ViennaGrid 1.1.0

User Manual

Institute for Microelectronics
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria

Institute for Analysis and Scientific Computing
Wiedner Hauptstraße 8-10 / E101
A-1040 Vienna, Austria/Europe

Main authors:

Karl Rupp (Project Head)
Florian Rudolf
Josef Weinbub

Contributors:

Peter Lagger
Markus Bina

Institute for Microelectronics
Vienna University of Technology
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria/Europe

Phone	+43-1-58801-36001
FAX	+43-1-58801-36099
Web	`http://www.iue.tuwien.ac.at`

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstraße 8-10 / E101
A-1040 Vienna, Austria/Europe

Phone	+43-1-58801-10101
Web	`http://www.asc.tuwien.ac.at`

Contents

Introduction

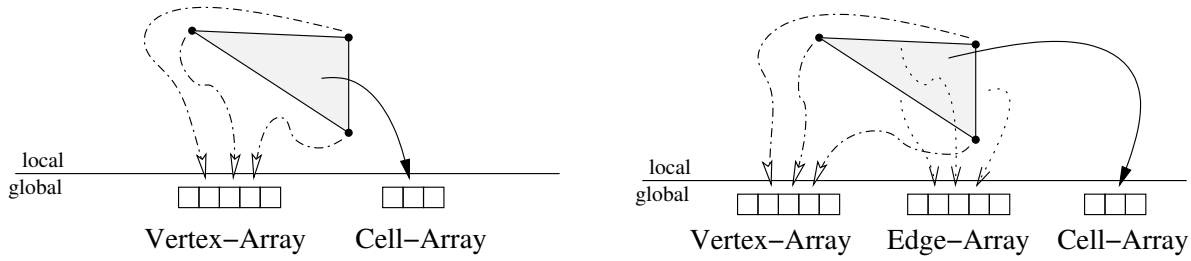
The tessellation of surfaces and solids into complexes of small elements such as triangles, quadrilaterals, tetrahedra or hexahedra is one of the major ingredients for many computational algorithms. Applications range from rendering, most notably in computer games, to computational science, in particular for the numerical solution of partial differential equations on complex domains for the study of physical phenomena. These various areas lead to a broad range of different requirements on a mesh library, which certainly cannot be fulfilled by a single, predetermined datastructure. Unlike other mesh libraries, `ViennaGrid` provides the abililty to easily adjust the internal representation of meshes, while providing a uniform interface for the storage and access of data on mesh elements as well as STL-compatible iteration over such elements.

As example, consider the basic building block of triangular meshes, a triangle: The three vertices fully define the shape of the triangle, the edges can be derived from vertices if a common reference orientation of the triangles is provided. Depending on the underlying algorithm, edges of the triangle may or may not be of interest:

- Consider a class `triangle`, holding the three vertices only. A triangular mesh is then some array or list of `triangle`s and an algorithm `algo1` working only on vertices on a per-cell basis can be executed efficiently. An example for such an algorithm is the assembly of a linear, nodal finite element method.
- An `algo2` may need to have global edge information available, i.e. only one instance of an interfacing edge of two triangles should exist in an explicit manner. Thus, storing the edges globally in the domain will allow the use of `algo2`, but will at the same time introduce unnecessary edge information for `algo1`. Finite volume schemes can be seen as an example for this second type of algorithms.
- A third algorithm `algo3` may need global edge information, as well as information about the local orientation of edges with respect to each triangular cell. In such a case it may be preferred to additionally store mappings from global orientations to local orientations of the edges on each triangle if fast execution is desired. Such an additional storage of orientations will render the datastructure well suited for `algo3`, but less suited for `algo1` and `algo2`. An example for such a third type of algorithm are to some extent high-order finite element methods.

The situation for tetrahedral meshes is even more complicated, because additional orientation issues of shared facets come into play.

The aim of `ViennaGrid` is to be highly customizable such that all three algorithms outlined above can be supported with an optimal data layout. In particular, `ViennaGrid` allows for a convenient specification of the storage of elements, in particular which boundary elements are stored inside the domain as well as which topological information is stored



(a) Store only vertices globally, do not store edges.

(b) Store vertices and edges globally.

Figure 1: Two storage schemes for a triangle and the underlying triangular mesh datastructure.

on each element. After a brief introduction into the nomenclature used in this manual in Chapter 2, the configuration of the data storage layout is explained in Chapter **??**, and the basic steps required to fill a mesh with cells is explained in Chapter 4.

In addition to a high flexibility with respect to the underlying data structures, `ViennaGrid` provides STL-compatible iterators and access to sub-elements of a mesh, cf. Chapter 5. This allows to write generic code that is a-priori independent of the underlying spatial dimension. In particular, a single implementation for algorithms operating in multiple dimensions and using different mesh types (triangular, hexahedral, etc.) can be obtained.

One of the strengths of `ViennaGrid` is the generic facility provided for storing arbitrary quantities on a mesh, cf. Chapter 6. This is achieved by the use of set of concepts which provide uniform access to data through so called accessors or fields.

A typical requirement for a meshing library is mesh refinement. This is in particular of interest for computational science, where singularities near corners need to be resolved sufficiently well. `ViennaGrid` provides both uniform and adaptive refinement algorithms, cf. Chapter 7, where also other geometric algorithms such as Voronoi information is covered.

Input/Output facilities are discussed in Chapter 8. Some of the library internals are discussed in Chapter 9 and design decisions are outlined in Chapter 10.

There are of course a number of other free software libraries having functional overlap with `ViennaGrid`. We give a brief discussion of the pros and cons of selected libraries in the following. This should allow potential users of our library to get a better feeling of what to expect and what not to expect from `ViennaGrid`. We have carefully checked the documentation of each project, but clearly cannot guarantee that all information is fully accurate.

- **CGAL** [1]: The focus of the Computational Geometry Algorithms Library (`CGAL`) is on geometrical algorithms such as the computation of convex hulls of point sets. It offers a mesh generation facility and provides iterators over cell vertices. However, the storage of quantities and the convenient traversal of mesh elements is not provided.
- **DUNE** [2]: `DUNE` follows a similar approach for the generic representation of meshes. It provides support for conforming and non-conforming grids, as well as support for parallel and distributed meshes. However, unlike `ViennaGrid`, we could not find any mechanism providing a convenient means to store data on mesh elements (users are essentially required to handle their data themselves), and for the customization about the internal storage of mesh elements.

- **GrAL** [3]: The Grid Algorithms library (`GrAL`) provides mesh data structured and algorithms operating on them. A number of principles used in `ViennaGrid` such as *n-cells* already show up in `GrAL` as *k-Elements*. The library does not provide any facility to store data on mesh elements. Mesh refinement is also not provided.

- **libmesh** [4]: The `libmesh` library is not only a mesh library, but also a framework for numerical simulations. Since `ViennaGrid` is designed to be as general as possible without prematurely restricting to a particular application, we only compare the parts in `libmesh` related to mesh handling. `libmesh` supports one-, two- and three-dimensional meshes and also allows to generate meshes for simple domains. Iterations over elements of a mesh are carried out in a runtime manner, thus causing potential overhead. One of the strengths of `libmesh` is the support for mesh refinement and parallel computations. Support for user-defined data on mesh elements is also provided.

- **OpenMesh** [5]: `OpenMesh` provides a generic datastructure for representing and manipulating polygonal meshes. The main goals are flexibility, efficiency and easy-to-use. Similar to `ViennaGrid`, generic programming paradigms are used. `OpenMesh` allows to store custom data of arbitrary type on mesh elements, but it seems to rely on potentially slow string comparisons at run-time to retrieve the data. Moreover, `OpenMesh` is specifically designed for surface (i.e. non-volumetric) meshes, and thus only the concepts of vertices, edges and faces are used.

- **trimesh2** [6]: `trimesh2` is a C++ library that is particularly designed for triangular meshes in 3D only. It explicitly targets efficiency, possibly at the expense of some generality. We could not find further information for a comparison with `ViennaGrid` from the documentation provided.

- **VCGlib** [7]: `VCGlib` processes triangular and tetrahedral meshes. Similar to `OpenMesh`, `VCGlib` uses the concepts of vertices, edges and faces only, so the processing of volume meshes is hampered. Again similar to `OpenMesh`, the provided facility to store data on mesh elements relies on potentially slow string comparisons.

Chapter 1

Installation

This description has not been updated for ViennaGrid 1.1.0 yet!

This chapter shows how ViennaGrid can be integrated into a project and how the examples are built. The necessary steps are outlined for several different platforms, but we could not check every possible combination of hardware, operating system, and compiler. If you experience any trouble, please write to the mailing list at

viennagrid-support@lists.sourceforge.net

1.1 Dependencies

- A recent C++ compiler (e.g. GCC version 4.2.x or above and Visual C++ 2005 or above are known to work)
- ViennaData [8], version 1.0.1 or above. To make ViennaGrid self-contained, a copy of the ViennaData sources is available in the viennadata/ folder.
- CMake [9] as build system (optional, but recommended for building the examples)

1.2 Generic Installation of ViennaGrid

Since ViennaGrid is a header-only library, it is sufficient to copy the viennagrid/ source folder either into your project folder or to your global system include path. If you do not have ViennaData installed, proceed in the same way for the respective source folder viennadata/.

On Unix-like operating systems, the global system include path is usually /usr/include/ or /usr/local/include/. On Windows, the situation strongly depends on your development environment. We advise to consult the documentation of the compiler on how to set the include path correctly. With Visual Studio 9.0 this is usually something like C:\Program Files\Microsoft Visual Studio 9.0\VC\include and can be set in Tools -> Options -> Projects and Solutions -> VC++-Directories.

