Proceedings of the 17th GAMM-Seminar Leipzig 2001, pp. 1–28

# A Generic Toolbox for the Grid Craftsman

Guntram Berti
Institut für Mathematik
BTU Cottbus, Germany

June 19, 2001

**Abstract**

Universally reusable tools for grid management tasks are scarce. We identify coupling of algorithms to data structures as main obstacle for reuse, and show how to overcome the difficulties by using generic programming. After introducing an abstract kernel of grid functionality, we present some universal generic grid tools based on that kernel which are usable for arbitrary grid data structures. For evaluating the approach, we describe how to leverage these tools in order to set up a framework for hybrid grid generation. Finally, we show how to use generic components with existing grid data structures, and discuss the efficiency of generic grid tools, which is overall quite satisfying.

## 1 The Grid Craftsman's Old Tools

Computational scientists who are in charge of constructing and processing grids for scientific applications are in need of a vast amount of different support tools.

First spring to mind basic tasks, such as reading and writing grids from and to files in different formats, including compressed ones, converting between various grid representations, or calculating additional incidence information.

In the grid generation process, one often has to deal with different generation procedures, producing grids which have to be combined afterwards. This calls for methods to find overlapping parts of different grids, to glue grids together or to cut parts off a grid, to efficiently locate points in grids, or to coarsen and refine a grid according to some criterion.

Then, the validity of the resulting has to be ensured (looking for self-intersections etc.), its quality has to be monitored and enhanced if necessary. For these tasks, adequate visualization of critical grid parts and their quality measures can be of great help, especially in more involved situations.

The list could obviously be continued. We did intentionally not list more domain-specific tasks such as numerical discretizations, which merely work *on* grids and not *with* grids, and thus do not properly fit into the toolbox we are concerned with here. However, the approach we are going to present applies to them as well.

Given the large number of more-or-less routine tasks which need to be performed by a grid craftsman, one might expect a well established set of standard tools to perform these tasks. However, the situation is different: The "standard toolbox" is almost empty.

Why is this claim justified, and what are the reasons for this situation? It is true, there exist a large number of tools for these tasks. But a *standard* tool, by definition, has to be *independent* of concrete details or contexts. We can use a standard hammer to drive a nail into the wall, regardless whether it is a copper nail or a steel nail. In contrast, almost no existing tool for the one of the grid tasks described before can used in such a general way: The typical tool is very tightly coupled to the internals of a specific grid representation. It is thus buried in a specific context; even changing the underlying grid data structure a bit often causes major headaches, let alone the effort required to make it work with a different representation.

Some degree of independence is achieved only by tools operating on some "standard" (or not-so-standard) file format. This approach, however, has too many drawbacks to be considered a general solution: First, there are performance issues: I/O is notoriously slow, and if the task involves local operation on a small part of the grid only, or the grid is represented implicitely (e. g. Cartesian), it is extremely wasteful. When chaining several operations, I/O quickly becomes a bottleneck. Also, if the grid has to be changed by the tool, the approach leads to difficulties. And finally, there is not format which can account for all types of grids; so, the notion of a "standard file format" itself is a chimera – think of chimera-type grids.

Closely related is the approach taken by libraries such as LAPACK [1], which operate on a set of standard data structures. This works more or less well for domains with data structures exhibiting comparatively low variability, such as dense matrices; but already there it leads to a considerable amount of redundant functionality. It does not scale to areas with highly variable representations, which is the case for grid data structures.

In the next section, we will explore a novel technique, generic programming, which overcomes the limitations of current approaches for designing libraries of grid tools. In section 2.2, we present a minimal kernel of grid functionality allowing to decouple algorithms from data structures. Section 3 gives a sample of reusable generic tools. In section 4, the approach is put into practice, implementing a framework for generating hybrid grids, using generic components as much as possible. Practical issues concerning the use of generic components and their efficiency are dealt with in section 5. Finally, we discuss the overall merits of our approach.

# 2 The Grid Craftsman's New Tools

## 2.1 The Problem

We have already stated that the core of the problem are the close links between grid data structures and implementations of algorithms operating on them. To illustrate this issue, lets look at a simple, yet instructive, example. The following algorithm calculates and stores, the surface area (or perimeter in 2D) for each cell in a grid $\mathcal{G}$:

```
**OUT:** surface: $\mathcal{G}^d \mapsto \mathbb{R}$
```

```
**for all** Cells $c \in \mathcal{G}^d$ **do**
  surface($c$) = 0
  **for all** Facets $f$ of $C$ **do**
    surface(c) += volume($f$)
```

Here $d$ is the dimension of $\mathcal{G}$, and $\mathcal{G}^d$ denotes the set of $d$-dimensional elements of $\mathcal{G}$, that is, the cells. Furthermore, a facet is an element of co-dimension 1, that is, an edge in 2D and a face in 3D (see also section 2.2.1).

The important point here is the generality of this algorithm: It abstracts from any representation details, and therefore works for any grid, regardless of its dimension, or whether it is Cartesian, simplicial or something different.

Now consider what happens if we implement this algorithm for a concrete representation. Let us assume we have a simple data structure for a 2D triangulation where cell-vertex incidences are stored in a plain array `cells`, such that `cells[3*c +v]` gives the index of the `v`th vertex of cell `c`. The array `geom` is assumed to hold the coordinates of vertex `v`:

```
double * surf = new double[nc];
for(int c = 0; c < nc; ++c) {
  surf[c] = 0.0;
  for(vc = 0; vc < 3; ++vc) {
    int vc1 = (vc+1)%3;
    double dx = (geom[2*cells[3*c+vc ]   ] - geom[2*cells[3*c+vc1]   ]);
    double dy = (geom[2*cells[3*c+vc ]+1 ] - geom[2*cells[3*c+vc1] +1]);
    surf[c] +=  sqrt(dx*dx+dy*dy);
  }
}
```

This implementation loses all of the generality of the abstract algorithm, and is therefore not usable for other types of grids. The situation is typical for current grid tools: Specialized to one concrete grid representation, much less general than the underlying algorithm, and therefore not reusable in different contexts, that is, for different grid data structures. While this might not be considered a great loss in the specific case at hand, it is clear that reuse of more complicated algorithms across a wide variety of grid representations would greatly enhance productivity.

The problem being the close relationship between data structures and algorithms implementations, the "obvious" way to go is to separate them. We must somehow avoid the *over-specification* present in the implementation shown before, and restrict ourselves to using only information *intrinsic* to the notion of a grid, that is, using nothing which is related to arbitrary representation details.

A technique trying to abstract from representational issues is known as *generic programming* [14] and has been popularized recently by the Standard Template Library (STL) [10], which is now part of the C++ standard. Among the languages used for scientific software, C++ now has by far the most sophisticated support for generic programmming. It has therefore been chosen for the Grid Algorithms Library GrAL [5], which is a freely available testbed for the ideas described in this paper.

Now, what is the intrinsic information on grids an algorithm might use? To get an idea, let us have a look at the needs of the surface algorithm:

1. iteration over all cells of a grid (combinatoric)

2. iteration over all facets of a cell (combinatoric)
3. area of facets (geometric)
4. store real numbers on cells (grid function)

By analyzing different algorithms, we come to similar results [4]. The required functionality falls into three large groups:

1. Combinatorial (mainly iteration over elements)
2. Geometric (coordinates, centers, areas, ... )
3. Grid functions (associating data to grid elements)

In the following, we will very briefly present a functional kernel for grids which covers these three areas and has proven sufficient to support a large group of algorithms, most notably non-mutating algorithms. In section 2.2.5, we introduce some coarse-grained mutating primitives which help to implement mutating algorithms in a generic way, too.

To give a short preview of how this looks in action, here is the generic implementation of the surface algorithm:

```
grid_function<Cell,double> surface(Grid);
for(CellIterator c(Grid); !c.IsDone(); ++c) {
  surface[c] = 0.0;
  for(FacetOnCellIterator f(c); !f.IsDone(); ++f)
    surface[c] += Geometry.volume(f);
}
```

We see that this implementation is as general as the abstract algorithm. In fact, there is a one-to-one correspondence between the generic C++ code and the pseudocode. The primitives used here can be implemented for virtually any concrete grid representation; in section 5.1 we sketch an implementation for the triangle data structure described above.

