The DUNE Grid Interface

> "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*

The DUNE Grid Interface

Weak formulation of boundary value problem:

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

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

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, dx, \quad l(v) = \int_\Omega f(x)\, v \, dx,$$

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

The DUNE Grid Interface

Weak formulation of boundary value problem:

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

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

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx, \quad l(v) = \int_\Omega f(x) \, v \, dx,$$

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

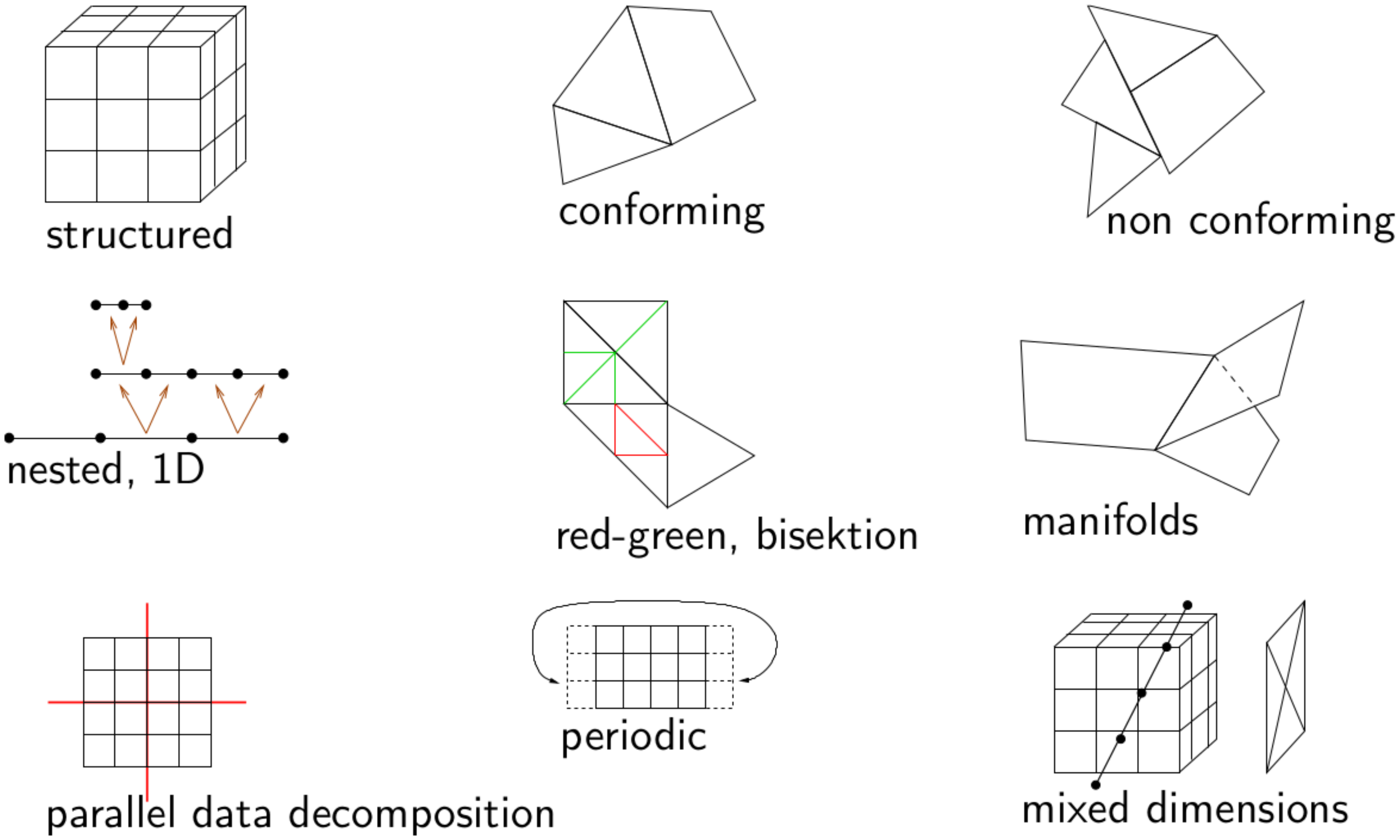
Grids are necessary for at least three reasons:

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

The DUNE Grid Interface

- 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



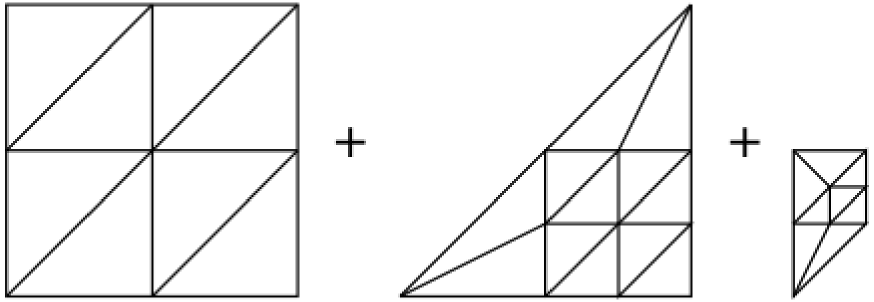
DUNE Grid Interface Features

- 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, SubdomainGrid, CurvedGrid)

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. (2008)

The Grid

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

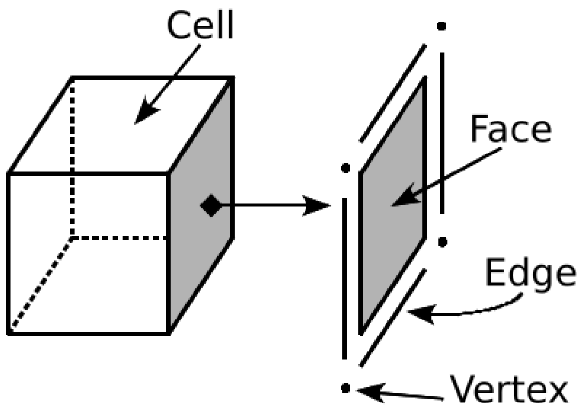


Hierarchic Grid

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

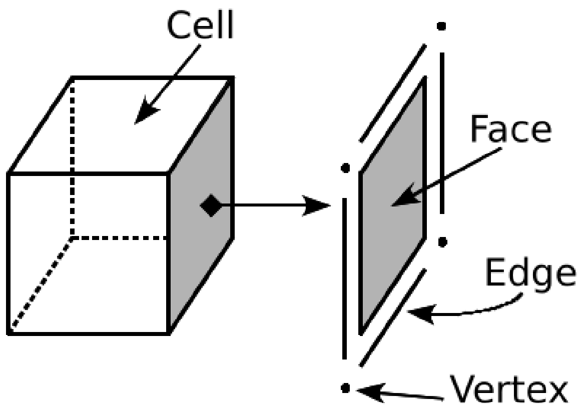
- the dimension `d`,
- the world dimension `w`
- and the maximum level `J`.

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



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

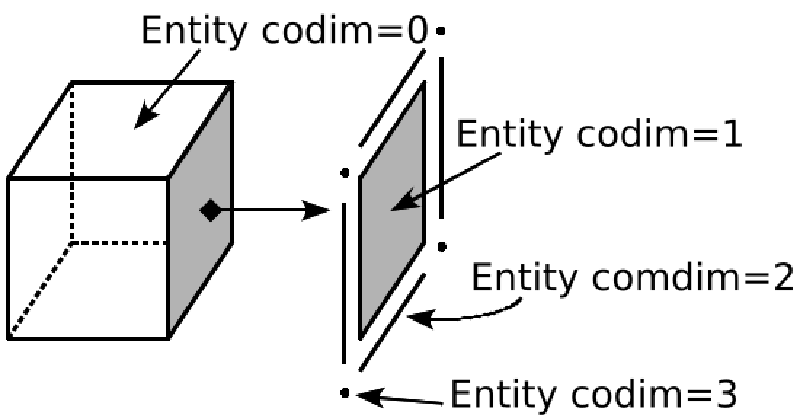
The Grid... A Container of Entities...



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

In order to do dimension independent programming, we need a dimension independent naming for different entities.

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



- vertices (Entity `codim = d`)
- edges (Entity `codim = d - 1`)
- faces (Entity `codim = d - 2`)
- ...
- 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

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

// ...