File	Purpose
tutorial/accessor.cpp	Demonstrates the use of accessors, cf. Chapter 6
tutorial/algorithms.cpp	Demonstrates the algorithms provided, cf. Chapter 7
tutorial/cboundary_iteration.cpp	Shows how to iterate over co-boundary elements, cf. Chapter 5
tutorial/domain_setup.cpp	Fill a domain with cells, cf. Chapter 4
tutorial/element_erase.cpp	Demonstrates how to erase single elements from a domain
tutorial/finite_volumes.cpp	Generic implementation of the finite volume method (assembly)
tutorial/interface.cpp	Demonstrates how the interface algorithm works, cf. Chapter 7
tutorial/io.cpp	Explains input-output operations, cf. Chapter 8
tutorial/iterators.cpp	Shows how the domain and segments can be traversed, cf. Chapter 5
tutorial/multi_segment.cpp	Explains multi-segment capabilities, cf. Chapter 4
tutorial/neighbour_iteration.cpp	Shows how to iterate over neighbour elements, cf. Chapter 5
tutorial/polygon.cpp	ViennaGrid also supports Polygons, cf. Chapter 4
tutorial/segments.cpp	Shows how to use segmentations and segments, cf. Chapter 4

Table 1.1: Overview of the sample applications in the examples/ folder

1.3 Building the Examples and Tutorials

For building the examples, we suppose that CMake is properly set up on your system. The various examples and their purpose are listed in Tab. 1.1.

1.3.1 Linux

To build the examples, open a terminal and change to:

```
$> cd /your-ViennaGrid-path/build/
```

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> make
```

to build the examples. If desired, one can build each example separately instead:

```
$> make algorithms     #builds the algorithms tutorial
```

Speed up the building process by using multiple concurrent jobs, e.g. `make -j4`.

1.3.2 Mac OS X

The tools mentioned in Section 1.1 are available on Macintosh platforms too. For the `GCC` compiler the Xcode [10] package has to be installed. To install `CMake`, external portation tools such as Fink [11], DarwinPorts [12], or MacPorts [13] have to be used.

The build process of `ViennaGrid` is similar to Linux.

1.3.3 Windows

In the following the procedure is outlined for `Visual Studio`: Assuming that `CMake` is already installed, `Visual Studio` solution and project files can be created using `CMake`:

- Open the `CMake` GUI.
- Set the `ViennaGrid` base directory as source directory.
- Set the `build/` directory as build directory.
- Click on 'Configure' and select the appropriate generator (e.g. `Visual Studio 9 2008`)
- Click on 'Generate' (you may need to click on 'Configure' one more time before you can click on 'Generate')
- The project files can now be found in the `ViennaGrid` build directory, where they can be opened and compiled with Visual Studio (provided that the include and library paths are set correctly, see Sec. 1.2).

Note that the examples should be executed from the `build/Debug` and `build/Release` folder respectively in order to access the correct input files.

Chapter 2

Main Entities

In the following, the main entities of ViennaGrid are explained. The nomenclature essentially follows the convention from topology and can partly be found in other mesh libraries. Note that the purpose of this manual is not to give exact definitions from the field of geometry or topology, but rather to establish the link between abstract concepts and their representation in code within `ViennaGrid`. First, geometrical objects are discussed, then topological objects and finally complexes of topological objects.

2.1 Points (Geometrical Objects)

The underlying space in `ViennaGrid` is the m-dimensional Euclidian space $\mathbb{E}^m$, which is identified with the real coordinate space $\mathbb{R}^m$ in the following. A *point* refers to an element $\boldsymbol{x}$ in $\mathbb{R}^m$ and does not carry any topological information. On the other hand, a point equivalently refers to the vector from the origin pointing to $\boldsymbol{x}$.

Given a configuration class `Config` for `ViennaGrid` (cf. Chap. **??**), a point is defined and manipulated as follows:

```
using namespace viennagrid;

// obtain the point type from a meta-function
typedef result_of::point<Config>::type    PointType;
// For a three-dimensional Cartesian space (double precision),
// the type of the point is returned as
// point_t<double, cartesian_cs<3> >

// Instantiate two points:
PointType p1(0, 1, 2);
PointType p2(2, 1, 0);

// Add/Subtract points:
PointType p3 = p1 + p2;
std::cout << p1 - 2.0 * p3 << std::endl;
std::cout << "x-coordinate of p1: " << p1[0] << std::endl;
```

The operators `+`, `-`, `*`, `/`, `+=`, `-=`, `*=` and `/=` can be used in the usual mnemonic manner. `operator[]` grants access to the individual coordinate entries and allows for a direct manipulation.

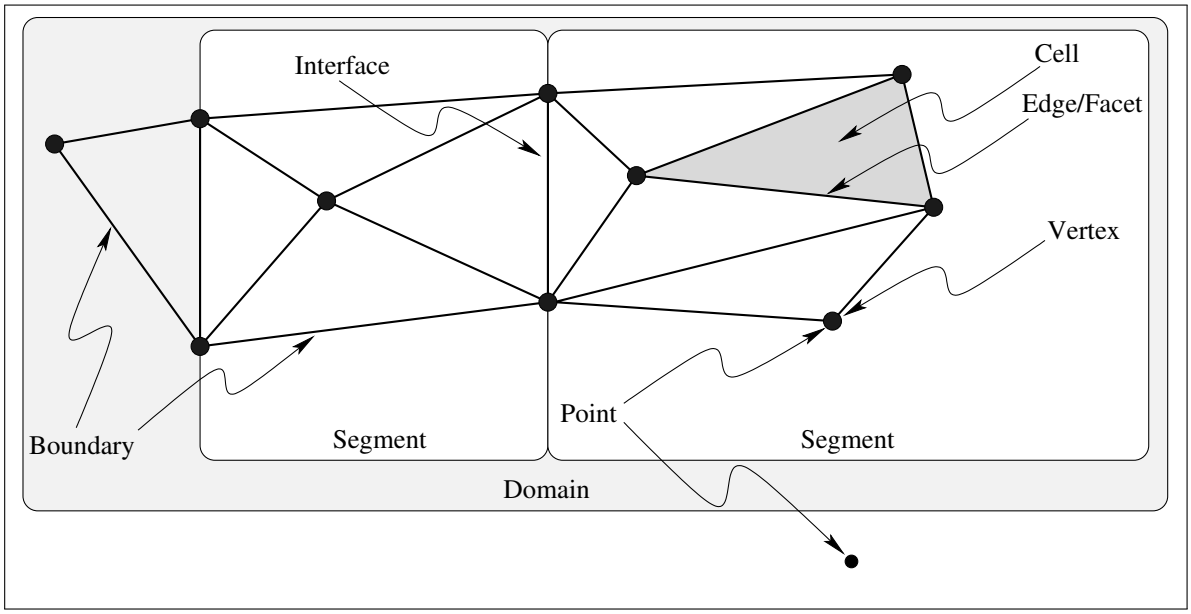


Figure 2.1: Overview of the main entities in `ViennaGrid` for a triangular mesh. A point refers to any location in the geometric space and does not carry topological information.

Aside from the standard Cartesian coordinates, `ViennaGrid` can also handle polar, spherical and cylindrical coordinate systems. This is typically defined globally within the configuration class `Config` for the whole domain, and the meta-function in the previous snippet creates the correct point type. However, if no global configuration class is available, the point types can be obtained as

```
typedef point<double, 1>                    CartesianPoint1d;
typedef point<double, 2>                    CartesianPoint2d;
typedef point<double, 2, polar_cs>          PolarPoint2d;
typedef point<double, 3>                    CartesianPoint3d;
typedef point<double, 3, spherical_cs>      SphericalPoint3d;
typedef point<double, 3, cylindrical_cs>    CylindricalPoint3d;
```

Conversions between the coordinate systems are carried out implicitly whenever a point is assigned to a point with a different coordinate system:

```
CylindricalPoint3d p1(1, 1, 5);
CartesianPoint3d p2 = p1; //implicit conversion
```

An explicit conversion to the Cartesian coordinate system is provided by the free function `to_cartesian()`, which allows for the implementation of generic algorithms based on Cartesian coordinate systems without tedious dispatches based on the coordinate systems involved.

For details on the coordinate systems, refer to the reference documentation in `doc/doxygen/`.

Since all coordinate systems refer to a common underlying Euclidian space, the operator overloads remain valid even if operands are given in different coordinate systems. In such a case, the coordinate system of the resulting point is given by the coordinate system of the left hand side operand:

```
CylindricalPoint3d p1(1, 1, 5);
CartesianPoint3d p2 = p1; //implicit conversion

// the result of p1 + p2 is in cylindrical coordinates
CylindricalPoint3d p3 = p1 + p2;

// the result of p2 + p1 is in Cartesian coordinates,
// but implicitly converted to cylindrical coordinates:
CylindricalPoint3d p4 = p2 + p1;
```

For additional algorithms acting on points, e.g. `norm()` for computing the norm/length of a vector, please refer to Chapter 7.

ViennaGrid is not restricted to one, two or three geometric dimensions! Cartesian coordinate systems for arbitrary dimensions are available.

2.2 Elements (Topological Objects)

While the point type defines the underlying geometry, elements define the topological connections among distinguished points. Each of these distinguished points is called a *vertex* and describes the corners or intersection of geometric shapes. Vertices are often also referred to as the *nodes* of a mesh.

An *edge* or *line* is a line segment joining two vertices. Note that this is a topological characterization – the underlying geometric space can have arbitrary dimension.

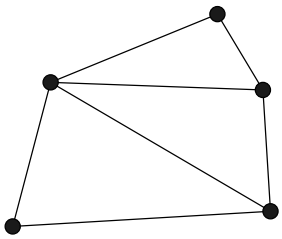
A *cell* is an element of maximum topological dimension N within the set of elements considered. The topological dimension of cells can be smaller than the underlying geometric space, which is for example the case in triangular surface meshes in three dimensions. Note that the nomenclature used among scientists is not entirely consistent: Some refer to topologically three-dimensional objects independent from the topological dimension of the full mesh as cells, which is not the case here.

The surface of a cell consists of *facets*, which are objects of topological dimension $N - 1$. Some authors prefer the name *face*, which is by other authors used to refer to objects of topological dimension two. Again, the definition of a facet refers in `ViennaGrid` to topological dimension $N - 1$.

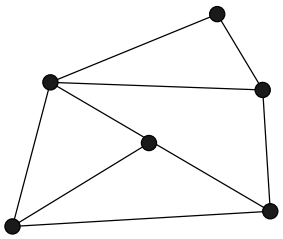
Boundary elements are elements which represent a boundary of another element. For example a triangle is a boundary element of a tetrahedron. But not only the direct boundaries are boundary elements in `ViennaGrid`, also a boundary element of boundary element of an element is a boundary element of that element: a vertex and a line is a boundary element of a tetrahedron as well.

A brief overview of the corresponding meanings of vertices, edges, facets and cells is given in Tab. 2.1. Note that edges have higher topological dimension than facets in the one-dimensional case, while they coincide in two dimensions. Refer also to Fig. 2.1.

`ViennaGrid` supports 3 kinds of element types: simplices, hypercubes and special elements. Theoretically, simplices and hypercube with arbitrary dimension are supported but they are not explicitly defined. Special elements are polygons and piecewise linear complexes (PLCs). Each element type can be identified with its own tag. Tab. 2.2 gives an



(a) Conforming cell complex.



(b) Non-conforming cell complex.

Figure 2.2: Illustration of conforming and non-conforming cell complexes. The vertex in the center of (b) intersects an edge in the interior, violating the conformity criterion.

overview of all supported element types and their tags.

2.3 Domain

A *domain* Ω is the top level object in `ViennaGrid` and is a container for its topological elements. There are no topological restrictions on the elements inside a domain.

A typical use-case for a domain is to store a cell complex. We characterize a cell complex as a collection of topological elements, such that the intersection of two elements (maybe of different type) e_0 and e_1 is another element e_i from the cell complex.

It should be noted that `ViennaGrid` allows the use of conforming and non-conforming complexes. Here, a *conforming* complex is characterized by the property that the intersection element e_i from above is a boundary element from both of the element e_0 and the element e_1.

If this is not the case, the complex is denoted *non-conforming*, cf. Fig. 2.2.

The instantiation of a `ViennaGrid` domain object requires a configuration class `Config`, as will be discussed in Chapter **??**. Given such a class, the domain type is retrieved and the domain object constructed as