Now that we have an idea how the generic approach works in principle, we can list some requirements on a functional kernel we would like to meet. First, it must capture the mathematical properties of grid reasonably well: If it does not allow to get the information inherent in a mathematical grid underlying a concrete representation, it is to weak. On the other hand, because it has to be implemented for each concrete grid data structure, it should be minimal and largely orthogonal. Hence, we must strive for a balance between completeness, minimality, efficiency and expressivity. For example, if we know that the cells are triangles (simplices), we can answer questions of the form "Give me the vertex opposite to this facet". Such queries have been omitted from the kernel for two reason: First, they assume a special type of grid (simplicial), and second, it turns one that information that detailed is only seldom needed (mostly by certain refinement strategies and higher-order FEM discretizations). The required functionality should be added in a separate layer.

An important point is that the kernel must allow for efficient implementations — too inefficient generic algorithms will not be used and are therefore pointless. Vice versa, a grid component will typically implement only those parts of the kernel which can meet reasonable efficiency requirements. For example, in the triangulation example, if `cells` are the only data stored, we cannot efficiently access neighbor cells, and hence will not implement the corresponding concept of the kernel (CellOnCellIterator). If we need this information, we can either switch to a more complete data structure, or calculate the information ad hoc, using the generic neighbor search algorithm presented in section 3.1.

## 2.2 A Functional Kernel for Grids

### 2.2.1 What Is a Grid?

What is a grid, then? Before going into the details of developing a functional kernel for grids, we have to be explicit about this question. In fact, there are many definitions in use, which differ in smaller or larger details. Being independent of any concrete grid representation, we cannot fall back to the pragmatic view that a grid is what our data structure can represent (which, besides, is often quite different from what one naively would expect).

It turns out that grids exhibit a rich variability with respect to their mathematical structure, affecting both algorithms and data structures. See [4] for more detailed information than we can present here. In order to come up with definitions for these mathematical grids, we will heavily exploit the body of knowledge built up by the field of combinatorial topology. The central notion is that of an *abstract complex*:[1]

**Definition 1 (Abstract complex)** *An abstract finite complex $\mathcal{C}$ is a set of* elements $e$*, together with a mapping* $\dim : \mathcal{C} \mapsto \{0, \ldots, d\} \subset \mathbb{N}$*, (*$\dim(e)$ *is called the dimension of $e$), and a partial order $<$ (side-of relation) with $e_1 < e_2 \Rightarrow$* $\dim(e_1) < \dim(e_2)$*. The dimension of $\mathcal{C}$ is the maximal dimension of an element, $d$. By $\mathcal{C}^k$ we denote the set of $k$-dimensional elements of $\mathcal{C}$. Elements with dimension* $0$ *are called* vertices*, elements with dimension $d$ are called* cells *(cf. table 1). A* morphism *between abstract complexes $\mathcal{C}_1, \mathcal{C}_2$ is a mapping $\Phi : \mathcal{C}_1 \mapsto \mathcal{C}_2$ with $e < f \Rightarrow \Phi(e) < \Phi(f)$.*

An abstract complex is a purely combinatorial entity, also known as *poset*. We need the notion of a geometric complex, too:

**Definition 2 (Geometric realization of an abstract complex)** *A geometric realization $\Gamma$ of an abstract complex $\mathcal{C}$ is a Hausdorff space $\|\mathcal{C}\|$ and a mapping*

$$\Gamma : \mathcal{C} \mapsto \Gamma(\mathcal{C}) = \|\mathcal{C}\| = \bigcup_{e \in \mathcal{C}} \Gamma(e)$$

*with*

$$e_1 < e_2 \Leftrightarrow \Gamma(e_1) \subset \partial\Gamma(e_2) \quad \textit{and} \quad \partial\Gamma(e_2) = \bigcup_{e_1 < e_2} \Gamma(e_1) \quad \forall e_1, e_2 \in \mathcal{C}$$

[1]Note that this definition is more general than the one given in [4]

| Element | dim | codim | Sequence Iterator |
|---|---|---|---|
| Vertex | 0 | d | VertexIterator |
| Edge | 1 | d-1 | EdgeIterator |
| Facet | d-1 | 1 | FacetIterator |
| Cell | d | 0 | CellIterator |

Table 1: Combinatorial grid entities

Note that this very general definition allows for cells with holes and several components. This might seem much too general; however, if we think of *quotient grids* induced by a partitioning, it makes perfect sense.

In the special case that every element $\Gamma(e)$ of a geometric complex is homeomorphic to the open unit ball of dimension $\dim(e)$, we obtain the notion of a (finite) *CW-complex* of algebraic topology. While this definition applies to a geometric complex, the condition of having a geometric realization which is a CW-complex obviously constrains the combinatorial structure of an abstract complex.

In the following, if speaking of a grid, we will only assume the general definition of a complex, indicating restrictions where necessary. Important special cases occur for example when the relation $<$ induces a *lattice* structure (see [4]) on the elements, implying unique vertex sets, or when the geometric complex is a *manifold* (with or without boundary).

The following presentation will be terse, a detailed description of the kernel syntax can be found in [4, Appendix A].

### 2.2.2 Combinatorial primitives

The combinatorial layer views a grid as a purely combinatorial entity, that is, it is concerned only with abstract complexes. The *elements* of the grid are its "atoms" and named according to their dimension or codimension, see table 1. A minimal representation of an element of a fixed grid is called *element handle*, which may be simply an integer. Handles are useful e. g. for subranges (see 3.3).

At a very basic level, a grid is a set of sequences: A sequence of its vertices, of its edges, and so on. We can model this property by introducing *grid sequence iterators*, see table 1.

In order to access the incidence relationship, we need *incidence iterators* (table 2). These allow for example to access the sequence of all vertices of a cell (VertexOnCellIterator), see fig. 1. The number of different incidence iterators is $d(d-1)$, where $d$ is the grid dimension.

A similar concept are *adjacency iterators*, which relate elements of the same dimension. We define them only for vertices and cells, because there is no "natural" definition for the intermediate dimensions, and they seem to be hardly used.

As already mentioned, it is not required to implement all types of elements or iterators. Also, even if the kernel interface for an element type is supported, it does not need to be stored explicitly. The best example here is a Cartesian grid, where everything is given implicitly. But also unstructured grids do not have to store all

| | | | |
|---|---|---|---|
| VertexOnVertexIt (A) | VertexOnEdgeIt | VertexOnFacetIt | VertexOnCellIt |
| EdgeOnVertexIt | | EdgeOnFacetIt | EdgeOnCellIt |
| FacetOnVertexIt | FacetOnEdgeIt | | FacetOnCellIt |
| CellOnVertexIt | CellOnEdgeIt | CellOnFacetIt | CellOnCellIt (A) |

Table 2: The full set of incidence and adjacency (A) iterators in 3D

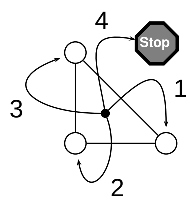


Figure 1: Action of a VertexOnCellIterator (*Incidence iterator*)

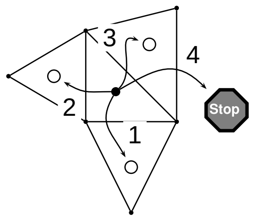


Figure 2: Action of a CellOnCellIterator (*Adjacency iterator*)

their elements: In the triangulation example, there is only storage for cells (the `cells` array), but it is nevertheless possible to define types `Vertex` and `Edge` with corresponding sequence iterators, see section 5.

### 2.2.3 Grid functions

Mathematically, grid functions are simply mappings from grid elements to objects of some type `T`. Behind this simple concept, there are a multitude of different options, which we cannot discuss here in detail, see [3] for more on this topic. The most important distinction is whether grid functions have to allocate storage for each element (*total* grid function), or only for some, assigning a default value to the rest (*partial* grid function). For instance, the grid function used in the generic code snippet on page 4 is a total grid function. Partial grid functions are handy if one has locally operating algorithms which put marks on only a few elements, for example on the boundary.

Total grid functions can be implemented conveniently using arrays, if elements are consecutively numbered; otherwise, and for partial grid functions, hash tables can be used.

Often, one finds grid functions merged into the representation of grid elements, e. g. cells having additional data elements. This approach has the severe drawback of coupling data structures to the algorithms using them, which is even worse than the inverse coupling we want to eliminate. In the generic framework, such implementations are difficult to use, if temporary grid functions are needed. However, adapters can be used to make the in-element data accessible to generic algorithms.

### 2.2.4 Grid Geometries

Grid geometries correspond to geometric realizations of abstract complexes. They may be simply flat (or straight-line) embeddings into $\mathbb{R}^d$, where each element is mapped into an affine subspace of the element's dimension. But we can also think of curved embeddings or more complicated spaces, such as manifolds.

