Dune Interface Change Request

Generalizing intersections for non-manifold grids

Oliver Sander

October 2, 2015

This text documents a request to change an interface in the `dune-grid` module. It is intended to describe the new interface with as much precision as possible. If the proposal is accepted, the document will be the reference on all syntactical and semantical questions.

Contents

1 Problem statement

In the following we call the $d-1$-dimensional faces of a d-dimensional element *facets*.

The current grid interface makes the implicit assumption that at each point of the interior of an element facet, the element either intersects with one other element, or the point is on the domain boundary. As a consequence, configurations as the one in Figure 1 are not explicitly catered for.

However, such configurations are useful in various situations.

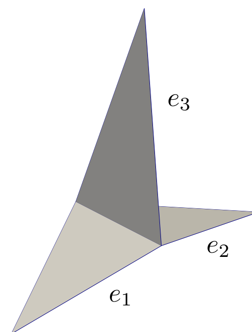

Figure 1: Grid with T-junction

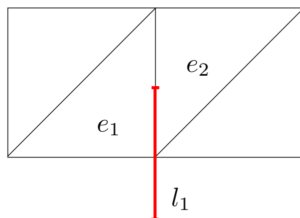

Figure 2: Ambiguous case occurring in a `dune-grid-glue` coupling between a 1d grid consisting of a single element, and a 2d triangle grid. Strictly speaking, the line element l_1 intersects *two* triangle elements, viz. e_1 and e_2, but this cannot be represented by the current intersection concept.

1.1 Network grids

Both one- and two-dimensional networks with fairly general topology appear as domains for PDEs in various applications. Examples are:

- 1d: Network flow and diffusion problems; with applications in geology, modeling of traffic and supply chains, neuroscience, root networks, etc.
- 2d: flows and transport in rock fracture networks, mechanics of closed-cell foams, etc.

In such networks, three or more lines may meet in a single point (1d networks), and three or more two-dimensional sheets may meet in a common line (2d networks).

Grids are needed for finite element and finite volume methods on such network domains. The `FoamGrid` grid manager[1] is an implementation of the DUNE grid interface for network grids.

1.2 dune-grid-glue

A closely related problem concerns `dune-grid-glue`. There, intersections are a central concept. However, compared to the intersections from the DUNE grid interface, the intersections in `dune-grid-glue` are generalized in two aspects:

1. Intersections relate elements from two different grids.
2. The two elements may have different dimensions, and the intersection need not have codimension 1 in either grid.

This generalized intersection concept works very nicely in many situations. However, in certain situations the concept is not general enough. The following example makes this more concrete: Suppose a two-dimensional triangle grid is to be coupled to a one-dimensional grid embedded in the triangle grid. In such a situation, an element of the

[1] users.dune-project.org/projects/dune-foamgrid

1d grid may intersect the edge between two triangles in more than one point (Figure 2). In other words it has a non-zero (relative to its own topology) intersection with either of the two triangles. This intersection must be represented by an Intersection object, of which the 1d element is the inside element. But which of the two triangles is the outside element? Currently there can be only one outside element, but picking one instead of the other is arbitrary.

2 Status quo

While there is no explicit support in the DUNE grid interface for network grids, such grids can nevertheless be implemented in DUNE by violating an assumption of the interface. This is what the current `FoamGrid` implementation does. The grid interface makes the unwritten assumption that for all intersections of a given element e, the corresponding geometries-in-inside form a nonoverlapping partition of the element boundary, modulo sets of measure zero.

`FoamGrid` currently violates this assumption. Instead, on the grid of Figure 1 and element e_1, the intersection iterator will stop *four* times. Of the four intersections, two are with the boundary. The other two are with elements e_2 and e_3, respectively. These two have identical geometries-in-inside.

Advantages: Implementing `FoamGrid` in this way, without explicit changes to the grid interface, has several advantages:

- No interface changes
- An important concept of the DUNE grid interface is retained: an intersection relates *two* elements (see Appendix **??** for a possible alternative to this).
- Allows to handle nonconforming situations such as the one in Figure **??** without having to artificially split intersections.
- It is possible to reach all neighbors of a given element with a single loop.

Disadvantages:

- There is no way to find out which intersections "belong together".
- In situations as the one in Figure 2, functions on the 1d element frequently appear as singular source terms for the 2d grid. However, in analogy to `FoamGrid`, `dune-grid-glue` currently does produce two intersections (one for each triangle). This implies that the source term will actually be added twice.
- The grid test for `FoamGrid` fails: it tests for each affine element whether the integrationOuterNormals of all intersections sum up to zero. This property is not true for the configuration in Figure 1: the normals for the non-boundary edge are added twice.

- What is worse: the test cannot be fixed. Natural fixes would be to either scale the normal at each integration point with the number of intersections at this point, or to integrate only over a subset of all intersections. However, the required information is not available.

3 Proposed changes

We propose the following changes to the DUNE grid interface. There are both semantical changes and changes to one interface method signature.

3.1 Semantic changes

We propose that the following rules shall be part of the DUNE grid interface:

Rule 1 *For any two intersections of a given element, the geometry-in-insides are either disjoint or identical.*

Consequences:

- For each intersection, there is a well-defined number that says how many intersections of the element have the same geometry-in-inside as that intersection. Expressed differently: there is a well-defined number that says how many neighbors the element has across a given intersection.
- For non-conforming situations like the one in Figure **??**, the intersection that relates e_1 and e_2 *must* be broken into two parts, to be consistent with the intersection for $\{e_1, e_3\}$.

