# The DUNE Grid Interface
# An Introduction

Christian Engwer[1], <u>Robert Klöfkorn</u>[2], Andreas Dedner[3]

[1] Applied Mathematics, WWU Münster
[2] Center for Mathematical Sciences, Lund University
[3] Mathematics Institute, Warwick University

# 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.
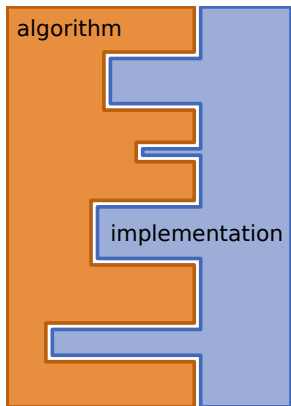
# Flexibility
**Seperate data structures and algorithms.**

- ▶ 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.
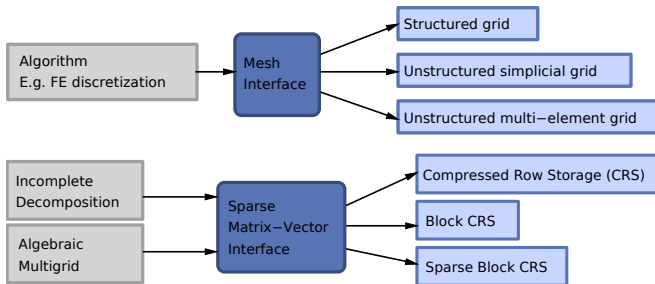
# Efficiency

**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.
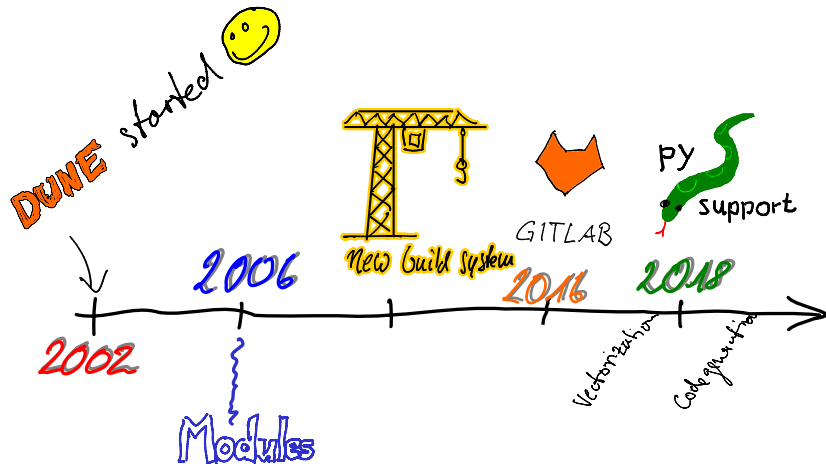
# Contents

# 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
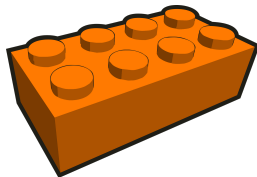- ▶ Open Development Process
- ▶ Free Software Licence



---

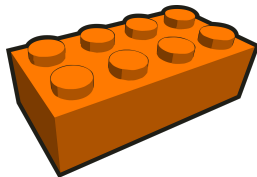[1]Dune review paper: https://doi.org/10.1016/j.camwa.2020.06.007
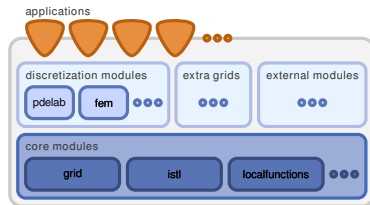
# Some historic remarks

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

- ▶ **Additional Grid Implementations**

- ▶ **Extension Modules**

# DUN(E)iverse



- modular structure

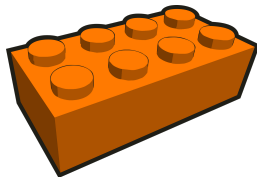- write your own DUNE modules

- available under different licenses

- **Discretization Modules**

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

- **Additional Grid Implementations**

- **Extension Modules**

# DUN(E)iverse



- ▶ modular structure

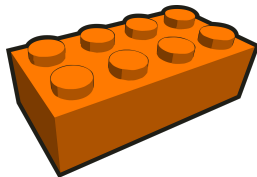- ▶ write your own DUNE modules

- ▶ available under different licenses

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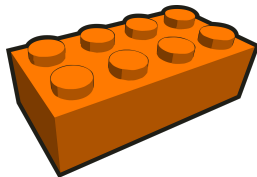
- ▶ **Additional Grid Implementations**

  **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.
  **opm-grid:** 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-metagrid** | meta grids for DUNE |
| **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**
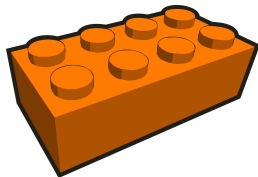
**Goals:** allow people to. . .