From a functional point of view, a geometry provides mappings from combinatorial entities to geometric entities, most notably vertices to points (coordinates). In addition, a geometry may provide measures such as volumes for each dimension, centers, and so on. We did not limit the possible functionality of a grid geometry to a predefined set of primitives, because different algorithms use quite diverging geometric information. On the other hand, not all geometric queries are meaningful for all geometric embeddings: What is the center of a curved segment? What is the normal of a facet, when the grid is embedded in a higher-dimensional space?

An important aspect of grid geometries is the encapsulation of geometric decisions. If, for example, we look at the generic implementation of the surface algorithm on page 4, we do not need to change anything when we switch from a flat embedding to an embedding into the surface of a sphere, an option useful for instance in a geographic information system. We can also define several geometries for the same (combinatorial) grid, using the "exact" curved geometry only when it is needed (say, for grid refinement), employing the "cheap" flat straight-line geometry otherwise.

Also, we can hide computational aspects, like whether to store or to compute a value. So, even if some geometric questions can be answered by combining lower-level primitives (such as vertex coordinates), it may be wise to leave it to the geometry.

### 2.2.5 Mutating Primitives

Operations that change data structures are in general more complicated than operations that merely read them. Grids are no exception to this rule.

Speaking of mutating grid operations, *Euler operators* [12] which delete or add one (or a few) elements at a time immediately spring to mind. These atomic operations are difficult to implement efficiently for some data structures, for instance the array-based triangulation discussed before. Therefore, we introduced *coarse-grained* primitives which do a better job hiding performance trade-offs, yet are sufficient for large part of mutating algorithms. For a discussion of Euler operators in a generic programming context, see [9].

The number of mutating primitives needed in practice is surprisingly small:

- *Copy*: Copy one grid into another
- *Enlarge*: Glue two grids together (by copying one part)
- *Cut*: Cut off part of a grid

As is shown below, *Copy* could even be considered a special case of *Enlarge*. We will not treat *Cut* here, see section 3.5 for a possible alternative.

Now, we often have data associated to grid, for example in grid functions, which have to be transferred from the source grid to the copy. Therefore, the primitives must support *associative copies*, that is, they return a mapping $M$ (grid morphism) between source $G_{\mathrm{src}}$ and copy $G_{\mathrm{dest}}$.

Thus, the *copy* primitive has the following interface:

$$\texttt{CopyGrid(}G_{\mathrm{dest}}, G_{\mathrm{src}},\ M_{\mathrm{src}\mapsto\mathrm{dest}}\ \texttt{)}$$

`CopyGrid()` is a *semi-generic* primitive: In general, it will be specialized for the type of $G_{\mathrm{dest}}$, whereas the type of $G_{\mathrm{src}}$ remains fully generic. For convenience, there are extended versions which copy also grid geometries, although they are redundant, due to the associative copy feature.

The *Enlarge* primitive needs as additional parameter an identification relation $I$ between source and destination vertices, where $I(v) = w$ means that $w$ is going to be identified with $v$. Similar to `CopyGrid()`, `EnlargeGrid()` produces a mapping $M$ between the source grid and the copied part of the destination grid $G_{\mathrm{dest}}$:

$$\texttt{EnlargeGrid(}G_{\mathrm{dest}}, G_{\mathrm{src}},\ I,\ M_{\mathrm{src}\mapsto\mathrm{dest}}\ \texttt{)}$$

In the special case that $G_{\mathrm{dest}}$ is initially empty (and consequently, so is $I$), we obtain the *Copy* primitive. The latter is provided as a separate primitive, because it is a basic operation and might be implemented more efficiently in some cases. Vice versa, we could use *Copy* and a fused grid view (see 3.5) to emulate *Enlarge*. However, we must then take special care not to destroy prematurely the contents of $G_{\mathrm{dest}}$ (which is referenced by $G_{\mathrm{src}}$).

The *Copy* operation can be used to achieve transparent I/O to different grid formats: For each format, there is one input and one output adapter. We use `CopyGrid` specialized for a given grid type with the input adapter as generic $G_{\mathrm{src}}$ parameter for reading a grid, and `CopyGrid` specialized for the output adapter with the given grid as $G_{\mathrm{src}}$ parameter for writing a grid. Thus, the entire knowledge on the specific format is encapsulated in the input and output adapters.

## 3 The Generic Toolsmith

The potential for generic grid tools is huge, ranging from very basic and general purpose (even parts of the kernel can be implemented generically) to very domain-specific, e. g. FEM discretizations. Having the general 'toolbox' in mind, we will concentrate on general purpose tools, forming a generic "swiss' army knife" for grid processing. For a discussion of components more specific to numerical solution of PDEs see [4].

Due to their genericity, the tools we are going to present are indeed usable with any grid representation, after an adaptation to the kernel interface introduced before (see section 5.1 for an example of how to adapt a given grid data structure). Instead of trying to give an exhaustive overview of all existing generic components, we pick some illustrative examples, discussing some in more depth, while only passing superficially on others. To wit, we treat incidence calculations (section 3.1), grid functions and morphisms (3.2), subranges of grids (3.3), grid boundary (3.4), grid views, which allow to operate on a grid with modified structure without actually changing it (3.5), and a geometric component for matching grids with touching boundaries (3.6).

## 3.1 Cell Neighbor Search

Two cells are *neighbors* if they share a common facet. Accessing cell neighbors is crucial to many algorithms, for example finite volume discretizations. On the other hand, this information is not readily available in the output of most grid generators, nor is it contained in typical file representations. If the grid is such that elements have unique vertex set, cell neighbor information can be deduced from the cell-vertex incidence relation (more precisely, from cell-facet and facet-vertex incidence relations).

**Algorithm 1** Cell Neighbor Search: Find cell neighbors from facet vertex sets

```
IN: A grid $\mathcal{G}$
OUT: A mapping $\mathcal{I} : \mathcal{G}^d \times \mathbb{N} \mapsto \mathcal{G}^d \cup \{c_\infty\}$ which maps each cell to its sequence of
    neighbors, such that the $n$th facet of a cell $c$ is incident to the neighbor in $\mathcal{I}(c,n)$.
    The value $c_\infty$ indicates a boundary facet.
OUT: $\mathcal{N} : \mathbb{P}\mathcal{G}^0 \mapsto \mathcal{G}^d \times \mathbb{N}$ is a mapping from vertex sets to $(c,n)$ = (cell,local side)
    pairs.
1:  for all cells $C \in \mathcal{G}^d$ do
2:    for all facets $F \prec C$ do
3:      $f_C \leftarrow$ local number of $F$ in $C$
4:      if $F^0 \notin \mathrm{dom}\mathcal{N}$ then
5:        $\mathcal{N}(F^0) \leftarrow (C, f_C)$   (Store $C$ and local facet no. at key $F^0$ (vertex set))
6:      else   (facet already found from the other neighbor $D$.)
7:        $(D, f_D) \leftarrow \mathcal{N}(F^0)$
8:        $\mathcal{N}(F^0) \leftarrow \emptyset$
9:        $\mathcal{I}(C, f_C) \leftarrow D$
10:       $\mathcal{I}(D, f_D) \leftarrow C$
11:     end if
12: Set $\mathcal{I}(C,f) = c_\infty \quad \forall (C,f) \in \mathcal{N}$   (Facets still in $\mathcal{N}$ are boundary facets.)
```

Cell Neighbor Search calculates cell-cell adjacencies. It returns a mapping $\mathcal{I} : \mathcal{G}^d \times \mathbb{N} \mapsto \mathcal{G}^d$, where $\mathcal{I}(c_1, f) = c_2$ means that $c_1$ and $c_2$ share facet no. $f$ of $c_1$. If a cell $c$ has a boundary facet $f$, $\mathcal{I}(c_1, f) = c_\infty$, and $(c, f)$ is reported in $\mathcal{N}$.

The algorithm will work correctly only if the grid is (part of) a manifold-with-boundary grid, which is satisfied by all grids commonly used for finite element modelling. In this case, each facet has at most two incident cells.

The algorithm requires only modest grid functionality: The grid type has to provide implementations for CellIterator, FacetOnCellIterator, and VertexOnFacetIterator. Cell Neighbor Search can be used incrementally, visiting only a subset of cells in a grid. Incremental usage typically occurs in `EnlargeGrid` implementations for grids that store cell-cell-adjacencies. Then $\mathcal{N}$ initially contains all identification facets, and $\mathcal{C}$ is the set of new cells of the grid.