```
using namespace viennagrid;

// Type retrieval, method 1: use meta-function (recommended)
typedef result_of::domain<Config>::type     DomainType;

// Type retrieval, method 2: direct (discouraged, may change in future
   versions)
```

	1-d	2-d	3-d	n-d
Vertex	Point	Point	Point	Point
Edge	Line	Line	Line	Line
Facet	Point	Line	Triangle, etc.	$n-1$-Simplex, etc.
Cell	Line	Triangle, etc.	Tetrahedron, etc.	n-Simplex

Table 2.1: Examples for the vertices, edges, facets and cells for various topological dimensions.

```
typedef domain_t<Config>     DomainType

DomainType domain; //create the domain object
```

2.4 Segmentation and Segment

A *segment* Ω_i refers to a subset of the elements in a domain Ω. Unlike a domain, a segment is not a container for its elements. Instead, only references (pointers) to the elements in the domain are stored. In common C++ language, a *segment* represents a so-called *view* on the domain.

A segmentation represents a collection of segments. The typical use-case for a segmentation is the decomposition of the domain into pieces of a common property. For example, a solid consisting of different materials can be set up in `ViennaGrid` such that each regions of the same material are represented in a common segment.

	Dimension	Generic Tag	Element Tag
Simplex	n	`simplex_tag<n>`	`simplex_tag<n>`
Hypercube	n	`hypercube_tag<n>`	`hypercube_tag<n>`
Vertex	0	`simplex_tag<0>`	`vertex_tag`
Line or **Edge**	1	`simplex_tag<1>`	`line_tag`, `edge_tag`
Triangle	2	`simplex_tag<2>`	`triangle_tag`
Tetrahedron	3	`simplex_tag<3>`	`tetrahedron_tag`
Quadrilateral	2	`hypercube_tag<2>`	`quadrilateral_tag`
Hexahedron	3	`hypercube_tag<3>`	`hexahedron_tag`
Polygon	2	`polygon_tag`	`polygon_tag`
PLC	2	`plc_tag`	`plc_tag`

Table 2.2: Element types and their tags

Chapter 3

Type Basics

A domain and all its topological elements as well as the underlying geometric space are specified in a common configuration class. The setup of such a configuration class is explained in detail in Section 3.1.

3.1 Domain Configuration

A domain is configured with a configuration class. This configuration class defines the topology as well as the geometry of a domain. `ViennaGrid` supports a bunch of different configurations listed in the tables 3.1, 3.2 and 3.3.

Although `ViennaGrid 1.1.0` supports flexible configuration of the domain class, a convenient public interface is not provided so far. A next version will access this issue.

With a configuration, a domain can be defined using the meta-function `domain`.

```
using namespace viennagrid;

typedef config::trianglular_3d Config;

// Type retrieval, method 1: use meta-function
typedef result_of::domain<Config>::type     DomainType;

// Type retrieval, method 1: use direct typedef
```

`line_1d`	Domain with lines and vertices in 1d
`line_2d`	Domain with lines and vertices in 2d
`line_3d`	Domain with lines and vertices in 3d
`triangular_2d`	Domain with triangles, lines and vertices in 2d
`triangular_3d`	Domain with triangles, lines and vertices in 3d
`tetrahedral_3d`	Domain with tetrahedrons, triangles, lines and vertices in 3d

Table 3.1: Configurations of simplex domains

`quadrilateral_2d`	Domain with quadilaterals, lines and vertices in 2d
`quadrilateral_3d`	Domain with quadilaterals, lines and vertices in 3d
`hexahedral_3d`	Domain with hexahedrons, quadilaterals, lines and vertices in 3d

Table 3.2: Configurations of hypercube domains

`polygonal_2d`	Domain with polygons, lines and vertices in 2d
`polygonal_3d`	Domain with polygons, lines and vertices in 3d
`plc_2d`	Domain with PLCs, lines and vertices in 2d
`plc_3d`	Domain with PLCs, lines and vertices in 3d

Table 3.3: Configurations of special domains

```
typedef trianglular_3d_domain DomainType;

DomainType domain; //create the domain object
```

Using a domain type a segmentation and its segments can be defined.

```
using namespace viennagrid;

// Type retrieval, method 1: use meta-function
typedef result_of::segmentation<DomainType>::type     SegmentationType;
typedef result_of::segmentation<SegmentationType>::type     SegmentType;

// Type retrieval, method 1: use direct typedef
typedef trianglular_3d_segmentation SegmentationType;
typedef trianglular_3d_segment SegmentType;

DomainType domain; // create the domain object
SegmentationType segmentation(domain); // segmentation needs the domain
    object as constructor argument
```

3.2 Elements and Handles

A central topic with `ViennaGrid` are elements. Elements types can be obtained by using the `element` meta-function and the corresponding element tag

```
using namespace viennagrid;

// obtain element type from domain type
typedef result_of::element<DomainType, vertex_tag>::type VertexType;
// this meta function might fail if there is no tetrahedron in the domain
typedef result_of::element<DomainType, tetrahedron_tag>::type
    TetrahedronType;

// obtain element type from another element
typedef result_of::element<TetrahedronType, line_tag>::type LineType;
// the same element as above
```

```
typedef result_of::element<TetrahedronType, edge_tag>::type LineType;

// this meta function will fail because there is no tetrahedron in a line
typedef result_of::element<LineType, tetrahedron_tag>::type
   TetrahedronType;

// elements can also be obtained from segmentations or segments
typedef result_of::element<SegmentationType, triangle_tag>::type
   TriangleType;
typedef result_of::element<SegmentType, vertex_tag>::type VertexType;
```

For each supported type there are also shortcut meta-functions available.

```
using namespace viennagrid;

// obtain element type from domain type
typedef result_of::vertex<DomainType>::type VertexType;
// this meta function might fail if there is no tetrahedron in the domain
typedef result_of::tetrahedron<DomainType>::type TetrahedronType;

// obtain element type from another element
typedef result_of::line<TetrahedronType>::type LineType;

// this meta function will fail because there is no tetrahedron in a line
typedef result_of::tetrahedron<LineType>::type TetrahedronType;

// elements can also be obtained from segmentations or segments
typedef result_of::triangle<SegmentationType>::type TriangleType;
typedef result_of::vertex<SegmentType>::type VertexType;
```

Often elements are not accessed directly but through a handle instead. For example the boundary elements are stored using their handles within an element. Handles can also be simple obtained by the meta-function `element` or by the shortcut meta-functions.

```
using namespace viennagrid;

// obtain element handle type from domain type
typedef result_of::handle<DomainType, vertex_tag>::type VertexHandleType;
typedef result_of::vertex_handle<DomainType>::type VertexHandleType;
// this meta function might fail if there is no tetrahedron in the domain
typedef result_of::handle<DomainType, tetrahedron_tag>::type
   TetrahedronHandleType;
typedef result_of::tetrahedron_handle<DomainType>::type
   TetrahedronHandleType;

// obtain element handle type from another element (obtaining handles from
    another handle is not supported)
typedef result_of::handle<TetrahedronType, line_tag>::type LineHandleType;
typedef result_of::line_handle<TetrahedronType>::type LineHandleType;
// the same element as above
typedef result_of::handle<TetrahedronType, edge_tag>::type LineHandleType;
typedef result_of::edge_handle<TetrahedronType, edge_tag>::type
   LineHandleType;

// this meta function will fail because there is no tetrahedron in a line
typedef result_of::handle<LineType>::type TetrahedronHandleType;
```

```
typedef result_of::tetrahedron_handle<LineType>::type
    TetrahedronHandleType;

// elements can also be obtained from segmentations or segments
typedef result_of::handle<SegmentationType, triangle_tag>::type
    TriangleHandleType;
typedef result_of::triangle_handle<SegmentationType>::type
    TriangleHandleType;

typedef result_of::handle<SegmentType>::type VertexHandleType;
typedef result_of::vertex_handle<SegmentType>::type VertexHandleType;
```

Chapter 4

Domain and Segment Setup

This chapter explains how a ViennaGrid domain can be filled with cells. Since this is a necessary step in order to do anything useful with ViennaGrid, it is explained in detail in the following. Existing file readers and writers are explained in Chapter 8.

A tutorial code can be found in examples/tutorial/domain_setup.cpp.

In the following, the simple triangular mesh shown in Fig. 4.1 will be set up. Thus, the domain type using the provided configuration class for two-dimensional triangular classes will be used:

```
typedef viennagrid::config::triangular_2d
    ConfigType;
typedef viennagrid::result_of::domain<ConfigType>::type
    DomainType;
typedef viennagrid::result_of::segmentation<DomainType>::type
    SegmentationType;
typedef viennagrid::result_of::segment<SegmentationType>::type
    SegmentType;

DomainType domain;                      // The domain to be set up in the
    following
SegmentationType segmentation(domain); // The segmentation for the example
     mesh in Figure 4.1

SegmentType segment_0 = segmentation.make_segment(); // Segment 0, the
    left one
SegmentType segment_1 = segmentation.make_segment(); // Segment 1, the
    right one
```

If these lines are used inside a template class or template function, an additional **typename** needs to be put after **typedef** in the second line. The created domain object will be filled with vertices and cells in the following.

4.1 Adding Vertices to a Domain or Segment

Since vertices carry geometric information by means of an associated point type, we first obtain the respective point type from the meta-function point<>:

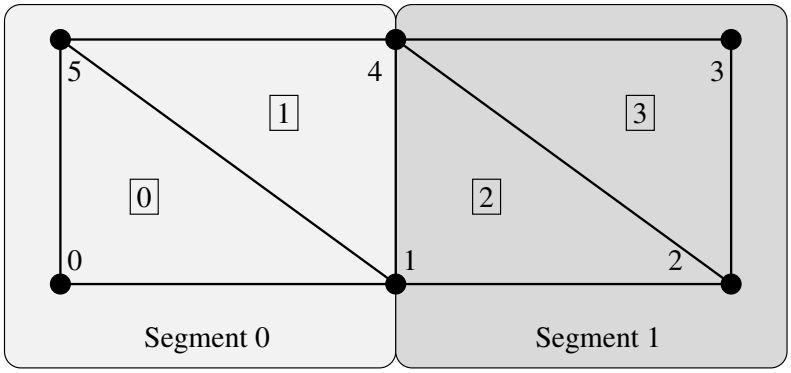


Figure 4.1: An exemplary mesh for demonstrating the use of `ViennaGrid`. Cell-indices are boxed.

```
typedef viennagrid::result_of::point<DomainType>::type    PointType;
```

This already allows us to add the vertices shown in Fig. 4.1 one after another to the domain by using `make_vertex`. This function returns a handle to the created vertex. We might need this handle so we have to define a type for our vertex handle:

```
typedef viennagrid::result_of::handle<DomainType, viennagrid::vertex_tag
    >::type    VertexHandleType;
```

Much simpler is the use of the convenience meta function `vertex_handle`

```
typedef viennagrid::result_of::vertex_handle<DomainType>::type
    VertexHandleType;
```

One example to add the first vertex is to push the respective point to the domain:

```
PointType p(0,0);
VertexHandleType vh0 = viennagrid::make_vertex(domain, p);  // add vertex
    #0
```

The member function `make_vertex` adds the vertex to the domain and assigns an ID.

To push the next vertex, one can either reuse the existing vertex:

```
p[0] = 1.0;
p[1] = 0.0;
VertexHandleType vh1 = viennagrid::make_vertex(domain, p);  // add vertex
    #1
```

or directly construct the respective points in-place:

```
VertexHandleType vh2 = viennagrid::make_vertex(domain, PointType(2,0));
    // add vertex #2
VertexHandleType vh3 = viennagrid::make_vertex(domain, PointType(2,1));
    // add vertex #3
VertexHandleType vh4 = viennagrid::make_vertex(domain, PointType(1,1));
    // add vertex #4
VertexHandleType vh5 = viennagrid::make_vertex(domain, PointType(0,1));
    // add vertex #5
```

It is also possible to explicitly specify the ID for a new vertex:

```
typedef viennagrid::result_of::vertex<DomainType>::type   VertexType;
typedef viennagrid::result_of::id<VertexType>::Type       VertexIDType;

VertexHandleType some_vh = viennagrid::make_vertex(domain, VertexIDType
    (42), PointType(0,0));  // add some vertex with ID 42
```

If the geometric location of a particular vertex needs to be adjusted at some later stage, the free function `point` can be used. To e.g. access vertex `#4`,

```
viennagrid::point(domain, vh4)
```

returns a reference to the respective element, which can then be manipulated accordingly.

Vertices can also be created in a segment.

```
VertexHandleType some_vh = viennagrid::make_vertex(segment_0, PointType
    (-1,-1));  // adding some other vertex to segment 0
```

In this case the vertex is created in the domain and a handle is stored in segment 0.

4.2 Adding Cells to a Domain or Segment

To optain the tag of the cell element use the meta-function `cell_tag`.

```
typedef viennagrid::result_of::cell_tag<DomainType>::type    CellTag;
```

The type of a cell in a domain is obtained by the `element` meta-function using the cell tag or directly by using the `cell` meta-function.

```
typedef viennagrid::result_of::element<DomainType, CellTag>::type
    CellType;
typedef viennagrid::result_of::cell<DomainType>::type
    CellType;
```

Note that an additional `typename` is required if these lines are put inside a template class or template function.

If more than one element has highest topological dimension within the domain, this meta function will fail due to ambiguity.

An overview of the generic procedure for adding a cell to a domain or segment is the following:

- Set up an array holding the handles to the vertices *in the domain*. Do not use handles to vertices defined outside the domain.
- Use `make_element` to create the element within the domain or segment

Thus, the array of handles to vertices is created as usual:

```
VertexHandleType cell_vertices[3];
```

Instead of hard-coding the number of vertices for a triangle, one can instead use

```
VertexHandleType cell_vertices[viennagrid::boundary_elements<CellTag,
    viennagrid::vertex_tag>::num];
```

`boundary_elements<CellTag,viennagrid::vertex_tag>::num` will only work on elements with static vertex count, e.g. triangles, tetrahedrons, ... This method will not work with polygons or PLCs

Next, the vertex addresses for the first triangle are stored:

```
cell_vertices[0] = vh0; // vertex #0
cell_vertices[1] = vh1; // vertex #1
cell_vertices[2] = vh5; // vertex #5
```

Make sure that the correct cell orientation is used, cf. Appendix!

If the vertex handles are not present at this point you can search them with their ID. Keep in mind that due to the underlying data structure this search might has linear runtime complexity.

```
typedef viennagrid::result_of::vertex<DomainType>::type   VertexType;
typedef viennagrid::result_of::id<VertexType>::Type       VertexIDType;

cell_vertices[0] = viennagrid::find_by_id(domain, VertexIDType(0)); //
    vertex #0
cell_vertices[1] = viennagrid::find_by_id(domain, VertexIDType(1)); //
    vertex #1
cell_vertices[2] = viennagrid::find_by_id(domain, VertexIDType(5)); //
    vertex #5
```

If you can ensure that the vertex with ID 0 was created first, the vertex with ID 1 was created second and so on, you can use direct random access:

```
typedef viennagrid::result_of::vertex<DomainType>::type   VertexType;
typedef viennagrid::result_of::id<VertexType>::Type       VertexIDType;

cell_vertices[0] = viennagrid::vertices(domain)[0]; // vertex #0
cell_vertices[1] = viennagrid::vertices(domain)[1]; // vertex #1
cell_vertices[2] = viennagrid::vertices(domain)[5]; // vertex #5
```

Now we are ready to create the element within the domain or segment. The generic function `make_element` requires the element-to-create type as well as begin and end iterator of a vertex container.

```
viennagrid::make_element<CellType>(segment_0, cell_vertices, cell_vertices
    +3);
```

In our case the following shortcut function can also be used, the handles are passed directly.

```
viennagrid::make_triangle(segment_0, vh0, vh1, vh5);
```

As for vertices, cells have to be pushed in ascending order in order to get the correct IDs assigned. Note that the cell is always stored inside the domain - a segment keeps a handle to the cell as well as its boundary cells only.

In the same way the other triangles are pushed to the respective segment. For triangle #3, the code is

```
viennagrid::make_triangle(segment_1, vh2, vh3, vh4);
```

Creating element with explicit ID is also supported:

```
typedef viennagrid::result_of::id<CellType>::Type       CellIDType;

viennagrid::make_element_with_id<CellType>(segment_0, cell_vertices,
   cell_vertices+3, CellIDType(42));
```

Chapter 5

Iterators

The (possibly nested) iteration over elements of a mesh is one of the main ingredients for a plethora of algorithms. Consequently, `ViennaGrid` is designed such that these iterations can be carried out in a unified and flexible, yet efficient manner.

At the heart of the various possibilities is the concept of a *range*. A range provides iterators for accessing a half-open interval `[first,one_past_last)` of elements and provides information about the number of elements in the range. However, a range does not 'own' the elements which can be accessed through it [14]. Employing the range-concept, any iteration over elements in `ViennaGrid` consists of two phases:

- Initialize the range of elements over which to iterate.
- Iterate over the range using the iterators returned by the member functions `begin()` and `end()`.

For convenience, a range may also provide access to its elements using **`operator`**`[]` (i.e. random access) and thus allowing an index-based iteration. The conditions for random access availability will also be given in the following.

A tutorial code can be found in `examples/tutorial/iterators.cpp`.

5.1 Elements in a Domain or Segment

As usual, the first step is to obtain the types for the range and the respective iterator. To iterate over all elements of a `domain` of type `DomainType`, the types can be obtained from the `element_range` and `iterator` meta-functions:

```
using namespace viennagrid;

//non-const:
typedef result_of::element_range<DomainType, ElementTag>::type
   ElementRange;
typedef result_of::iterator<ElementRange>::type          ElementIterator;
```

For segments, the occurances of `DomainType` and `domain` have to be replaced by `SegmentType` and `segment` here and in the following. If **`const`**-access to the elements is sufficient, the

metafunction `const_element_range` should be used instead of `element_range`. For instance, the required types for a **`const`**-iteration over vertices is given by

```
//const:
typedef result_of::const_element_range<DomainType, vertex_tag>::type
   ConstVertexRange;
typedef result_of::iterator<ConstVertexRange>::type
   ConstVertexIterator;
```

The next step is to set up a range object using the `elements` function. The general case of elements is handled by