// Create a 2d structured grid of [0,1] x [0,1]
using Grid = Dune::YaspGrid<2>;
Grid grid{ {1.0, 1.0}, {4, 4} };

// traverse the grid
auto gv = grid.leafGridView();
for (auto const& cell : elements(gv)) {
  // do something
}
```

The DUNE Grid Interface

The DUNE Grid Interface is a collection of classes and methods

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

// ...

// Create a 2d structured grid of [0,1] x [0,1]
using Grid = Dune::YaspGrid<2>;
Grid grid{ {1.0, 1.0}, {4, 4} };

// traverse the grid
auto gv = grid.leafGridView();
for (auto const& 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*)

Modifying a Grid

The DUNE Grid interface follows the View-only Concept.

View-Only Concept

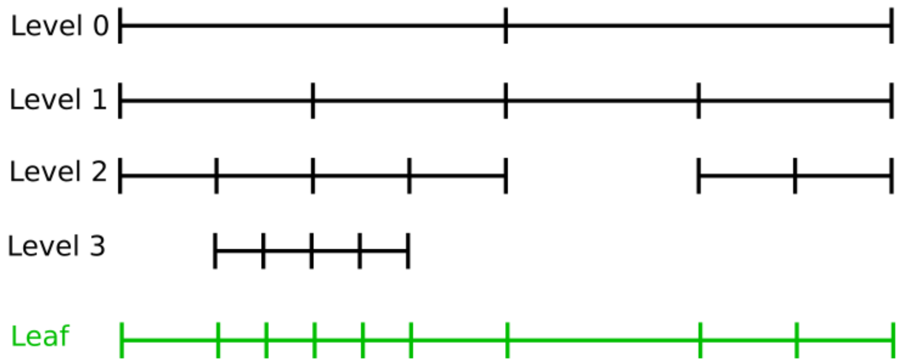
- Views offer (read-only) access to the data
  - Read-only access to grid entities allow the consequent use of `const`.
  - Access to entities is only through iterators for a certain view.
  - This allows on-the-fly implementations.
- Data can only be modified in the primary container (*the Grid*)

Modification Methods:

- Global Refinement
- Local Refinement & Adaption
- Load Balancing

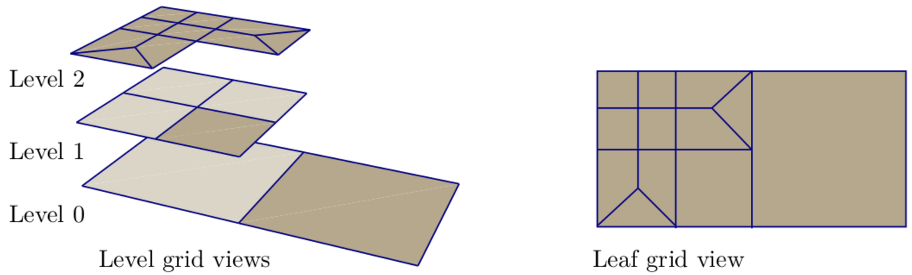
Views to the Grid

A Grid offers two major views:



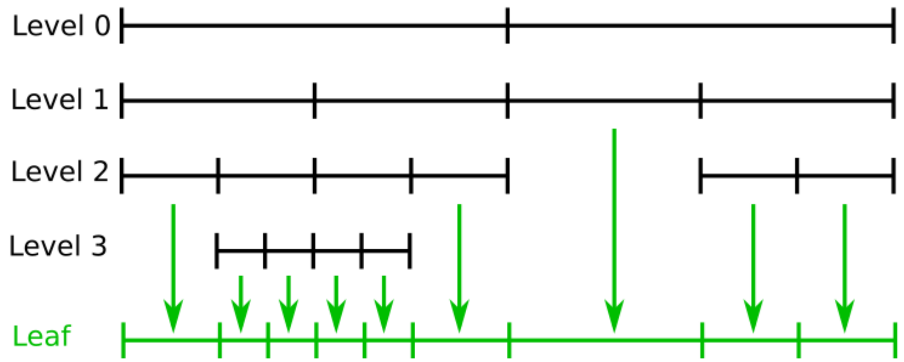
levelwise: all entities associated with the same level.

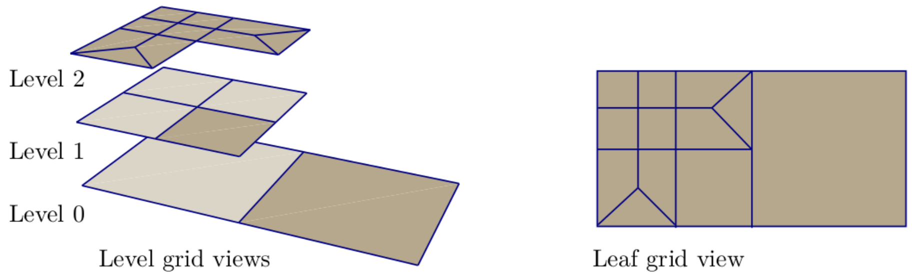
leafwise: all leaf entities (entities which are not refined).



Views to the Grid

A Grid offers two major views:





Level grid views

Leaf grid view

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.

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`.

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

Iteration functions

You can do similar calls for other entity types:

```
// 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>{}))
```

Entities

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.

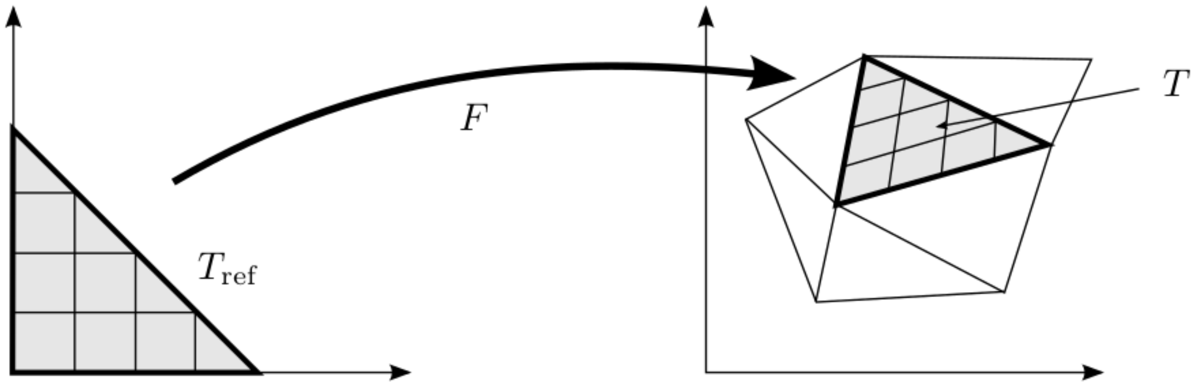
Entities

An Entity `T` 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 `T` is defined by...

- Reference Element T_{ref}
- Transformation F_T

`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.

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();
```

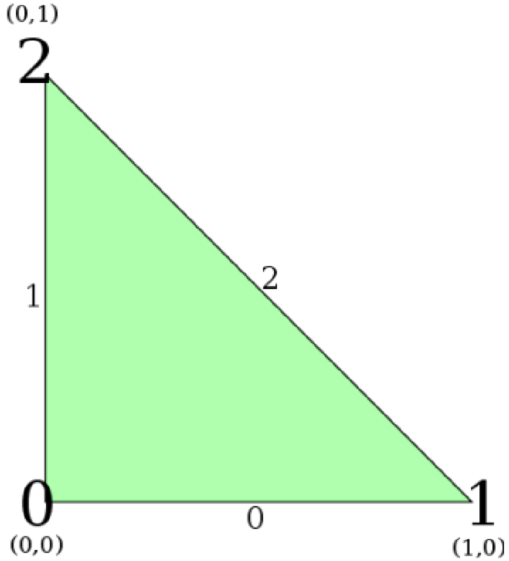
- The grid can create a new Entity object from an EntitySeed:

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

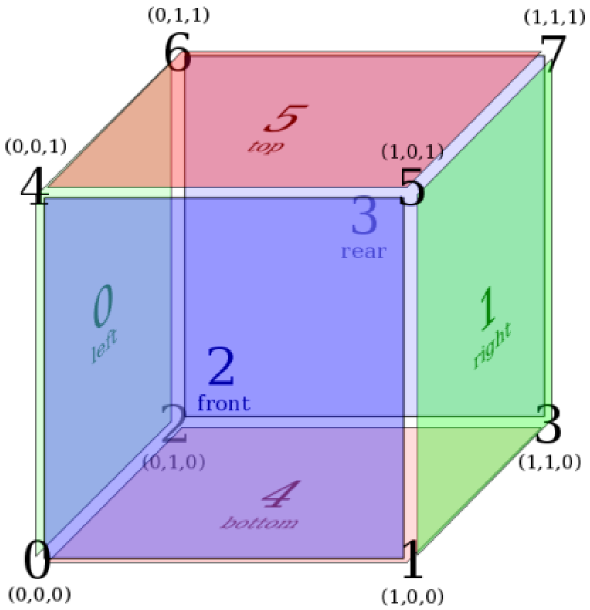
- Memory efficient, but run-time overhead to recreate entity.