The result map $\mathcal{I}$ is a (writable) mapping from FacetOnCellIterators to cells. If the grid has an internal data structure for storing cell neighbors, we can define the type corresponding to $\mathcal{I}$ such that this information is written directly into the grid's internal data structures, thus avoiding any copying overhead.

For ease-of-use, the interface has been layered, from the most general to the simplest. In the latter, all possible defaults are substituted for the most common case of calculating cell neighbors for an entire grid. A generic implementation can be found in GrAL [5].

## 3.2 Grid Functions & Grid Morphisms

Both total and partial grid functions can be implemented generically. GrAL offers an array-based total grid function and a hash-table based (partial or total) grid function, which are suitable in the majority of cases – only rarely a special grid function needs to be tailored for a concrete grid representation. For more details, see [3]. The use of the generic versions for a given grid type, say `MyGrid`, can be triggered by a *partial specialization* of the general grid function templates to the grid's element types:

```
class MyGrid { ... };

template<class T>
grid_function<MyGrid::Vertex,T>
  : public grid_function_vector<MyVertex,T>
{ /*  repeat constructors */ };

template<class T>
partial_grid_function<MyGrid::Vertex,T>
  : public partial_grid_function_hash<MyEdge,T>
{ /*  repeat constructors */ };
```

Grid morphisms can be based conveniently on top of grid functions. We just use a total grid function for each element type:

```
// map G1 to G2
template<class G1, class G2>
class grid_morphism {
  // typedefs omitted
  grid_function<Vertex1,Vertex2> vertex_map;
  grid_function<Edge1,  Edge2>   edge_map;
  ...

  Vertex2 operator()(Vertex1 v) const { return vertex_map(v);}
```

In a practical implementation, we would store only vertex *handles* instead of vertices.

## 3.3 Subranges & Closure Iterators

Grid subranges $\mathcal{R} \subset \mathcal{G}^k$ are ubiquitous. Often, we need the topological closure $\overline{\mathcal{R}} \subset \mathcal{G}$, especially if $\mathcal{R} \subset \mathcal{G}^d$ is a cell set.

Implementing the sequence iterators for the elements of $\overline{\mathcal{R}}$ not in $\mathcal{R}$ can be achieved through *closure iterators*. These iterators use incidence iterators and a marking strategy to visit each element of a given type in a subrange. For instance, a vertex closure iterator uses the CellIterator of the subrange, the VertexOnCellIterator of the underlying grid, and a partial grid function Vertex $\mapsto \{0, 1\}$ to mark already visited vertices. Hence, closure iterators are an example where *partial* grid functions are crucial for achieving optimal run time complexity – expected $O(|\overline{\mathcal{R}}^0|)$ in this case.

## 3.4 Boundary Iterators

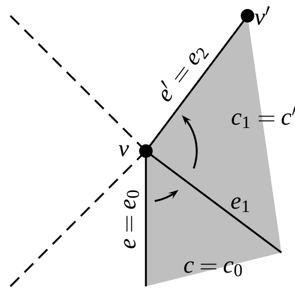
The boundary of a (manifold-with-boundary) grid has a purely combinatorial definition: A facet is on the boundary iff it is incident to exactly one cell (cf. also section 3.1 on cell-cell adjacencies). All elements of co-dimension higher than 1 contained in the closure of all boundary facets are also on the boundary.

The central algorithmic question is now how to iterate efficiently over the boundary elements. A brute force approach to boundary iteration consists in scanning all facets and to look whether they are on the boundary, and to mark incident elements of lower dimension accordingly. This leads to (preprocessing) time $O(|\mathcal{G}|)$ and storage $O(|\partial\mathcal{G}|)$. A disadvantage is that it does not allow us to handle boundary components: Neither can we tell how many there are, nor can we iterate separately over them.

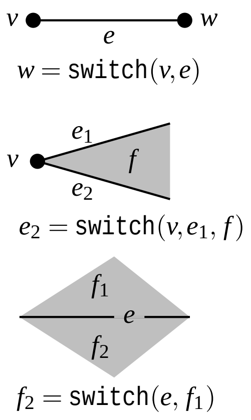
A more sophisticated approach taking boundary components into account is the following:

1. Find a starting *flag* (i. e. a tuple of incident vertex, edge, and cell in 2D, see fig. 3 ) for each boundary component (a *germ* of the component)

2. A component can be iterated over by proceeding from one boundary facet to the next (sharing an element of co-dimension $d-2$), by using local incidence information, to wit, the switch operator together with the NEXT-BOUNDARY-FLAG2D algorithm (see [4] and figure 3).



(a) finding the next boundary flag: $(v,e,c) \mapsto (v',e',c')$


$v$
$w$
$e$
$w = \mathtt{switch}(v,e)$
$e_1$
$v$
$f$
$e_2$
$e_2 = \mathtt{switch}(v,e_1,f)$
$f_1$
$e$
$f_2$
$f_2 = \mathtt{switch}(e,f_1)$

(b) The switch operation in 2D

Figure 3: How to traverse a boundary component of a 2D grid

In general, the preprocessing step 1 also takes time linear in the size of the grid, and (at least temporary) storage of the size of the boundary. Its output, however, uses only storage of one flag (germ) per boundary component, which is typically much smaller. In two dimensions, no additional storage is needed when traversing the boundary (see figure 3). Because of the circular order in boundary components, we just need to check if we are back at the start. In 3D, we need to perform a breadth-first traversal of the boundary, using a partial grid function to mark visited elements.

Using the switch operator requires more grid functionality than the first approach. Internally, switch uses cell-cell adjacencies which can be calculated by CELL-NEIGHBOR-SEARCH of section 3.1. As we will make heavy use of boundary components in section 4, we describe this approach in some more detail.

The building block is a BoundaryComponent, which in turn offers a BoundaryFlagIterator. BoundaryComponent defines also iterators for all element types (except cells), which are simple wrappers around BoundaryFlagIterator. A BoundaryComponent is initialized with a boundary flag. A BoundaryFlagIterator uses the approach in step 2 for its increment operation.

```
template<class G>
class BoundaryComponent {
  flag germ;

  class FlagIterator {
    flag current;

    FlagIterator & operator++() { /* use next-boundary-flag2d */}
    Vertex current_vertex();
    Edge   current_edge  ();
    // ...
  };

  class VertexIterator {
    FlagIterator it;
    // ...
    Vertex operator*() const { return it.current_vertex();}
  };

  class EdgeIterator { /* ... */ };
};
```

It is the job of a BoundaryRange to provide access to the whole boundary. It manages a set of germ flags (one for each component), and provides the possibility to iterate both over components (BoundaryComponentIterator) or elements (by nesting a BoundaryComponentIterator with the element iterators of the latter).

```
template<class G>
class BoundaryRange {
 grid_type  *g;
 list<flag> germs;

 BoundaryRange(G const& g_) : g(&g_) {
   // eager version
   find_boundary_component_germs(g_);
 }

 BoundaryComponentIterator FirstComponent() const
```

```
  { return BoundaryComponentIterator(germs.front()); }
};
```

In the above implementation, we initially perform a global loop over all facets in order to find a germ for each component, using `find_boundary_component_germs()`. If a boundary facet is found which has not yet been visited, the algorithm uses `BoundaryComponent<G>` to loop over the corresponding component, and marks all facets of that component.

A possible optimization would be to use a 'lazy' version which could be beneficial if only few boundary components are eventually visited. This, however, is more involved, especially if the user controls iteration by using e. g. `ComponentIterator` and `ComponentIterator::VertexIterator`, because now `BoundaryRange` must know whether a component has been traversed completely or not.

## 3.5 Grid Views

Grid *views* are among the most powerful components of a generic grid library. They allow to hide disparate grid operations under the common grid kernel interface; therefore, no new syntax has to be learned in order to use them.

In the sequel we will briefly present three examples of grid views which are used in the hybrid grid generation case study (section 4): A disjoint union grid view, a fused grid view, and a cell difference grid view.

Common to all these views is that they hold a *reference* to their base grid(s) (the *viewee*), which they never modify; thus, no copying is involved. Also, views can be nested – a view can be defined on top of another view, see section 4 for examples.

In principle, views should try to preserve as much properties of the underlying grid as possible. For example, if the viewee has a `CellOnCellIterator`, the view should also provide one, at least if possible without effort. This, however, requires some introspection mechanism for detecting the actual extent to with the kernel is supported by the viewee, and is not yet fully developed.