```
NCellRange elements = viennagrid::elements<ElementTag>(domain);
```

For the example of const-iteration over vertices, this results in

```
ConstVertexRange vertices = viennagrid::elements<vertex_tag>(domain);
```

Since the left hand side operand already contains the topological dimension of elements over which to iterate, the template argument to the `elements` function can also be omitted:

```
ConstVertexRange vertices = viennagrid::elements(domain);
```

While the advantage of this shorter variant is clearly shorter code and simpler copy&pasting, the disadvantage is that the topological dimension is specified only once in the respective **`typedef`**. The longer variant adds a second check for the use of the correct topological dimension.

Once the range is set up, iteration is carried out in the usual C++ STL manner:

```
for (ElementIterator it = elements.begin();
                     it != elements.end();
                   ++it)
{ /* do something */}
```

For reference, the complete code for printing all vertices of a domain without a **`using namespace`**-directive is:

```
typedef viennagrid::result_of::const_element_range<DomainType, vertex_tag
   >::type   ConstVertexRange;
typedef viennagrid::result_of::iterator<ConstVertexRange>::type
   ConstVertexIterator;

ConstVertexRange vertices = viennagrid::elements(domain);
for (VertexIterator vit = vertices.begin();
                    vit != vertices.end();
                  ++vit)
{  std::cout << *vit << std::endl; }
```

It should be emphasized that this code snippet is valid for arbitrary geometric dimensions and arbitrary domain configurations (and thus cell types). Inside a template function or template class, the **`typename`** keyword needs to be added after each **`typedef`**.

In some cases, e.g. for a parallelization using `OpenMP`[15], it is preferred to iterate over all cells using an index-based for-loop rather than an iterator-based one. If the range is either a vertex range of a domain, or a cell range of a domain or segment, this can be obtained by

```
ElementRange elements = viennagrid::elements<ElementTag>(domain);
for (std::size_t i=0; i<elements.size(); ++i)
{ do_something(elements[i]); }
```

It is also possible to use the range only implicitly:

```
for (std::size_t i=0; i<viennagrid::elements<ElementTag>(domain).size();
    ++i)
{ do_something(viennagrid::elements<ElementTag>(domain)[i]); }
```

However, since the repeated construction of the range object can have non-negligible setup costs, the latter code is not recommended.

In `ViennaGrid 1.1.0`, **`operator`**`[]` is not available for ranges obtained from a domain other than vertex or cell ranges. For segments, **`operator`**`[]` is only available for cell ranges.

Shortcut meta-function are available for ranges too

```
typedef viennagrid::result_of::const_vertex_range<DomainType>::type
    ConstVertexRange;
typedef viennagrid::result_of::iterator<ConstVertexRange>::type
    ConstVertexIterator;

ConstVertexRange vertices = viennagrid::vertices(domain);
for (VertexIterator vit = vertices.begin();
                    vit != vertices.end();
                  ++vit)
{  std::cout << *vit << std::endl; }
```

```
for (std::size_t i=0; i<viennagrid::triangles(domain).size(); ++i)
{ do_something(viennagrid::triangles(domain)[i]); }
```

5.2 Boundary Elements Iteration

In addition to an iteration over all elements of a domain or segment, it may be required to iterate over boundary elements.

As in the previous section, the range and iterator types are obtained from the `element_range` and `iterator` meta-functions:

```
//non-const:
typedef viennagrid::result_of::ncell_range<CellType, 1>::type
    EdgeOnCellRange;
typedef result_of::iterator<EdgeOnCellRange>::type    EdgeOnCellIterator;
```

The **`const`**-version is again obtained by using `const_element_range` instead of `element_range`. Mind that the first argument of `element_range` denotes the enclosing entity (the cell) and the second argument denotes the boundary element tag (`line_tag` or `edge_tag` for an edge), and thus preserves the structure already used for the type retrieval for iterations on the domain.

Iteration is then carried out in the same manner as for a domain, with `element` taking the role of the `domain` in the previous chapter. The following snippet print all edges of a `element`:

```
//Note: ... = viennagrid::elements(element); will also work in the next
    line
EdgeOnCellRange edges_on_cell = viennagrid::elements<viennagrid::edge_tag
    >(element);
for (EdgeOnCellIterator eocit = edges_on_cell.begin();
                        eocit != edges_on_cell.end();
                      ++eocit)
{  std::cout << *eocit << std::endl; }
```

For all topological dimensions, an index-based iteration is possible provided that the storage of the respective boundary elements has not been disabled. The previous code snippet can thus also be written as

```
EdgeOnCellRange edges_on_cell = viennagrid::elements<viennagrid::edge_tag
    >(element);
for (std::size_t i=0; i<edges_on_cell.size(); ++i)
{
  do_something(edges_on_cell[i]);
}
```

or

```
for (std::size_t i=0; i<viennagrid::elements<viennagrid::edge_tag>(element
    ).size(); ++i)
{
  std::cout << viennagrid::elements<viennagrid::edge_tag>(element)[i] <<
      std::endl;
}
```

The use of the latter is again discouraged for reasons of possible non-negligible repeated setup costs of the ranges involved.

Finally, `ViennaGrid` allows for iterations over the vertices of boundary elements of an element in the reference orientation imposed by the element, which is commonly required for ensuring continuity of a quantity along cell interfaces. Note that by default the iteration is carried out along the orientation imposed by the element in the way it is stored globally inside the domain. The correct orientation of vertices with respect to the hosting element is established by the free function `local_vertex()`. For instance, the vertices of a boundary element `boundary_element` at the boundary of an element `element` are printed in local orientation using the code lines

```
for (std::size_t i=0; i<viennagrid::vertices(boundary_element).size(); ++i
    )
  std::cout << viennagrid::local_vertex(element, boundary_element, i) <<
      std::endl;
```

The use of `local_vertex` can be read as follows: For the element `element`, return the vertex of the boundary element `boundary_element` at local position i.

5.3 Coboundary Element Iteration

A frequent demand of mesh-based algorithms is to iterate over so-called *coboundary elements* of an element T. The coboundary elements of an element T are given by all elements of a set Ω, where one of the boundary elements is T. For example, the coboundary edges of a vertex T are all edges where one of the two vertices is T.

In contrast to boundary elements, the number of coboundary elements of an element from the family of simplices or hypercubes is not known at compile time. Another difference to the case of boundary elements is that the number of coboundary elements depends on the set Ω under consideration. Considering the interface edge/facet connecting vertices 1 and 4 in the sample domain from Fig. 4.1, the coboundary triangles within the domain are given by the triangles 1 and 2. However, within segment 0, the set of coboundary triangles is given by the triangle 1 only, while within segment 1 the set of coboundary triangles consists of triangle 2 only. Thus, the use of segments can substantially simplify the formulation of algorithms that act on a subregion of the domain only.

The necessary range types are obtained using the same pattern as in the two previous sections. Assuming that a vertex type `VertexType` is already defined, the range of coboundary edges as well as the iterator are obtained using the `coboundary_range` and `iterator` metafunctions in the `viennagrid` namespace:

```
//non-const:
typedef result_of::coboundary_range<DomainType, VertexType, edge_tag>::
   type  EdgeOnVertexRange;
typedef result_of::iterator<EdgeOnVertexRange>::type EdgeOnVertexIterator;
```

The first argument to `coboundary_range` is the context set Ω, the second is the reference element for the iteration, and the third argument is the element tag of the elements in the range. A range of **`const`**-edges is obtained using the `const_coboundary_range` metafunction instead of the non-**`const`** metafunction `coboundary_range`. Moreover, it shall be noted that an additional **`typename`** keyword is required inside template functions and template classes.

To set up the range object, the `coboundary_elements` function from the `viennagrid` namespace is reused. Unlike in the previous sections, it requires two arguments for setting up a coboundary range: The first argument refers to the enclosing container of elements and must be either a domain or a segment and the second argument is the reference element. The range holding all edges in the `domain` sharing a common vertex `v` is thus set up as

```
EdgeOnVertexRange edges_on_v = viennagrid::coboundary_elements<VertexType,
    edge_tag>(domain, v);
```

If the range should hold only the coboundary edges from a segment `seg`, the above code line has to be modified to

```
EdgeOnVertexRange edges_on_v = viennagrid::coboundary_elements<VertexType,
    edge_tag>(seg, v);
```

An iteration over all edges is then possible in the usual STL-type manner. For example, all coboundary edges of v in the range are printed using the code:

```
for (EdgeOnVertexIterator eovit = edges_on_v.begin();
                          eovit != edges_on_v.end();
                        ++eovit)
```

```
{ std::cout << *eovit << std::endl; }
```

One may also use a shorter form that does not set up the range explicitly:

```
for (EdgeOnVertexIterator eovit = viennagrid::coboundary_elements<
   VertexType, edge_tag>(domain, v).begin();
                          eovit != viennagrid::coboundary_elements<
                              VertexType, edge_tag>(domain, v).end();
                         ++eovit)
{ std::cout << *eovit << std::endl; }
```

Random access, i.e. `operator[]` is available for all topological levels. Thus, the loop above may also be written as

```
for (std::size_t i=0; i<edges_on_v.size(); ++i)
{
  std::cout << edges_on_v[i] << std::endl;
}
```

or

```
for (std::size_t i=0; i<viennagrid::coboundary_elements<VertexType,
    edge_tag>(domain, v).size(); ++i)
{
  std::cout << viennagrid::coboundary_elements<VertexType, edge_tag>(
      domain, v)[i] << std::endl;
}
```

where the latter form is not recommended for reasons of overheads involved in setting up the tempoarary ranges.

Finally, it should be noted that coboundary information is not natively available in the domain datastructure. If and only if for the first time the coboundary elements C_n of an element T, are requested, an iteration over all elements of coboundary type of the domain or segment with nested element T boundary iteration is carried out to collect the topological information. This results in extra memory requirements and additional computational costs, hence we suggest to use boundary iterations over coboundary iterations whenever possible.

Prefer the use of boundary iterations over coboundary iterations to minimize memory footprint.

5.4 Neighbour Element Iteration

Beside coboundary iteration `ViennaGrid 1.1.0` also supports iteration over neighbouring elements of the same type. Two elements are neighbours if they share a connector element of different type (with lower topologic dimension). For example iteration over all neighbouring triangles of a reference triangle with vertex as connector element type. As well as with coboundary iteration, a context domain or segment has to be provided.

The necessary range types are obtained using the same pattern as with cobundary iteration: Assuming that a triangle type `TriangleType` is already defined, the range of

neighbour triangles as well as the iterator are obtained using the `neighbour_range` and `iterator` metafunctions in the `viennagrid` namespace:

```
//non-const:
typedef result_of::neighbour_range<DomainType, TriangleType, vertex_tag>::
   type  NeighbourTriangleRange;
typedef result_of::iterator<NeighbourTriangleRange>::type
   NeighbourTriangleIterator;
```

The first argument to `neighbour_range` is the context set Ω, the second is the reference element type for the iteration, and the third argument is the connector element tag. A range of **`const`**-edges is obtained using the `const_neighbour_range` metafunction instead of the non-**`const`** metafunction `neighbour_range`. Moreover, it shall be noted that an additional **`typename`** keyword is required inside template functions and template classes.

To set up the range object, the `neighbour_elements` function from the `viennagrid` namespace is reused. Unlike in the previous sections, it requires two arguments for setting up a neighbour range: The first argument refers to the enclosing container of elements and must be either a domain or a segment and the second argument is the reference element. The range holding all triangles in the `domain` sharing a common vertex with triangle `t` is thus set up as

```
EdgeOnVertexRange neighbour_triangles_of_t = viennagrid::
   neighbour_elements<TriangleType, vertex_tag>(domain, t);
```

If the range should hold only the neighbour triangles from a segment `seg`, the above code line has to be modified to

```
EdgeOnVertexRange neighbour_triangles_of_t = viennagrid::
   neighbour_elements<TriangleType, vertex_tag>(seg, t);
```

An iteration over all triangles is then possible in the usual STL-type manner. For example, all neighbour triangles of `t` in the range are printed using the code:

```
for (NeighbourTriangleIterator ntit = neighbour_triangles_of_t.begin();
                               ntit != neighbour_triangles_of_t.end();
                             ++ntit)
{ std::cout << *ntit << std::endl; }
```

One may also use a shorter form that does not set up the range explicitly:

```
for (NeighbourTriangleIterator ntit = viennagrid::neighbour_elements<
   TriangleType, vertex_tag>(domain, t).begin();
                               ntit != viennagrid::neighbour_elements<
                                   TriangleType, vertex_tag>(domain, t).end();
                             ++ntit)
{ std::cout << *ntit << std::endl; }
```

Random access, i.e. **`operator`**`[]` is available for all topological levels. Thus, the loop above may also be written as

```
 for (std::size_t i=0; i<neighbour_triangles_of_t.size(); ++i)
 {
   std::cout << neighbour_triangles_of_t[i] << std::endl;
 }
```

or

```
for (std::size_t i=0; i<viennagrid::neighbour_elements<TriangleType,
    vertex_tag>(domain, t).size(); ++i)
{
  std::cout << viennagrid::neighbour_elements<TriangleType, vertex_tag>(
      domain, t)[i] << std::endl;
}
```

where the latter form is not recommended for reasons of overheads involved in setting up the tempoarary ranges.

Chapter 6

Data Storage and Retrieval

One of the central operations whenever dealing with meshes is the storage and the retrieval of data. A common approach is to model vertices, edges and the like as separate classes and add data members to them. ViennaGrid does not follow this approach for three reasons:

1. **Reusability**: As soon as a data member is added to any of these classes, the class is refined towards a particular use case. For example, adding a color data member to a triangle class reduces reusability for e.g. Finite Element methods considerably.
2. **Flexibility**: Whenever a data member needs to be added for a particular functionality, one has to carefully extend the existing class layout. Moreover, it is somewhere between hard to impossible to 'just add a data member for the moment' in a productive environment. Moreover, the class needs to be adjusted if the data type changes.
3. **Efficiency**: A data member that is never used obviously wastes memory. For large numbers of object it might be even advisable to use special containers for data that is relevant for a tiny fraction of all objects only (e.g. domain boundary flags). Apart from reduced memory footprint, the possibly tighter grouping of data allows for better CPU caching.

Previous version of ViennaGrid relies on ViennaData [8] for the storage of data associated with topological objects. ViennaGrid 1.1.0 dropped the dependencies on ViennaData in favour of accessor concepts.

An accessor is a simple class which manages access to data stored on objects. For example one might want to store potential values with type `double` on vertices. An accessor provides an `operator()` returning a reference to the data stored for that object:

```
SomeAccessorType my_accessor;
my_accessor(vertex) = 42.0;
std::cout << my_accessor(vertex) << std::endl;
```

Each class fulfilling the accessor concepts presented in table Tab. 6.1 can be used in ViennaGrid.

Beside accessors, ViennaGrid defines the field concept. A field is similar to an accessor but more suitable for sparse storage. The same concepts as for accessor apply to field concepts with the exepction of the concepts in table Tab.6.2.

Member name	Description
`value_type`	the value type which is stored
`access_type`	the access type on which the da
`reference`	a reference to `value_type`
`const_reference`	a const reference to `value_typ`
`pointer`	a point to `value_type`
`const_pointer`	a const pointer to `value_type`
`bool is_valid() const`	return true if the accessor is in
`pointer find(AccessType const & element)`	searches for the data of elemer
`const_pointer find(AccessType const & element) const`	same as above
`reference operator()(AccessType const & element)`	optains the data for element, if
`const_reference operator()(AccessType const & element) const`	optains the data for element, if
`reference at(AccessType const & element)`	same as `operator()`, throws s
`const_reference at(AccessType const & element) const`	same as `operator()`, throws s

Table 6.1: Accessor concepts

Member name	Description
`value_type operator()(AccessType const & element) const`	optains the data for element, if data

Table 6.2: Field concepts

Normally an accessor or field does not store the data on its own. Instead they reference a container where the data is stored. `ViennaGrid` provides meta-functions and function to define and create accessor or fields from base containers. The example below demonstrates the use of `ViennaGrid` accessors. Fields can be used in a similar way.

```
std::vector<int> some_int_vector; // instancing a simple int vector
// instancing an accessor which uses the int vector
viennagrid::result_of::accessor<std::vector<int>, VertexType>::type
    some_int_accessor(some_int_vector);

some_int_accessor(my_vertex) = -3;

// obtaining a container for storing double values on vertices, std::map
    should be used
typedef viennagrid::result_of::accessor_container<VertexType, double,
    viennagrid::storage::std_map_tag>::type ContainerType;

// instancing a container and an accessor for the container
ContainerType container;
viennagrid::result_of::accessor<ContainerType, VertexType>::type
    container_accessor(container);
```

Chapter 7

Algorithms

This description has not been updated for ViennaGrid 1.1.0 yet!

The iterations and data accessors described in the previous Chapters allow for a plethora of algorithms. Most of them make use of basic functionalities such as the inner product of vectors, or the volume of a n-cell. `ViennaGrid` ships with a number of such basic tools, which are listed in Tab. 7.1 and discussed in the following.

The individual algorithms are located in the `viennagrid/algorithm/` folder. A tutorial covering the algorithms explained in this chapter can be found in `examples/tutorial/algorithms.cpp`.

Make sure to include the respective header-file when using one of the algorithms explained below!

7.1 Point/Vector-Based

This section details algorithms in `ViennaGrid` requiring geometric information only. The point type in `ViennaGrid` should be seen in this context mostly as a vector type rather than a representation of a geometric location, reflecting the duality of points and vectors in the Euclidian space.

7.1.1 Cross Products

The cross-product of two vectors (i.e. `ViennaGrid` points) $p0$ and $p1$ is defined for the three-dimensional space and computed with `ViennaGrid` as

```
viennagrid::cross_prod(p1, p2)
```

The following code calculates and prints the cross-product of the vectors $(1,0,0)^{\mathrm{T}}$ and $(0,1,0)^{\mathrm{T}}$:

```
PointType p0(1, 0, 0);
PointType p1(0, 1, 0);
std::cout << viennagrid::cross_prod(p1, p2) << std::endl; //0 0 1
```

Algorithm	Filename	Interface Function
Cross Product	`cross_prod.hpp`	`cross_prod(a, b)`
Inner Product	`inner_prod.hpp`	`inner_prod(a, b)`
Vector Norms	`norm.hpp`	`norm(a, tag)`
Induced Volume	`spanned_volume.hpp`	`spanned_volume(a, b, ...)`
Centroid computation	`centroid.hpp`	`centroid(element)`
Circumcenter comp.	`circumcenter.hpp`	`circumcenter(element)`
Surface computation	`surface.hpp`	`surface(element)`
Volume computation	`volume.hpp`	`volume(element)`
Boundary detection	`boundary.hpp`	`is_boundary(domseg, element)`
Interface detection	`interface.hpp`	`is_interface(seg1, seg2, element)`
Simplex refinement	`refine.hpp`	`refine(tag, domain)`
Surface computation	`surface.hpp`	`surface(domseg)`
Volume computation	`volume.hpp`	`volume(domseg)`
Voronoi grid	`voronoi.hpp`	`apply_voronoi(domseg, ...)`

Table 7.1: List of algorithms available in `viennagrid/algorithm/` grouped by the objects they are acting on. `a` and `b` denote vectors, `element` refers to an element, `domain` to a domain, `seg1` and `seg2` to segments, and `domseg` to either a domain or a segment.

If the two vectors are given in different coordinate systems, the result vector will have the same coordinate system as the first argument.

7.1.2 Inner Products

Unlike cross products, inner products (aka. dot products) are well defined for arbitrary dimensions. In `ViennaGrid 1.1.0` an inner product of the form

$$(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{N-1} x_i y_i \tag{7.1}$$

is available with the function `inner_prod()`. The following code calculates and prints the inner product of the vectors $(1,0,0)^{\mathrm{T}}$ and $(0,1,0)^{\mathrm{T}}$:

```
PointType p0(1, 0, 0);
PointType p1(0, 1, 0);
std::cout << viennagrid::inner_prod(p1, p2) << std::endl;  //0
```

If the two vectors are given in different coordinate systems, the result vector will have the same coordinate system as the first argument.

7.1.3 Vector Norms

Currently, p-norms of the form

$$\|x\|_p = \sqrt[p]{\sum_{i=0}^{N-1} x_i^p} \tag{7.2}$$

are implemented in the N-dimensional Euclidian space for $p = 1$, $p = 2$ and $p = \infty$. The three norms for the vector $(1, 2, 3)^{\mathrm{T}}$ are computed and printed using the lines

```
PointType p(1, 2, 3);
std::cout << viennagrid::norm_1(p)   << std::endl; //6
std::cout << viennagrid::norm_2(p)   << std::endl; //3.74
std::cout << viennagrid::norm_inf(p) << std::endl; //3
```

which are equivalent to

```
PointType p(1, 2, 3);
std::cout << viennagrid::norm(p, viennagrid::one_tag()) << std::endl;
std::cout << viennagrid::norm(p, viennagrid::two_tag()) << std::endl;
std::cout << viennagrid::norm(p, viennagrid::inf_tag()) << std::endl;
```

7.1.4 Volume of a Spanned Simplex

It is often handy to compute the n-dimensional volume of a n-simplex embedded in a possibly higher-dimensional geometric space by providing the locations of the vertices only. This is provided by `spanned_volume()`, which is, however, currently limited to $n \in \{1, 2, 3\}$. As an example, the two-dimensional volume a triangle with corners at $(1, 0, 0)$, $(2, 1, 1)$ and $(1, 1, 2)$ is computed and printed by

```
PointType p0(1, 0, 0);
PointType p1(2, 1, 1);
PointType p2(1, 1, 2);
std::cout << viennagrid::spanned_volume(p0, p1, p2) << std::endl;
```

7.2 Element-Based

In this section, algorithms defined for geometric objects with additional structure are discussed. Additional algorithms are likely to be introduced in future releases.

7.2.1 Centroid

The centroid of an element object `element` in Cartesian coordinates is obtained as

```
PointType p = viennagrid::centroid(cell_n);
```

and works for arbitrary topological and geometrical dimensions.

7.2.2 Circumcenter

The circumcenter of a simplex `element` is obtained in Cartesian coordinates as

```
PointType p = viennagrid::circumcenter(element);
```

The computation is restricted to simplices of topologic dimension $n \leq 3$. For reasons of uniformity, also hypercubes can be passed, for which the circumcenter of an embedded simplex is computed. This leads to valid results and makes sense only for certain regular

hypercubes. Thus, the user has to ensure that the hypercube actually has a circum-sphere. This is e.g. the case for structured tensor-grids.

There is no warning or error issued if a hypercube passed to `circumcenter()` does not have a circumcenter.

7.2.3 Surface

The surface of an element `element` is defined as the sum of the volumes of its facet elements. Therefore, in order to make the calling code

```
NumericType surf = viennagrid::surface(element);
```

Currently, `surface()` is restricted to element with topologic dimension $n \leq 4$.

7.2.4 Volume

The n-dimensional volume of an element `element` is returned by

```
NumericType vol = viennagrid::volume(element);
```

and currently restricted to elements with topologic dimension $n \leq 3$. No restrictions with respect to the storage of boundary elements apply.

7.3 Domain/Segment-Based

Algorithms acting on a collection of cells are now considered. These collections are given in `ViennaGrid` either as the whole domain, or as segments.

7.3.1 Boundary

Whether or not an element object `element` is located on the boundary depends on the collection of elements considered. For example, consider the edge/facet $[1, 4]$ in the triangular sample mesh in Fig. 4.1, which we will refer to as `f`. It is in the interior of the whole `domain`, while it is also at the boundary of the two segments `seg0` and `seg1`. A sample code snippet reflecting this is given by

```
std::cout << viennagrid::is_boundary(domain, f) << std::endl; //false;
std::cout << viennagrid::is_boundary(seg0, f)   << std::endl; //true;
std::cout << viennagrid::is_boundary(seg1, f)   << std::endl; //true;
```

Note that `is_boundary()` induces some additional setup costs at the first time the function is called. However, subsequent calls are accelerated and will usually compensate for the setup costs.

7.3.2 Interface

Similar to the detection of boundary facets, elemets on the interface between two segments are frequently of particular interest. An element `element` can be checked for being on the interface of two segments `seg0` and `seg1` using

```
std::cout << viennagrid::is_interface(seg0, seg1, element) << std::endl;
```

Note that `is_interface()` induces some setup costs the first time it is called for a pair of segments.

7.3.3 Refinement

`ViennaGrid 1.1.0` allows a uniform and a local refinement of simplicial domains. The refinement of hypercuboidal domains is scheduled for future releases. It has to be noted that the resulting refined mesh is written to a new domain, thus there are no multigrid/-multilevel capabilities provided yet.

To refine a `domain` uniformly, the line

```
DomainType refined_domain;
viennagrid::refine_uniformly(domain, refined_domain);
```

is sufficient.

Refinement is also supported for domains with segmentations. Segment information is preserved upon refinement.

```
DomainType refined_domain;
SegmentationType refined_segmentation;
viennagrid::refine_uniformly(domain, segmentation, refined_domain,
    refined_segmentation);
```

The local refinement of a mesh requires that the respective cells or edges for refinement are tagged. This refinement information is applied to the domain using accessors or fields, cf. Chapter 6. First of all a tag container and an accessor has to be created:

```
std::vector<bool> cell_refinement_container;
viennagrid::result_of::accessor<std::vector<bool>, CellType>::type
    cell_refinement_accessor(cell_refinement_container);
```

To tag a cell `c` for refinement, the line

```
cell_refinement_accessor(c) = true;
```

is sufficient. In a similar way one proceeds for other cells or edges in the domain. The refinement process is then triggered by

```
DomainType refined_domain;
viennagrid::cell_refine(domain, refined_domain, cell_refinement_accessor);
```

or, for segmentaions

```
DomainType refined_domain;
SegmentationType refined_segmentation;
viennagrid::cell_refine(domain, segmentation, refined_domain,
    refined_segmentation, cell_refinement_accessor);
```

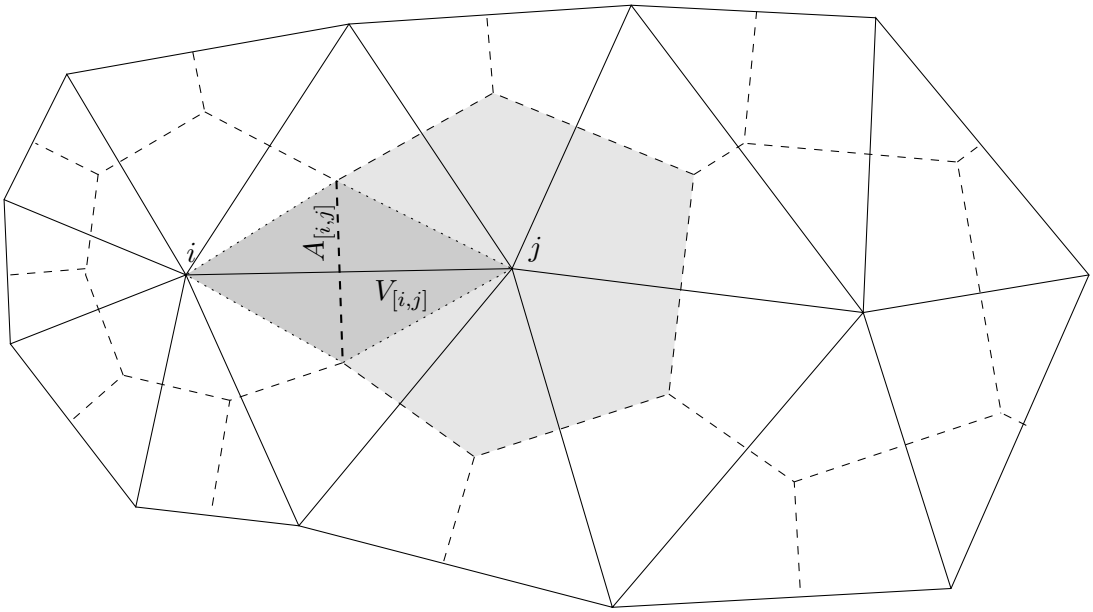


Figure 7.1: Schematic of a Delaunay mesh with its dual Voronoi diagram, where the box containing vertex j is highlighted. The function `voronoi()` computes and stores the Voronoi volume $V_{[i,j]}$ and the interface area $A_{[i,j]}$ associated with each edge $[i, j]$. The total box volume associated with each vertex is also stored on the vertex.

7.3.4 Surface

The surface of a domain or segment `domseg` is given by the sum of the volumes of the boundary facets and returned by the convenience overload

```
NumericType surf = viennagrid::surface(domseg);
```

Note that facets interfacing cells outside the segment are also considered as boundary facets of the segment.

7.3.5 Volume

The volume of a domain or segment `domseg` is returned by

```
NumericType vol = viennagrid::volume(domseg);
```

and currently restricted to maximum topological dimension $n \leq 3$.

7.3.6 Voronoi Information

A Voronoi diagram of a Delaunay tessellation (or triangulation) is a decomposition of the domain into certain boxes containing one vertex each. The boxes have the property that all points inside the box are closer to the vertex within the box than to any other vertex in the domain. By simple geometric arguments one finds that the corners of Voronoi boxes are given by the circumcenters of the triangles.

The function `apply_voronoi()` computes the volumes and interfaces associated with a Voronoi diagram. The following values are stored on the domain (cf. Fig. 7.1)

- The volume $V_{[i,j]}$ of the polyhedron centered around the edge $[i,j]$ with edges given by the connections of the vertices i and j with the circumcenters of the coboundary cells of the edge is stored on the edge $[i,j]$.
- The interface area $A_{[i,j]}$ of the boxes for the vertices i and j on the edge $[i,j]$.
- The box volume V_i of the box containing vertex i for each vertex i.

Using the lines

