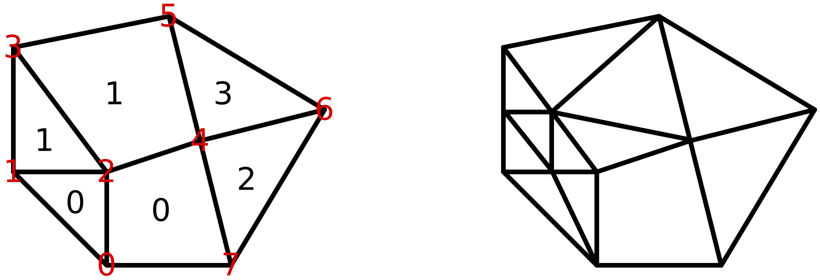


Consecutive index for vertices

Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:

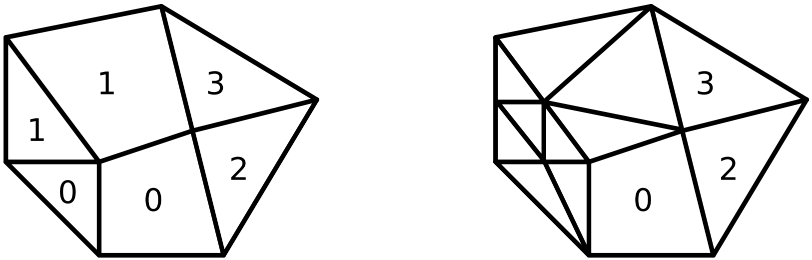


. . . and cells

Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:

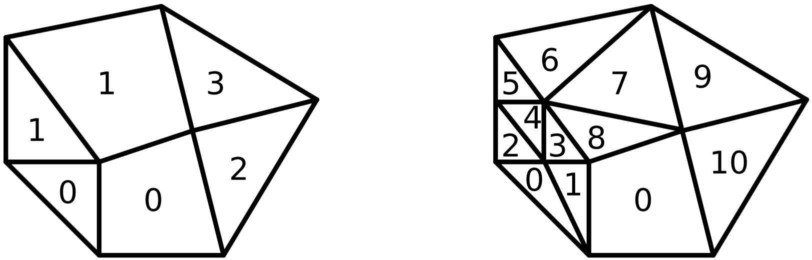


Old cell indices on refined grid

Example: 2D Multi-Element Grid – Indices

Locally refined grid:

Indices:

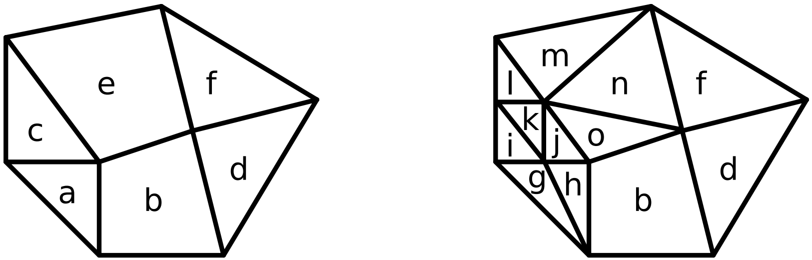


Consecutive cell indices on coarse and refined grid

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,Layout>`.

Example: Mapper (I)

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

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

/* create a mapper*/
// Layout description (equivalent to Dune::MCMGElementLayout)
template<int dim>
struct CellData {
    bool contains (Dune::GeometryType gt) {
        return gt.dim() == dim;
    }
};

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

Example: Mapper (II)

```
using Mapper =
    Dune::MultipleCodimMultipleGeomTypeMapper<GridView,CellData>;
Mapper mapper(gridview);

/* 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

The Grid

Views to the Grid

Entities

Attaching Data to the Grid

Further Reading

Further Reading

What we didn't discuss...

- grid creation
- I/O
- grid adaptation
- parallelization
- further specialized methods

Further Reading

Literature

- 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.