Grid views can be accompanied by corresponding geometry and grid function views, thus completing the support of the grid kernel, and making them for most practical purposes indistinguishable from "real" grids.

**Disjoint union grid view** Perhaps the conceptually simplest operation on grids is their *formal* or *combinatorial union*, that is, all grids are considered disjoint. They can be "glued together" by using `fused_grid_view` later.

Such a disjoint union grid view just holds a container of references to grids. The sequence iterators can be implemented by nesting iteration over the grids and than over the corresponding elements; incidence iterators can be simply adopted from the underlying base grids.

**Fused Grid View** The next slightly more complicated case occurs if some of the vertices of a grid need to be identified, for example to glue two grids together (seen as a single grid by virtue of disjoint union grid view). Here we need, in addition to a grid, an equivalence relation $I$ on its vertices, similar to *Enlarge* (section 2.2.5). The equivalence classes of $I$ are the vertices of the view. Also, other elements have to be fused if necessary; we assume that cells are never fused ("identification occurs on the boundary only").

Besides a reference to the viewed grid, a partial mapping from vertices to their representants will be stored, which needs storage proportional to the number of identified vertices. As mentioned above, a fused grid view on top of a disjoint union grid view can be used to create a *geometric union* of grids, that is, elements which match geometrically are identified (see also section 3.6 for finding those elements).

**Cell Difference Grid View** Often, we need to perform operations on a grid minus a few cells. Instead of physically removing those cells, we can use a cell difference grid view. When implementing such a view, we must take care to "remove" also all elements of lower dimension incident only to removed cells. An implementation typically contains a partial grid function for each element type, marking removed elements.

In principle, the same effect could be achieved with grid subranges. Cell difference grid views are more efficient when only few cells are to be removed. If, however, there are no cell-cell adjacencies, the preprocessing time is $O(|\mathcal{G}|)$, regardless of the number of removed cells.

## 3.6 Matching vertices

For performing the *geometric union* of grids mentioned before, we need to find the sets of vertices to be identified. If no additional information is available, we have to match vertices geometrically, that is, have to look which (boundary) vertices are "at the same location": Given two grids $\mathcal{G}_1$ and $\mathcal{G}_2$ with geometries $\Gamma_1$ and $\Gamma_2$, find the pairs of vertices $v_1 \in \mathcal{G}_1, v_2 \in \mathcal{G}_2$ with $\Gamma_1(v_1) \equiv \Gamma_2(v_2)$.

What should the exact meaning of $\equiv$ be? Direct comparison of vertex coordinates for equality is only appropriate in certain circumstances: Either exact representation and arithmetic is used, or we must ensure identical representations of coordinates in the $\Gamma_i$, with coordinates of matching vertices being copies of each other. Else, representations of the "same" coordinates will differ, possibly even if they are "just" copied, but converted into different intermediate representations (`double` to `float` or back).

So, in the general case, we must test whether $\text{dist}(v_1, v_2) \leq \varepsilon$, with a suitable $\varepsilon$. On the one hand, $\varepsilon$ must be small enough to exclude false matching, on the other hand, it should be large enough to allow for the losses of e. g. `double` to `float` conversions. As there is no way of finding an optimal $\varepsilon$ in all situations, the user should be able to provide $\varepsilon$. However, a good default is to set $\varepsilon$ to a small fraction of the minimum boundary edge length of any of the grids, which is hopefully the minimum distance of boundary vertices.

Now, the simplest (brute-force) approach to finding matching vertices is to simply nest two loops over the boundary vertices of the two grids, checking for every vertex in $\partial\mathcal{G}_1$ if there is a matching vertex in $\partial\mathcal{G}_2$. This has complexity $O(|\partial\mathcal{G}_1||\partial\mathcal{G}_2|)$. If we assume both boundaries of the same size and the grids "well-shaped", then $|\partial\mathcal{G}_i| = O(|\mathcal{G}|^{1-1/d})$, that is, the algorithm has complexity $O(|\mathcal{G}_1|)$ in 2D and $O(|\mathcal{G}_1|^{4/3})$ in 3D.

To achieve better performance, we could use a search structure to speed up finding matching vertices to $O(|\partial\mathcal{G}_1| \log |\partial\mathcal{G}_2|)$, at the cost of some preprocessing. Also, we could use bounding boxes of boundary components to filter vertices – vertices in two components of two grids can match only if they are contained in the intersection of both bounding boxes. This is of no help if the whole boundary component matches. We may therefore complement the bounding boxes approach by first searching for matching vertices locally in a neighborhood of an already found pair of vertices.

Again, no single strategy will be optimal for each situation, so the final decision has to be left to the user. Currently, GrAL only implements the simplest strategy.

# 4 Using the Toolbox for Hybrid Grid Generation

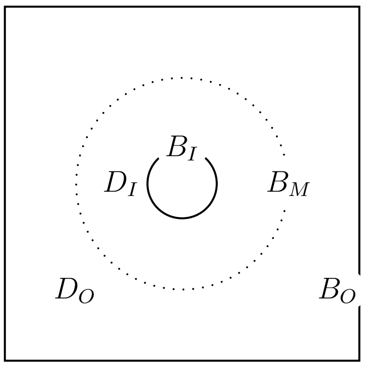
In order to benchmark the ideas presented so far for a realistic task, let us now look at an example involving hybrid grid generation. Our aim is to accomplish as many tasks as possible with *standard* tools — i. e. tools *not* designed specifically for hybrid grid generation. The ability to use *general* tools for *specialized* tasks will clearly demonstrate the usefulness and feasibility of the generic toolbox approach, but we will also encounter some typical trade-offs.

What is hybrid grid generation, and what is it good for? The key problem is that the decision of what constitutes a good grid can vary considerably depending on the physical phenomena to be approximated. For example, for viscous flows the fluid behaves very differently near the solid wall boundary. In the so-called boundary layer, velocity increases from zero at the wall very rapidly in direction normal to the wall to the free-stream velocity in the interior of the domain. In order to numerically approximate boundary layers, cells must be made very thin in direction normal to the boundary, yet should be of regular size in other directions. This requires a special grid generation process which is aware on the directional dependencies, leading to body-fitted grids, see [21]. The rest of the domain can be filled with an unstructured grid, which is better suited for complicated geometries.

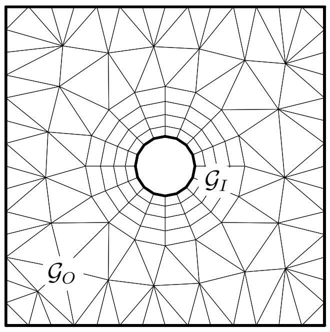
## 4.1 The Basic Task of Hybrid Grid Generation

We begin with the simplest incarnation of the hybrid grid generation problem: Given an inner boundary $B_I$ of circular topology (in 2D), and an arbitrary outer boundary $B_O$, we seek to generate a body-fitted grid $\mathcal{G}_I$ around the inner boundary, and to fill the rest with an unstructured grid $\mathcal{G}_O$, finally yielding a single hybrid grid $\mathcal{G}_H = \mathcal{G}_I \cup \mathcal{G}_O$, see figure 4. After being able to handle this situation, we will gradually extend the task to several inner boundaries, finally allowing the generated body-fitted grids to overlap, a situation which has to be detected and corrected by our framework.

Our description is targeted towards two dimensions; we will indicate where differences and commonalities with respect to the 3D case occur. The implementation does not yet provide us with an industrial strength hybrid grid generator, but it clearly indicates the feasibility of such a task with generic tools.



(a) Schematic configuration with inner domain $D_I$,outer domain $D_O$ and $D = D_I \cup D_O$



(b) The hybrid grid $\mathcal{G}_H$ is the union of the two partial grids $\mathcal{G}_I$ and $\mathcal{G}_O$

Figure 4: The basic task of hybrid grid generation in 2D

For solving the basic task with only one inner boundary, we will roughly proceed as follows (cf. figure 4):

1. Generate a structured grid $\mathcal{A}_I = \mathcal{G}_I$ with annular topology around the inner boundary (an *O-grid*), covering the domain $D_I$ (its outer boundary $B_M$ gets defined by the process)
2. extract the outer boundary $B_M$ of $\mathcal{G}_I$, and create a suitable input (geometric representation of $D_O$) for an unstructured grid generator
3. generate the unstructured grid $\mathcal{G}_O$ in the annular domain $D_O$.
4. Glue both grids together to get a hybrid grid $\mathcal{G}_H = \mathcal{G}_O \cup \mathcal{G}_I$
5. Control the quality of $\mathcal{G}_H$, especially at the junction (jumps in cell sizes, edge lengths, directions etc.). We will not discuss this issue further in this paper.