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

# 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.
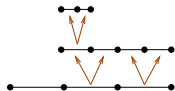
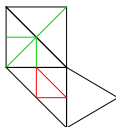# Designed to support a wide range of Grids
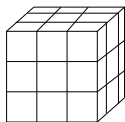

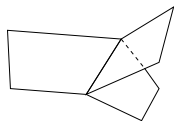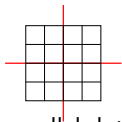
structured
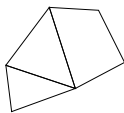


conforming



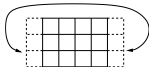non conforming



nested, 1D


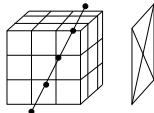
red-green, bisektion



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.*
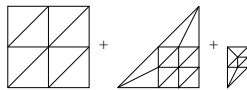


Hierarchic Grid

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

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

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

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



- ▶ vertices ,
- ▶ edges ,
- ▶ faces ,
- ▶ elements , . . .

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 ,
- ▶ elements , . . .

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 $= 1$)*,
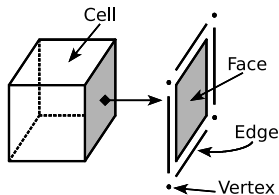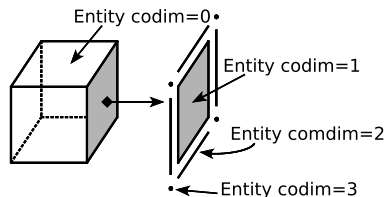- ▶ elements *(Entity codim $= 0$)*, . . .

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

```python
dim = 1,2,3,...

from dune.grid import yaspGrid as hierarchicGrid
domain = cartesianDomain([0]*dim, [1]*dim, [4]*dim)
view = hierarchicGrid(domain)

# using a dictionary with vertices and elements
from dune.alugrid import aluConformGrid as leafGridView
mesh = { "vertices":[ [ 0, 0], [ 1, 0], [ 1, 1],
                      [-1, 1], [-1,-1], [ 0,-1] ],
  "simplices": [ [2,0,1], [2,3,0], [3,4,0], [0,4,5] ] }
view = leafGridView(constructor=mesh)

# or using a grid reader
domain = (reader.gmsh, "mymesh.msh")
view = leafGridView( domain, dimgrid=2 )
```

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

# The DUNE Grid Interface

The DUNE Grid Interface is a collection of classes and methods

```
dim = 1,2,3,...

from dune.grid import yaspGrid as hierarchicGrid
domain = cartesianDomain([0]*dim, [1]*dim, [4]*dim)
view = hierarchicGrid(domain)

# using a dictionary with vertices and elements
from dune.alugrid import aluConformGrid as leafGridView
mesh = { "vertices":[ [ 0, 0], [ 1, 0], [ 1, 1],
                      [-1, 1], [-1,-1], [ 0,-1] ],
  "simplices": [ [2,0,1], [2,3,0], [3,4,0], [0,4,5] ] }
view = leafGridView(constructor=mesh)

# or using a grid reader
domain = (reader.gmsh, "mymesh.msh")
view = leafGridView( domain, dimgrid=2 )
```

*We will now get to know the most important concepts 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:



- ▶ LeafGridView: `grid.leafView`

- ▶ LevelGridView:
  `grid.levelView`

**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:



▶ LeafGridView: `grid.leafView`

▶ LevelGridView:
  `grid.levelView`

**levelwise:**
all entities associated with the same
level.
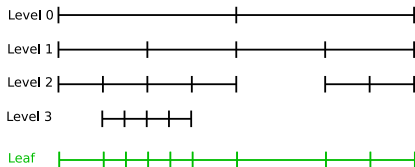*Note: not all levels must cover the
whole domain.*
**leafwise:**
all leaf entities (entities which are
not refined).
The leaf view can be seen as the
projection of a levels onto a flat
grid. It again covers the whole
domain.