The following two rules allow to find out which intersections belong to a common geometry-in-inside.

Rule 2 *If an intersection between two elements is split up into several parts, then the intersection iterator shall traverse them consecutively.*

Rule 3 *If more than one neighbor is reachable over a common geometry-in-inside, then all intersections for this geometry-in-inside shall be traversed consecutively.*

The following two rules should be rather obvious, but we state them anyway for precision.

Rule 4 *If an element has two intersections that share a common geometry-in-inside, then neither of them is a boundary intersection.*

Rule 5 *If an element has two intersections that share a common geometry-in-inside, then the two respective outside elements are not the same.*

3.2 Changes to the Intersection interface class

Only a single method is to be changed.

3.2.1 Neighbor

Change the signature of the Intersection :: neighbor method from

```
bool neighbor () const;
```

to

```
std::size_t neighbor () const;
```

The current version returns **true** if the inside element has a neighbor across the intersection, or, in other words, if there is a valid outside element. The new version should return the *number* of different outside elements across intersections that share the current geometry-in-inside.

3.3 Discussion

All changes are fully backward compatible. In particular, the change to the neighbor method is backward-compatible: Users of grids without multiple intersections will start to receive 1 and 0 instead of **true** and **false**, which is no problem because 1 and 0 are automatically cast to the correct boolean values.

Advantages:

- No backward-incompatible changes
- Important interface concept retained: an intersection object still relates exactly two elements.
- All neighbors of an element can be visited in a single loop.
- The test whether integrationOuterNormals of affine elements sum up to zero can be fixed: while integrating, simply scale each normal with $1/\max\{1, neighbors()\}$.

A Appendix: Alternative approach

There is a second approach to solve the same problem. This approach was discussed for a while, and then discarded in favor of the approach presented above. It is left in this appendix for the curious reader.

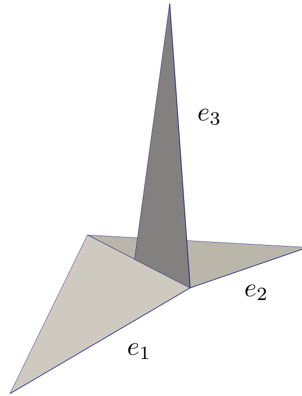

Figure 3: Grid with nonconforming T-junction. Suppose e_1 is the *inside* element. With the current grid interface, a hypothetical nonconforming network grid manager could represent this situation by two intersections $\{e_1, e_2\}$ and $\{e_1, e_3\}$, where the geometry-in-inside fo $\{e_1, e_3\}$ is a true subset of the one for $\{e_1, e_2\}$. When allowing intersections to relate more than two elements, then this configuration must be represented by two intersections $\{e_1, e_2\}$ and $\{e_1, e_2, e_3\}$.

A.1 Semantic changes

Rule 6 *Intersections cease to be objects that relate* pairs *of elements. Rather, they now become objects that relate* groups *of elements.*

- Each intersection still has only one intersectionInInside. This implies that non-conforming configurations as the one in Figure **??**, can only be handled by using two intersections for the common edge.
- There is more than one outside element, each with corresponding geometry-InOutside and indexInOutside

A.2 Changes to the Intersection interface class

A.2.1 Neighbor

Change the signature of the neighbor method from

```
bool neighbor () const
```

to

```
std::size_t neighbor () const
```

The current version returns **true** if the inside element has a neighbor across the intersection, or, in other words, if there is a valid outside element. The new version should return the *number* of outside elements.

A.2.2 Outside

Change the signature of the outside method from

```
Entity outside () const
```

to

```
Entity outside (std :: size_t i=0) const
```

Rather than returning the unique outside element, the method now returns the i-th outside element.

A.2.3 geometryInOutside

Change the signature of the geometryInOutside method from

```
LocalGeometry geometryInOutside () const
```

to

```
LocalGeometry geometryInOutside (std::size_t i=0) const
```

Rather than returning the unique geometry of the intersection in the outside element, the method now returns the geometry of the intersection in the i-th outside element.

A.2.4 indexInOutside

Change the signature of the geometryInOutside method from

```
int indexInOutside () const
```

to

```
int indexInOutside (std:: size_t i=0) const
```

Rather than returning the unique index of the intersection in the outside element, the method now returns the index of the intersection in the i-th outside element.

A.3 Discussion

The changes to the neighbor method is fully backward compatible. Users of grids without multiple intersections will start to receive 1 and 0 instead of **true** and **false**, which is no problem because 1 and 0 are automatically cast to the correct boolean values.

Also, the changes to the outside method (etc.) is fully backward compatible. In grids without multiple intersections, at most the 0-th outside element will be available. The default parameter ensures that this intersection will be returned when the method is called without argument.

Advantages:

- Retain the rule that the geometries-in-inside must form a disjoint partition of the element boundary (modulo zero-sets).
- Fully backward-compatible. No existing code needs to be changed.

Disadvantages:

- To iterate over all outside elements, two nested loops are needed, instead of only one.