Using standard tools for these tasks means in particular, that the components for generation of the structured and the unstructured part should not know about each other. In fact, we would like to create a framework in which we can plug any grid generator able to produce the desired type of grid.

For the first task, a generator for circular structured grids is needed. Here we assume that a routine written in a more "classic" style is available, which takes as input a one-dimensional field `B_I` containing the boundary (plus some optional grid spacing parameters `spacing`) and has as output a two-dimensional field of grid points in `RGeom`, which are assumed to have identical values at both ends in one coordinate direction. For the examples, we used the orthogonal grid generation procedure described in [8].

```
reg_grid_t R;
reg_geom_t RGeom(R); // grid geometry for reg_grid_t
generate_orthogonal_O_grid(B_I, R, RGeom, spacing);
```

As a standard type for Cartesian grids, `reg_grid_t` does not provide an annular topology. Instead of creating a specialized Cartesian view with one pair of sides identified, we can reuse a more general component, to wit, fused grid view of 3.5, which allows to identify arbitrary subsets of vertices. Thus, we loose some information on the Cartesian structure of the underlying grid `R`, but the loss does not hurt here. The technique also extends more smoothly to other configurations like so-called *C-grids*, where only parts of a block side are identified. We can find the identification vertices either by index calculations on the original grid `R`, or, as shown below, by some geometric methods discussed in section 3.6:

```
typedef partial_vertex_morphism<reg_grid_t> mapping_t;
mapping_t R2R;
match_boundary_vertices(R,RGeom, R, RGeom, R2R);

typedef fused_grid<reg_grid_t, mapping_t>    annular_grid_t;
typedef fused_geom<annular_grid, reg_geom_t> annular_geom_t;
annular_grid_t  G_I   (R,R2R);
annular_geom_t  Geom_I(G_I,RGeom);
```

Next, we need to extract the outer boundary $B_M$ of the inner grid $\mathcal{G}_I$. Here, too, we could exploit the Cartesian structure, but we adopt a more general approach which covers also the case of an unstructured $\mathcal{G}_I$. For instance, in 3D hybrid grid generation, one often starts with an unstructured triangulation of the inner boundary $B_I$, which is then extruded using prisms, leading to a sort of 'semi-structured' grid $\mathcal{G}_I$, see [21].

We copy the inner grid $\mathcal{G}_I$ to the resulting hybrid grid $\mathcal{G}_H$, using the semi-generic copy operation for the type `hybrid_grid_t` of $\mathcal{G}_H$ (see section 2.2.5):

```
hybrid_grid_t G_H;
hybrid_geom_t Geom_H(G_H);
CopyGrid(G_H, Geom_H,  G_I, Geom_I);
```

In order to find the outer boundary components `B_M` of `G_H` = `G_I`, we use generic tools for extracting connected components of the boundary, described in section 3.4. Here, we just need to select the outer component, which can be done for instance by checking if some inner grid point is contained in the polygon (polytope in 3D) defined by the component.

```
BoundaryComponent<hybrid_grid_t> B_M = get_outer_bd_component(G_H, Geom_H);
```

The extracted boundary component `B_M` represents the inner boundary of the outer annulus $D_O$. Using a representation for domains with one (or several) holes, we can create appropriate input data for a given unstructured grid generator. In the practical implementation, we used the TRIANGLE generator [16], to produce the unstructured grid `G_O` covering $D_O$:

```
annular_domain_t D_O(B_M, B_O);

us_grid_t G_O;
us_geom_t Geom_O;
generate_us_grid(D_O, G_O, Geom_O);
```

Then, we need to enlarge G_H by G_O. In order to do so, we must detect which vertices of both grids will have to be identified. There are several possibilities to determine these vertices. First, we could maintain a relationship (grid morphism) of the vertices in the outer boundary B_M of G_H (which is equal to G_I at this stage) to the corresponding vertices of G_O. This is certainly the cleanest solution; however, not every grid generator does support such a tracking. Second, we may take the vertices in the outer boundary of G_I and find geometrically matching counterparts in G_O. And finally, we can forget everything we know about G_I and G_O, and identify the vertices entirely by geometric means, using match_boundary_vertices(), as before.

Whatever the procedure for finding the identification vertices, given them, we use the semi-generic EnlargeGrid operation (see section 2.2.5) on $\mathcal{G}_H$ to produce the final grid:

```
partial_vertex_morphism<us_grid_t, hybrid_grid_t> Id;
/* ...  calculate Id by some means ... */
EnlargeGrid(G_H, Geom_H, G_O, Geom_O, Id);
```

## 4.2 Broadening the Task — Several Holes

If there are several holes, the inner grid $\mathcal{G}_I$ will be the union of grids $\mathcal{G}_i$, which in the simplest case are just regular annular grids $\mathcal{A}_i$, created just as before:

```
// create Cartesian grids
vector<reg_grid_t> R    (nholes);
vector<reg_geom_t> RGeom(nholes);
for(unsigned i = 0; i < R.size(); ++i)
  generate_orthogonal_O_grid(B_I[i], R[i], RGeom[i], spacing);


// create annular views
vector<annular_grid_t> A_I(nholes);
for(unsigned i = 0; i < nholes; ++i) {
  mapping_t R2R;
  match_boundary_vertices(R[i], RGeom[i], R[i], RGeom[i], R2R);
  A_I[i] = annular_grid_t(R[i], R2R);
}
```

It is algorithmically easier to extract outer boundary components of the single annular grids before performing their union. This is feasible if annular_grid_view supports boundary iteration (cf. section 3.4). If it does not — that is, the implementation suffers from the problem mentioned at the beginning of section 3.5 – we have to calculate outer boundary components in the single grid $\mathcal{G}_I$. (This part is not shown.)

At any rate, we have to form the union of the annular grids $\mathcal{A}_i$:

```
// create union grid
typedef disjoint_union_grid<annular_grid_t> inner_grid_t;
inner_grid_t G_I(A_I.begin(), A_I.end());
```

We can now proceed as before, using the extracted outer boundary grids (as well as the outer domain boundary $B_O$) as input for a domain representation which allows for multiple holes (not shown).

Now, for a robust implementation, we need to check for the possibility that the inner grids might overlap, or even extend into the outside of the domain. A first step is to determine which cells actually do overlap. This problem is a very general one, occurring also in the context of *overset grids*, see [21]. The most primitive strategy for finding overlapping cells is to simply check each cell of an annular grid $\mathcal{A}_i$ for inclusion in each other annulus $\mathcal{A}_j, j \neq i$. Possible optimizations are bounding boxes to exclude non-overlapping pairs of grids a priori, and search structures to speed up point location, such as provided by the generic library CGAL [15].

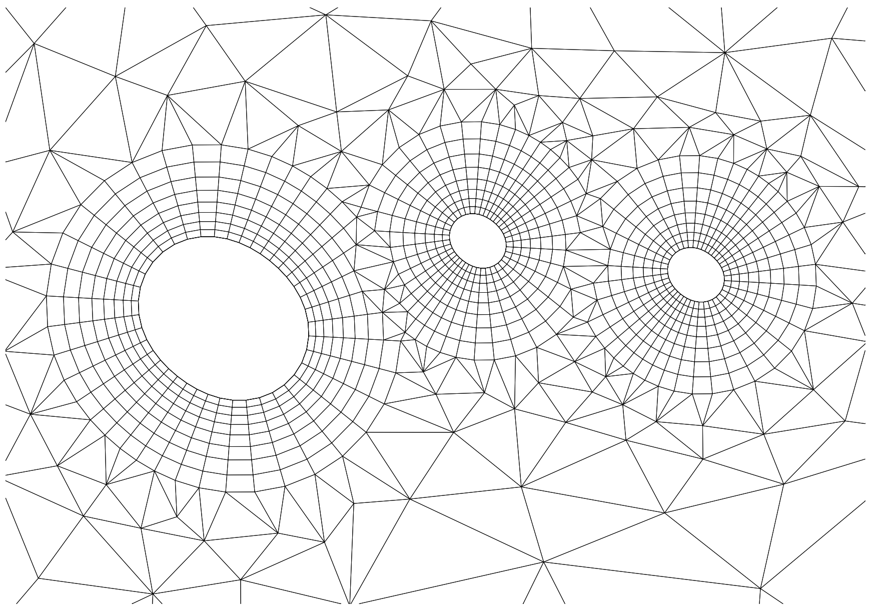


Figure 5: Hybrid grid with 3 holes

Different strategies can be used to decide which cells to remove:

- Establish a minimum distance between clipped grids
- Remove all cells which overlap (could “eat up” all cells of a small inner hole contained in another grid)
- Remove the least possible amount of cells (beware of small distances!)
- remove “rings” of cells until no overlap occurs, thus keeping the Cartesian structure intact
- keep those cells that are nearer to their inner boundary than their conflicting counterparts

Unfortunately, the decision will usually interfere with an optimal implementation of the cell overlap query, because often not all cells have to be tested. What we try to explain here is how to arrive at a quick, but not necessarily optimal solution;

it should be clear that this first shot can be improved without compromising the generic paradigm. Regardless of the strategy chosen, we arrive at a set $C_i$ of cells to be removed from the annular grids $\mathcal{A}_i$.