# Views to the Grid

A Grid offers two major views:



▶ LeafGridView: `grid.leafView`

▶ LevelGridView:
  `grid.levelView`

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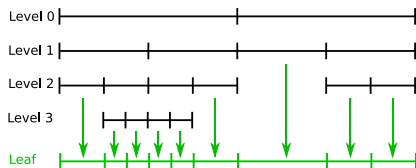*Note: not all levels must cover the whole domain.*

**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:



- ▶ LeafGridView: `grid.leafView`
- ▶ LevelGridView:
  `grid.levelView`

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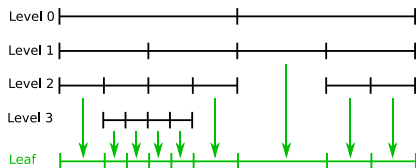*Note: not all levels must cover the whole domain.*

**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.

# Plotting

```python
# using a dictionary with vertices and elements
from dune.alugrid import aluConformGrid as leafGridView
mesh = { "vertices":[ [ 0, 0], [ 1, 0], [ 1, 1],
                      [-1, 1], [-1,-1], [ 0,-1] ],
  "simplices": [ [2,0,1], [2,3,0], [3,4,0], [0,4,5] ] }
view = leafGridView(constructor=mesh)

view.plot()
```

# Plotting

```python
# using a dictionary with vertices and elements
from dune.alugrid import aluConformGrid as leafGridView
mesh = { "vertices":[ [ 0, 0], [ 1, 0], [ 1, 1],
                      [-1, 1], [-1,-1], [ 0,-1] ],
  "simplices": [ [2,0,1], [2,3,0], [3,4,0], [0,4,5] ] }
view = leafGridView(constructor=mesh)

view.plot()
```
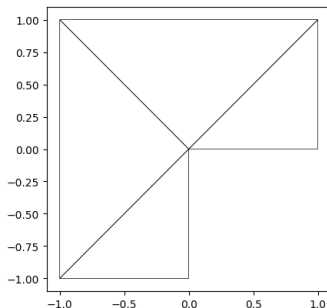
# Iterating over grid entities

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

▶ `GridView` supports iteration over all entities of one codimension.

▶ Generation of iterators looks like this:

```python
for entity in view.entities(codim=0):
    # do something with entity (element in this
        case)
```

▶ More specialized like this:

```python
for element in view.elements:
    # do something with element
```

# Contents

# Entities

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

```
for element in view.elements:
  element.type # what else 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 element in view.elements:
  element.type # what else 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 element in view.elements:
  element.type # what else 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.

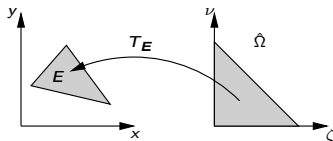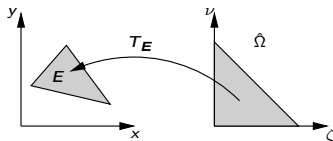`gridView.entities(c)` implement the entity concept.

# 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.entities(c)` implement 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.entities(c)` implement 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:

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

▶ `codim(element)== 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:

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

- ▶ `codim(element)== 0`
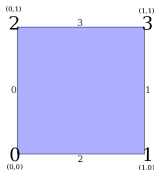- ▶ `codim(face)== 1`

- ▶ `...`
- ▶ `codim(vertex)== dim(grid)`
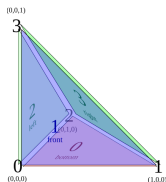
# Reference Elements

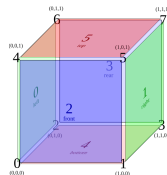`entity.type`: provides the type of the entity's reference element[2]
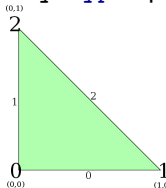


triangle  quadrilateral  tetrahedron  hexahedron

---

[2]https://dune-project.org/doxygen/master/group_
_GeometryReferenceElements.html

# Geometry Types

A geometry type is a simple identifier for a reference element

▶ Obtain from entity or geometry object using `.type`

▶ Geometry type for specific reference elements