Reference Elements

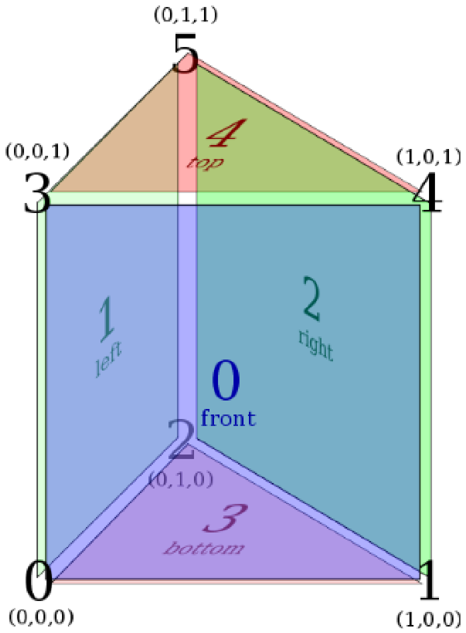
`Dune::GeometryType` identifies the type of the entity's reference element.
`cell.type()` returns the GeometryType of an entity.



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`:

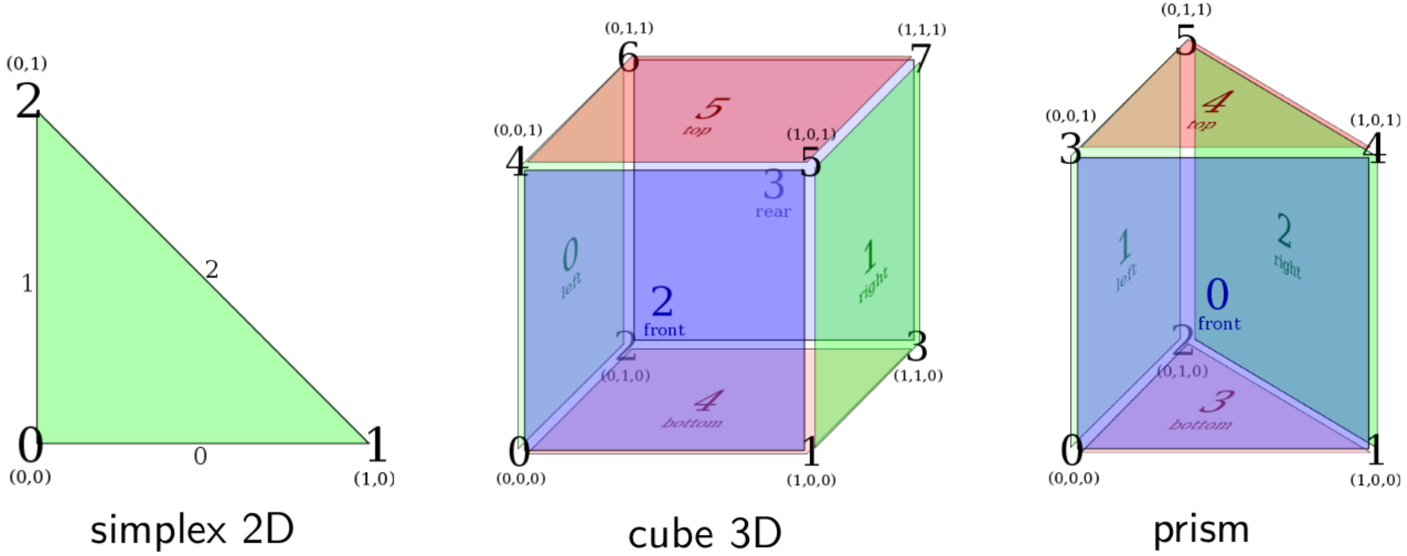
```
#include <dune/geometry/type.hh>
...
namespace GeometryTypes = Dune::GeometryTypes;
Dune::GeometryType gt;

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:



- 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