In the present example, we simply collected all overlapping cells in `clip_cells`, that is,

$$C_i = \mathcal{A}_i \cap \bigcup_{j \neq i} \mathcal{A}_j$$

The clipped grids $\mathcal{G}_i = \mathcal{A}_i \setminus C_i$ are readily represented by a cell difference view:

```
vector<annular_grid_t> A_I(nholes);

/* ... create annular views as before ... */

// determine cells to be cut away
vector<cell_range<annular_grid_t> >   C_I(nholes);
clip_cells(A_I.begin(), A_I.end(),
           C_I.begin());

// create difference views
typedef cell_difference_grid<annular_grid_t> clipped_grid_t;
vector<clipped_grid_t> D_I(nholes);
for(unsigned i = 0; i < nholes; ++i)
  D_I[i] = clipped_grid_t(A_I[i], C_I[i]);

typedef disjoint_union_grid<clipped_grid_t> inner_grid_t;
inner_grid_t G_I(D_I.begin(), D_I.end());
```

Now we can proceed as above by determining outer boundary components. As before, we first copy the union `G_I` of the clipped view to the hybrid grid `G_H`, in order to exploit its better support for incidence iterators, necessary to handle boundary components (cf. also the discussion on view and capabilities of underlying grids in page 14).

Of course, there is a lot of room left for improvements. For instance, we could use more sophisticated strategies inside `clip_cells()`. Also, the whole aspect of grid quality has not been covered at all, and is clearly beyond the scope of this section. However, the author hopes that the example given is a convincing point of departure for applying the generic principle further to this class of problems.

## 5 Nuts & Bolts

Now that we have seen how powerful the generic approach is, two questions might arise: First, how can we use it with existing grid data structures (which we do not intend to throw away)? And second, can generic programming keep its promise of efficiency in the context of grids? We will see that both questions have quite favorable answers in general; however, there are some caveats we will discuss.

### 5.1 Adapting a Given Grid Data Structure

Say we have a lot of code centered around the simple triangulation data structure mentioned before — an array `cells` with cell-vertex incidences, and an array `coords` with vertex coordinates. If we want to use generic algorithms with this data structure, we have to implement the kernel interface at least as far as necessary for the algorithms, or as far as possible for this data structure.

```
class triang2d  {
  int * cells;
  ...
};
```

Starting with the combinatorial functionality, we see that from the data present in `cells` we can support the following:

| | | |
|---|---|---|
| Vertex | VertexIterator | VertexOnCellIterator |
| Edge (= Facet) | EdgeIterator | EdgeOnCellIterator |
| Cell | CellIterator | |
| | | VertexOnFacetIterator |

The implementation of `Cell`/`CellIterator`, as well as `Vertex`/`VertexIterator` is straightforward; also, the incidence iterator `VertexOnCellIterator` is not difficult to do. See [4, Appendix B.1] for details (note that `Vertex` is the same type as `VertexIterator` there).

But what about `Edge` and `EdgeIterator`? The first observation is that an edge can be represented by a pair of vertices. If we require the pair to be ordered, this representation is even unique, assuming no two edges have the same set of vertices.

```
class triang2d::Edge {
  vertex_handle        v1, v2;   // v1 > v2
  triang2d      const* g;
  // ...
};
```

Now, implementing `EdgeOnCellIterator` is also straightforward. But can we iterate over all edges of a grid? It turns out we can, but at some cost. The idea is to nest global iteration over all cells and local iteration over the edges of the current cell. Thus, we will visit each inner edge exactly twice, and we need a means to skip one of these. We can do so by keeping track of already visited edges by using a (partial) grid function `Edge` $\mapsto \{0, 1\}$. Then, `EdgeIterator` looks like this:

```
class triang2d::EdgeIterator {
  CellIterator        c;
  EdgeOnCellIterator ec;
  partial_grid_function<Edge, bool> visited;
public:
   EdgeIterator& operator++() {
     if(! ec.IsDone()) ++ec;
     else {
      ++c;
      if(! c.IsDone()) ec = (*c).FirstEdge();
     }
   }
  // ...
};
```

As it turns out, this situation is so common that the whole approach has been bundled into a generic component in GrAL, so we just need to say

```
class triang2d {
  // ...
  typedef facet_iterator<triang2d> EdgeIterator;
};
```

Next, we have to provide grid functions for `triang2d`. This is simple in the case of vertices and cells, because they are consecutively numbered. So we can use a generic array-based implementation for total grid functions and a hash-based implementation for partial grid functions, as already discussed before (page 11). For some technical issues, see [3].

In the case of edges, we have to use hash tables for both total and partial grid functions. Again, we can use generic implementations. As hash function, we have found $pv_1 + v_2 \mod p$ useful, with some small $p$.

Finally, if we want to use geometric algorithms, we need to define a geometry for `triang2d`. At first sight, this looks rather simple, but there is a subtle problem, relating to the fact that vertex coordinates are stored in a plain array of doubles:

```
class geom_triang2d_base {
   double* coords;
   typedef ... coord_type;  // 2D point type

   ??? const& coord(Vertex v) const;
   ???      & coord(Vertex v);
};
```

What should we return from the non-const `coord(Vertex v)`? On the one hand, we want allow code like `coord_type p = geom.coord(v)`, on the other hand, an assignment like `geom.coord(v) = coord_type(1,0)` should change the array `coords`. Thus, we cannot return `coord_type &`.

The solution is to define a proxy type:

```
class geom_triang2d_base {

   class coord_proxy {
     double * coo;
     void operator=(coord_type const& p)
     {  coo[0] = p[0]; coo[1] = p[1]; }
   };

   coord_proxy      & coord(Vertex v)
   { return coord_proxy(coords[2*v.handle()]);}
};
```

For the 'real' `coord_type`, we are free to choose any suitable implementation of a 2D geometric point. In order to get a bunch of useful geometric functionality (like volumes or centers), we use a generic 2D geometry:

```
 typedef geometry2d<geom_triang2d_base>  geom_triang2d;
```

The efficiency of this generic implementation will depend on how well the compiler can optimize out the overhead (see also the discussion in section 5.3); if it turns out to be too inefficient, parts have to be specialized for `triang2d` using direct access to its data. This does not break the generic paradigm, as grid geometries are part of the kernel.

## 5.2 Some Issues with Generic Programming

With the adaption `triang2d` we have just created, *all* algorithms and generic data structures (respecting the kernel interface) can be used for this data structure, *provided* the requirements of the generic components are met.

For example, the boundary component iterator described in section 3.4 uses cell-cell adjacency information and cannot be used. This is not a shortcoming of the generic paradigm, but reflects an intrinsic limitation of our data structure which just cannot provide this information in an efficient way.

The kernel concepts thus provide a way to classify algorithms and data structures according to the functionality they require or offer, respectively. A generic component can be used with a concrete grid data structure iff the requirements of the former are a subset of the capabilities of the latter. An interesting option would be to automatically create a list of all generic components (out of some set of such components) usable with a given data structure.

The adaptation effort can be eased considerably by using generic implementations of kernel concepts, like `facet_iterator`, total and partial grid functions, or grid geometries. This path should be pursued further; for instance, there could be a generic version for index-based vertex iterators like `triang2d::VertexIterator`.

In spite of the favorable picture we have painted so far, there are some difficulties associated to the generic approach, which make it somewhat harder to use than the more common methods. Most of the problems stem from technical sources and are partly due to the relative novelty of the approach.

First, compile times and memory requirements tend to blow up, at least with state-of-the-art compilers. Also, precompiling code, as done in classical libraries, is not possible for generic libraries. Support (e. g. incremental compilers) for avoiding recompiling templates over and over for a fixed application is still in its infancy.