```python
import dune.geometry
geometryType = dune.geometry.simplex(2)
referenceElement =
    dune.geometry.referenceElement(geometryType)
```

▶ Geometry type's are small objects, cheap to store and pass
around

# ReferenceElement (I)

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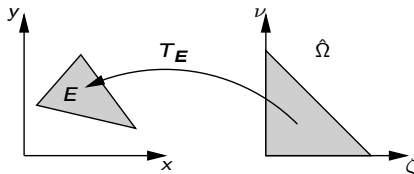


simplex 2D      cube 3D      prism
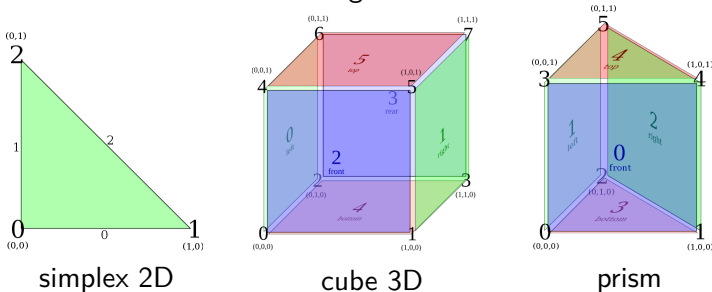
▶ Numbering of subentities within the reference element
▶ Geometrical mappings from reference elements of subentities to the current reference element

# 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

```
geo = entity.geometry
```

▶ Convert local coordinate to global coordinate

```
xGlobal = geo.toGlobal(xLocal)
```

▶ Convert global coordinate to local coordinate

```
xLocal = geo.toLocal(xGlobal)
```

▶ Get center of geometry in global coordinates

```
center = geo.center
```

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

```
n = geo.corners
```

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

```
cornerGlobal = geo.corner
```

# Geometry Interface (II)

▶ Get type of reference element

```
type = geo.type # quadrilateral, triangle, ...
```

▶ Find out whether geometry is affine

```
if geo.affine:
  # do something
```

▶ Get volume of geometry in global coordinate system

```
volume = geo.volume
```

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

```
mu = geo.integrationElement(xLocal)
```

# GridFunction

```
view = leafGridView(domain)
from dune.fem.function import gridFunction
from numpy import sin, log, pi

@gridFunction(view)
def f1(y):
    return sin(20*y[0]*y[1])*y.two_norm
f1.plot()

@gridFunction(view)
def f2(element,localx):
    y = element.geometry.toGlobal(localx)
    return sin(20*y[0]*y[1])*y.two_norm
f2.plot()

@gridFunction(view)
def center(element,localx):
    return element.geometry.center.two_norm
center.plot()
```

# Plotting

```python
# plotting using matplotlib (only single core)
center.plot()

# write a vtk file (works for parallel runs)
view.writeVTK("name", pointdata=[f1,f2],
                      celldata=[center])

# create a writer instance (works for parallel runs)
vtk = view.sequenceVTK("name", pointdata=[f1,f2],
                 celldata=[center], subsampling=1)

vtk()
...
vtk()
```

vtk files can be visualized with paraview[3] or other tools.

--------

[3]https://paraview.org

# Gradient Transformation

Assume
$$f : \Omega \to \mathbb{R}$$
evaluated on a element $E$, i.e. $f(T_E(\hat{x}))$.

The gradient of $f$ is then given by
$$J_T^{-T}(\hat{x})\hat{\nabla}f(T_E(\hat{x})) :$$