```
#include <dune/geometry/referenceelement.hh>
...
Dune::ReferenceElement<double, dim> ref_el = ...;
```

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

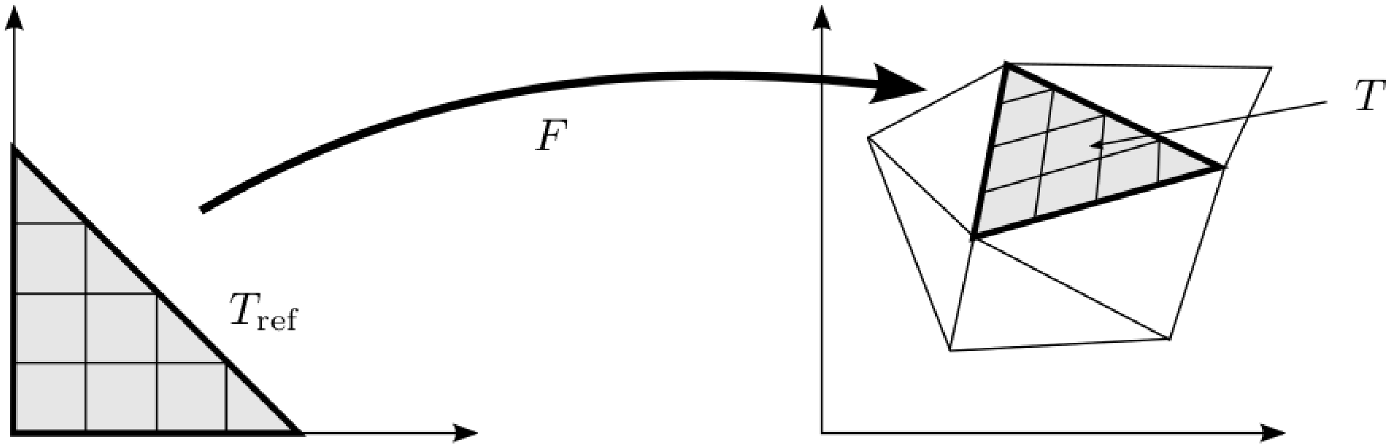
```
auto ref_el = Dune::referenceElement(entity.geometry()); // or
auto ref_el = Dune::referenceElement(entity);
```

  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)

Geometry

Transformation F_T

- 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_F^{-T}.
- Gramian determinant of transformation Jacobian for quadrature.



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 (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);
```

- 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 (II)

- 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 T, i.e., $f(F_T(\hat{x}))$.

The gradient of f is then given by

$$J_F^{-T}(\hat{x})\hat{\nabla} f(F_T(\hat{x}))$$

```
auto x_global = geo.global(x_local);
auto J_inv = geo.jacobianInverseTransposed(x_local);
auto tmp = grad(f)(x_global); // grad(f) supplied by user
<grad_type> gradient;         // something like FieldVector<double,dimworld>
J_inv.mv(tmp, gradient);
```

Obtaining Quadrature Rules

Numerical quadrature rules given by

$$\int_{T_{\mathrm{ref}}} \hat{f}(\hat{x})\,\mathrm{d}\hat{x} \approx \sum_{i=0}^{N} w_i \hat{f}(\hat{x}_i)$$

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

```
#include <dune/geometry/quadraturerules.hh>
...
Dune::GeometryType gt = ...;
auto const& rule = 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, e.g. `Entity::mydimension`
- The rule is exact for polynomials up to the given `order`
- Use `auto const&` 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`s.
- `QuadraturePoint` provides weight and position:
  - `QuadraturePoint::weight()`
  - `QuadraturePoint::position()`

Using Quadrature Rules

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

Example

```
auto fLocal = some_function_to_integrate(...);
double integral = 0.0;
auto geo = cell.geometry();
for (const auto& qp : rule)
{
  integral += fLocal(qp.position()) * qp.weight() * geo.integrationElement(qp.position());
}
```

Using Quadrature Rules

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

Example

```
auto fLocal = some_function_to_integrate(...);
double integral = 0.0;
auto geo = cell.geometry();
for (const auto& qp : rule)
{
  integral += fLocal(qp.position()) * qp.weight() * geo.integrationElement(qp.position());
}
```

> **Attention:** When integrating over cells in a grid, keep in mind that the quadrature point coordinates are local to the reference element.

Exercise 2

Exercise 2

1. Create a new dune module `dune-grid-exercise` using the tool `duneproject`
2. Create a structured grid for the domain $\Omega = [0,2] \times [0,2]$ with 8 elements in each direction. Therefore, use either `YaspGrid` or `SPGrid`.
3. Implement a function that computes the numerical integration of a given function $f : \Omega \to \mathbb{R}$ over the domain Ω. Therefore, traverse the grid and create a `QuadratureRule` on each grid entity. For each quadrature point, evaluate the function f in global coordinates by mapping element-local coordinates using the element geometry.

Exercise 2

Concrete setup:

- Compute the numerical quadrature of the function

$$f(x) = \sin(x_0) \cdot \cos(x_1)$$