Second, tool support for generic programming is scarce: Error messages of compilers relating to templates are long, not always helpful and often do not point to the source of the error. For example, if a requirement of an algorithm is not met, the error will be reported at some point deep in the algorithm's implementation, while the real error occurs where some user-level algorithm interface is called with an inappropriate type. Techniques like concept checks [19, 13] can help to catch these errors at the place where they occur. Of course, such errors are also a consequence of the additional degree of freedom introduced by type parameters.

Documentation tools also have their problems with templates. Specialization relationships, which are central to generic programming, are hardly supported. This is not very surprising, as also compiler writers had (and still have) a hard job implementing these new C++ features.

So, although the situation keeps improving, using generic libraries is not for beginners. The practitioner, on the other hand, will quite surely greatly benefit after an initial learning effort. As W. S. Humphrey has put it in [7], reuse is the only currently available technology having the potential to increase productivity by an order of magnitude. And generic programming is a technique enabling reuse of algorithms in areas where this has not been practical before.

## 5.3 Efficiency of Generic Components

Efficiency is an important aspect of a software component's (re-)usability. In high-performance applications, a too inefficient component is practically useless, or at least usable only for small problems with instructional or reference purpose.

It is obvious that a generic component can be at most as efficient as an implementation specialized for a specific data structure. There seem to be two main reasons for inefficiencies introduced by a generic version:

1. *Syntactic overhead* caused by the syntactic side of abstraction, such as additional layers of indirection, nested function calls, and small intermediate objects
2. *Semantic overhead* caused by the abstraction process loosing information which could be used to produce a more efficient implementation

Inefficiency stemming from the first source can be measured by comparing a specialized version implemented in an low-level language (like F77) to a generic version, choosing sufficiently simple test cases and data structures where the second source can be excluded.

It turns out that the results are highly compiler dependent, and that some compilers already do a very good job, up to completely eliminating the overhead, for example for the generic version of the surface algorithm shown before. Clearly, the depth of abstractions stacked on top of each other has an adverse effect on efficiency. More details can be found in [4].

Besides the generic vs. low-level dichotomy, we found that data layout issues are normally more critical to performance. Also, in grid processing, which mostly is preprocessing, memory often tends to be a more constraining bottleneck than pure speed. Here generic components like views which need not copy their data have an advantage over classical solutions.

Inefficiencies of the second kind range from gross over-generalizations (as applying a general search-structure for locating a point in a Cartesian grid with axis-parallel geometry) to rather minor differences, as whether indexing of vertices starts from 1 or 0.

In case of insufficient efficiency it is always possible to *specialize* the generic component towards the concrete situation. Note that specializations do *not* break the generic paradigm. In fact, they are an integral part of generic programming, in sharp contrast to classical object-oriented programming (see e. g. the problems mentioned in [6]).

In the case of syntactic overhead, specialization is merely a workaround, the need for which might vanish with the next compiler release. In contrast, for inefficiency stemming from too coarse abstractions, specializations only reflect the need to model the rich inner structure of the domain more faithfully. There is a permanent trade-off between the desire for genericity (minimizing programming effort) and the need for optimized algorithms, exploiting special structure of the data.

As specialization fits so nicely into the framework of generic programming, we can use it in an incremental fashion as the need arises, without breaking existing design. A very powerful construct which helps to stay as generic as possible is *partial specialization*, which has been added only recently to C++. This technique leads to a specialization tree, the most general version at the root. For example, the point location component alluded to before could be specialized for *all* grids with a Cartesian structure. This of course would require to distill the commonalities of Cartesian grids into a set of abstractions, as has been done for general grids.

Finding significant abstractions demands hard conceptual work and continuing effort. It is the essence of generic programming.

## 6 Discussion

The generic approach presented in this paper allows for the first time to create universally reusable tools for grid processing and thus has the potential to increase productivity by an order of magnitude in this field.

Its great practical advantage is that it can be used incrementally and non-exclusively, allowing to continue the use of existing bodies of code, while exploiting the power of generic programming where advantageous. The associated reuse effort is constant (that is, independent of the number of generic components used); the necessary interface programming involves only rather simple code, and could be further reduced by generic versions of additional kernel components.

However, it must be taken into account that generic programming enters a new level of abstraction. It thus has a steep learning curve and is not suited very well for beginners. Tool support is still limited, and technical difficulties must be expected, although the situation is improving.

If used properly, however, the gains should outweigh the learning investment by far. The GrAL (Grid Algorithms Library, [5]) is a reference implementation available online, helping to put these ideas into practice. Making the components offered increasingly user friendly is an ongoing effort, coupled to the growing experience with generic libraries.

One of the big challenges for future work will be the establishment of a critical mass of "standard" generic tools for meshing applications which cover the common needs. This task necessitates the joint effort of many researchers. It includes further work on the abstract concepts, for example to enhance support for modifying operations, and to allow generic implementations of algorithms which need some sort of local coordinate systems on cells, as FEM discretizations do.

There is also a need for interfacing with related efforts from other domains: The BGL (Boost graph library, [20, 18]) defines an analogous kernel for general graphs. For instance, partitioning algorithms could be based upon that. The MTL (Matrix Template Library, [17, 11]) is a generic library for sparse and dense linear algebra, and hence could be coupled via FEM discretizations. CGAL [15] is a library for geometric computing using generic concepts. It offers a rich set of geometric data structures and algorithms, and covers a similar problem domain as GrAL.

Besides the more directly measurable effects in terms of reusable code, generic programming also changes our understanding of the problem domain. It fosters a thorough analysis of the mathematical structure of grids, their representation in data structures, and the requirements of algorithms operating on them. This *domain analysis* produces a taxonomy of both data structures and generic components with respect to their functionality and requirements. It leads to the establishment of a precise, common domain vocabulary which helps to communicate ideas more clearly, and deepens our overall understanding of the whole domain. Which, in turn, will help us creating the next generation of grid tools.

## References

[1] Ed Anderson et al. *LAPACK users' guide.* SIAM, 2nd edition, 1995.

[2] Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors. *Modern Software Tools in Scientific Computing.* Birkhäuser Press, 1997.

[3] Guntram Berti. Generic components for grid data structures and algorithms with C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[4] Guntram Berti. *Generic software components for Scientific Computing.* PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany, 2000.

[5] Guntram Berti. GrAL – the Grid Algorithms Library. `http://www.math.tu-cottbus.de/~berti/gral`, 2001.

[6] Are Magnus Bruaset, Erik Jarl Holm, and Hans Petter Langtangen. Increasing the efficiency and reliability of software development for systems of pdes. In Arge et al. [2].

[7] Watt S. Humphrey. *A Discipline of Software Engineering.* SEI series in software engineering. Addison Wesley, 1995.

[8] Patrick Knupp and Stanly Steinberg. *Fundamentals of Grid Generation.* CRC Press, Boca Raton, FL, 1994.

[9] Ullrich Köthe. *Generische Programmierung für die Bildverarbeitung.* PhD thesis, Universität Hamburg, 2000.

[10] Meng Lee and Alexander A. Stepanov. The standard template library. Technical report, Hewlett-Packard Laboratories, February 1995.

[11] Andrew Lumsdaine and Jeremy Siek. The Matrix Template Library (MTL). `http://www.lsc.nd.edu/research/mtl/`, 1999.

[12] Martti J. Mäntylä. Computational topology: a study of topological manipulations and interrogations in computer graphics and geometric modeling. *Acta Polytech. Scand. Math. Comput. Sci. Ser.*, 37:1–46, 1983.

[13] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[14] David R. Musser and Alexander A. Stepanov. Generic programming. In Patrizia Gianni, editor, *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings*, number 358 in LNCS, pages 13–25. Springer, 1989.

[15] The CGAL project. The CGAL home page – Computational Geometry Algorithms Library. `http://www.cs.uu.nl/CGAL/`, 1999.

[16] Jonathan R. Shewchuk. Triangle: A two-dimensional quality mesh generator. `http://www.cs.cmu.edu/~quake/triangle.html`, 1999.

[17] Jeremy Siek. A modern framework for portable high performance numerical linear algebra. Master's thesis, University of Notre Dame, 1999.

[18] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. BGL – the Boost Graph Library. `http://www.boost.org/libs/graph/doc/table_of_contents.html`, 2000.

[19] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[20] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. The generic graph component library. *Dr. Dobbs Journal*, 25(9):29–38, September 2000.

[21] Joe F. Thompson, editor. *Handbook of grid generation.* CRC Press, 1999.