```
xGlobal = geo.toGlobal(xLocal) # or p.position
Jinv = geo.jacobianInverseTransposed(xLocal);
tmp = gradF(xGlobal) # gradF supplied by user
gradient = Jinv @ tmp
```

# 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

$$E_g^c = \{ e \in E \mid e \text{ has codimension } c \text{ and geometry type } g \}.$$
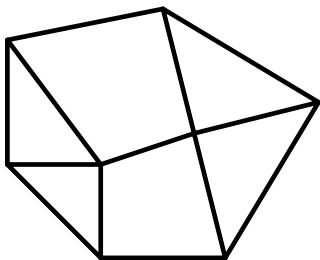
- ▶ unique within the subsets $E_g^c$.
- ▶ consecutive and zero-starting within the subsets $E_g^c$.
- ▶ distinct leaf and a level index.

**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.
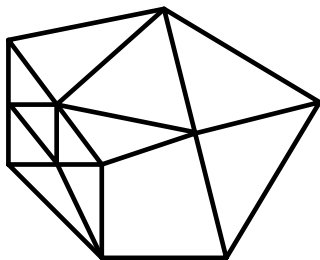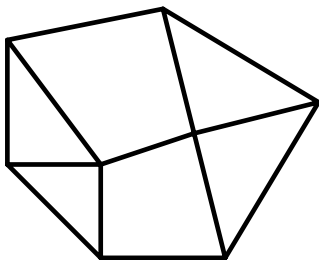- ▶ persistent with respect to grid modifications.

# 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: 2D Multi-Element Grid – Indices

**Locally refined grid:**

**Locally refined grid:**

**Locally refined grid:**

*Indices:*



Consecutive index for vertices
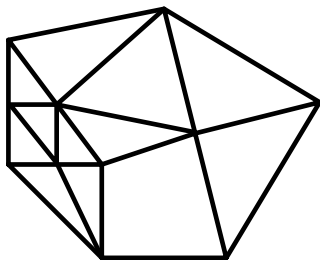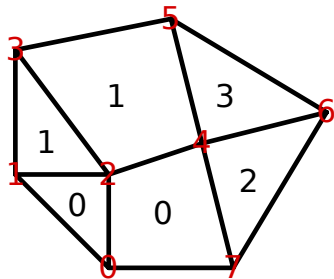
**Locally refined grid:**

*Indices:*



. . . and cells

**Locally refined grid:**
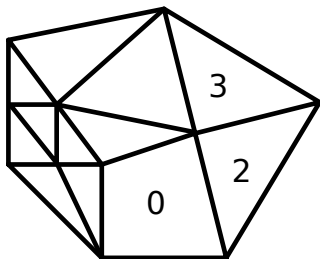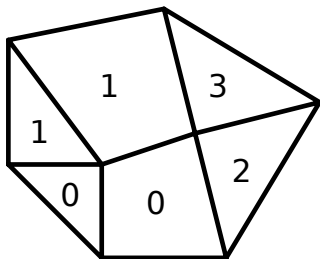
*Indices:*



Old cell indices on refined grid

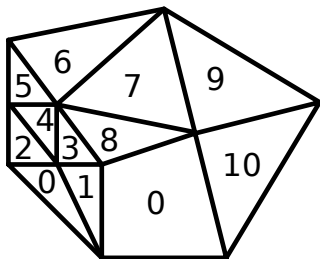**Locally refined grid:**

*Indices:*



Consecutive cell indices on coarse and refined grid

# Example: 2D Multi-Element Grid – Indices

**Locally refined grid:**

*Ids:*



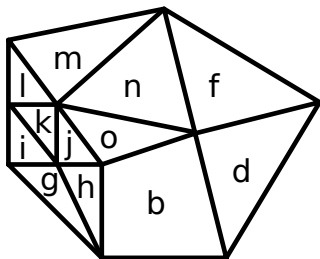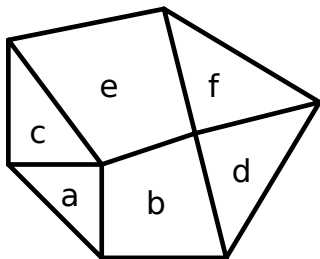Persistent Ids on coarse and refined grid

# 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 `MultipleCodimMultipleGeomTypeMapper` accessed from the grid view by

```
mapper = view.mapper(lambda gt: gt.dim ==
    view.dimension)
```

# 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 `MultipleCodimMultipleGeomTypeMapper` accessed from the grid view by

```
mapper = view.mapper(lambda gt: gt.dim ==
    view.dimension)
```

# Example: Mapper (I)

```python
# create a mapper to assign indices to elements
# in order to attach data to grid entities
mapper = view.mapper(lambda gt: 1 if gt.dim == 0
    or gt.dim == view.dimension else 0 )

# iterate over all cells
for element in view.elements:
  elemIdx = mapper.index( element )

  # iterate over all vertices of an element
  for vertex in element.vertices:
    vxIdx = mapper.index( vertex )
```

# 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, M. Dedner, N.-A. Dreier, R. Engwer, Ch. Fritze, C. Gräser, Ch. Grüninger, D. Kempf, R. Klöfkorn, M. Ohlberger, and O. Sander.
The Dune framework: Basic concepts and recent developments
*CAMWA* 81, 2021, https://doi.org/10.1016/j.camwa.2020.06.007.

A. Dedner, R. Klöfkorn.
Python Bindings for the DUNE-FEM module: Tutorial.
*https://dune-project.org/sphinx/content/sphinx/dune-fem/*

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.