```
using namespace viennagrid;
typedef result_of::voronoi_cell_contribution<ConstCellHandleType>::type
   ContributionType;

std::vector<double> interface_areas;
std::vector<ContributionType> interface_contributions;

result_of::accessor< std::vector<double>, EdgeType >::type
   interface_areas_accessor(interface_areas);
result_of::accessor< std::vector<ContributionType>, EdgeType >::type
   interfaces_contribution_accessor(interface_contributions);

std::vector<double> vertex_box_volumes;
std::vector<ContributionType> vertex_box_volume_contributions;

result_of::accessor< std::vector<double>, VertexType >::type
   vertex_box_volumes_accessor(vertex_box_volumes);
result_of::accessor< std::vector<ContributionType>, VertexType >::type
   vertex_box_volume_contributions_accessor(
   vertex_box_volume_contributions);

std::vector<double> edge_box_volumes;
std::vector<ContributionType> edge_box_volume_contributions;

result_of::accessor< std::vector<double>, EdgeType >::type
   edge_box_volumes_accessor(edge_box_volumes);
result_of::accessor< std::vector<ContributionType>, EdgeType >::type
   edge_box_volume_contributions_accessor(edge_box_volume_contributions);

apply_voronoi<CellType>(
        domain,
        interface_areas_accessor, interfaces_contribution_accessor,
        vertex_box_volumes_accessor,
           vertex_box_volume_contributions_accessor,
        edge_box_volumes_accessor, edge_box_volume_contributions_accessor
           )
);
```

The voronoi values are stored in the corresponding accessors/containers.

Chapter 8

Input/Output

This description has not been updated for ViennaGrid 1.1.0 yet!

This chapter deals with the typical input and output operations: Reading a mesh from a file, and writing a mesh to a file. In order not to give birth to another mesh file format, `ViennaGrid` does not bring its own file format. Instead, the library mainly relies on the XML-based VTK [16] file format [17].

A tutorial code can be found in `examples/tutorial/io.cpp`.

Let us know about your favorite file format(s)! Send an email to our mailinglist: `viennagrid-support@lists.sourceforge.net`. It increases the chances of having a suitable reader and/or writer included in the next `ViennaGrid` release.

8.1 Readers

Due to the high number of vertices and cells in typical meshes, a manual setup of a domain in code is soon inefficient. Therefore, the geometric locations as well as topological connections are typically stored in mesh files.

Currently, `ViennaGrid` supports only two file formats natively. However, readers for other file formats can be easily added by the user when following the explanations for domain setup in Chapter 4. A different approach is to convert other file formats to one of the formats detailed in the following.

8.1.1 Netgen

The `.mesh`-files provided with `Netgen` [18] can be imported directly. These files are obtained from `Netgen` from the `File->Export Mesh...` menu item. Note that only triangular and tetrahedral meshes are supported.

To read a mesh from a `.mesh` file with name `filename` to a `domain`, the lines

```
viennagrid::io::netgen_reader my_netgen_reader;
my_netgen_reader(domain, segmentation, filename);
```

should be used. Note that the reader might throw an `std::exception` if the file cannot be opened or if there is a parser error.

The fileformat is simplistic: The first number refers to the number of vertices in the domain, then the coordinates of the vertices follow. After that, the number of cells in the domain is specified. Then, each cell is specified by the index of the segment and the indices of its vertices, each using index base 1. For example, the `.mesh`-file for the sample domain in Fig. 4.1 is:

```
6
0 0
1 0
2 0
2 1
1 1
0 1
4
1 1 2 6
1 2 5 6
2 2 3 5
2 3 4 5
```

8.1.2 VTK

The VTK file format is extensively documented [17] and allows to store mesh quantities as well. The simplest way of reading a VTK file `my_mesh.vtu` is similar to the `Netgen` reader:

```
viennagrid::io::vtk_reader<DomainType, SegmentationType>  my_vtk_reader;
my_vtk_reader(domain, segmentation, "my_mesh.vtu");
```

Note that the domain type is required as template argument for the reader class, the segmentation type is optional. By default `result_of::segmentation<DomainType>::type` is used.

`ViennaGrid` supports single-segmented `.vtu` files consisting of one `<piece>`. and always reads a single-segmented mesh to its first segment. If a segment does not already exist on the domain, one is created.

For multi-segment meshes, the Paraview [19] data file format `.pvd` can be used, which is a XML wrapper holding information about the individual segments in `.vtu` files only. Vertices that show up in multiple segments are automatically fused. Example meshes can be found in `examples/data/`.

The VTK format allows to store scalar-valued and vector-valued data sets, which are identified by their names, on vertices and cells. These data sets are directly transferred to the `ViennaGrid` domain using accessor as described in Chapter 6. By default, data is stored using the data name string as key of type `std::string`. Scalar-valued data is stored as `double`, while vector-valued data is stored as `std::vector<double>`.

Function name	Data description
`add_scalar_data_on_vertices`	scalar-valued, vertex-based
`add_vector_data_on_vertices`	vector-valued, vertex-based
`add_scalar_data_on_cells`	scalar-valued, cell-based
`add_vector_data_on_cells`	vector-valued, cell-based

Table 8.1: Free functions in namespace `viennagrid::io` for customizing reader and writer objects. The three parameters to each of these functions is the reader object, the accessor/-field and the VTK data name.

There are two ways to obtain the data stored on vertices and cells: query the data after import or register an accessor/field before import. Registering an accessor/field with an import is simply done by using the function `add_*_data_on_*`, see Tab. 8.1

```
std::vector<double> scalar_values;
viennagrid::result_of::field< std::vector<double>, VertexType >::type
   scalar_values_field;

std::vector< std::vector<double> > vector_values;
viennagrid::result_of::field< std::vector< std::vector<double> >, CellType
    >::type vector_values_field;

viennagrid::io::vtk_reader<DomainType>      my_vtk_reader;
viennagrid::io::add_scalar_data_on_vertices(my_vtk_reader,
   scalar_values_field, "potential");
viennagrid::io::add_vector_data_on_cells(my_vtk_reader,
   vector_values_field, "potential_vector_field");

my_vtk_reader(domain, segmentation, "my_mesh.vtu");
```

A list of all names identifying data read from the file can be obtained the functions

Function name	Data description
`get_scalar_data_on_vertices`	scalar-valued, vertex-based
`get_vector_data_on_vertices`	vector-valued, vertex-based
`get_scalar_data_on_cells`	scalar-valued, cell-based
`get_vector_data_on_cells`	vector-valued, cell-based

The list of all scalar vertex data sets read is then printed together with the respective segment index as

```
using namespace viennagrid::io;
for (size_t i=0; i<get_scalar_data_on_vertices(reader).size(); ++i)
  std::cout << "Segment " << get_scalar_data_on_vertices(reader)[i].first
            << ": "
            << get_scalar_data_on_vertices(reader)[i].second << std::endl;
```

After the import process a scalar/vector data field can be obtained with the member functions described in table Tab. 8.2. In the next example a scalar field is obtained for the data values with VTK name `potential` and for the segment with ID 42.

Function name	Data description
`vertex_scalar_field`	scalar-valued, vertex-based
`vertex_vector_field`	vector-valued, vertex-based
`cell_scalar_field`	scalar-valued, cell-based
`cell_vector_field`	vector-valued, cell-based

Table 8.2: Free functions in namespace `viennagrid::io` for customizing reader and writer objects. The three parameters to each of these functions is the reader object, the accessor/-field and the VTK data name.

```
viennagrid::result_of::field<std::vector<double>, VertexType>
    scalar_field =
  my_vtk_reader.cell_scalar_field( "potential", 42 );
```

8.2 Writers

Since `ViennaGrid` does not provide any visualization capabilities, the recommended procedure for visualization is to write to one of the file formats discussed below and use one of the free visualization suites for that purpose.

8.2.1 OpenDX

`OpenDX` [20] is an open source visualization software package based on IBM's Visualization Data Explorer. The writer supports either one vertex-based or one cell-based scalar quantity to be written to an `OpenDX` file. OpenDX writer does not support segmentations.

The simplest way to write a `domain` of type `DomainType` to a file `"my_mesh.out"` is

```
viennagrid::io::opendx_writer<DomainType> my_dx_writer;
my_dx_writer(domain, "my_mesh.out");
```

To write quantities stored on the domain, the free functions from Tab. 8.1 are used. For example, to visualize a scalar vertex-quantity of type `double` stored in an accessor (cf. Chapter 6), the previous snippet is modified to

```
using viennagrid::io;
opendx_writer<DomainType> my_dx_writer;
add_scalar_data_on_vertices(my_dx_writer,    // writer
                            value_accessor,  // some accessor
                            "some_name");    // ignored
my_dx_writer(domain, "my_mesh.out");
```

Note that the data name provided as third argument is ignored for the `OpenDX` writer.

`ViennaGrid` can write only one scalar quantity to an `OpenDX` file!

8.2.2 VTK

A number of free visualization tools such as Paraview [19] are available for the visualization of VTK files. The simplest way of writing to a VTK file is

```
viennagrid::io::vtk_writer<DomainType, SegmentationType> my_vtk_writer;
my_vtk_writer(domain, segmentation, "outfile");
```

Each segment is written to a separate file, leading to `"outfile_0.vtu"`, `"outfile_1.vtu"`, etc. In addition, a Paraview data file `"outfile_main.pvd"` is written, which links all the segments and should thus be used for visualization.

If no segmentation is given, the file `"outfile.vtu"` is written using the following code example

```
viennagrid::io::vtk_writer<DomainType> my_vtk_writer;
my_vtk_writer(domain, "outfile");
```

To write quantities stored on the domain to the VTK file(s), the free functions from Tab. 8.1 are used. For example, to visualize a vector-valued cell-quantity of type `std::vector<double>` stored in an accessor (cf. Chapter 6), the previous snippet is modified to

```
using viennagrid::io;
vtk_writer<DomainType> my_vtk_writer;
add_vector_data_on_vertices(my_vtk_writer,        // writer
                            value_accessor,       // some accessor
                            "vtk_name");          // VTK name
 my_vtk_writer(domain, "outfile");
```

In this way, the quantity is written to all segments and values at the interface coincide.

If discontinuities at the interfaces should be allowed, vertex or cell data may also be written per segment.

As closing example, a quantity `"jump"` with type `double` is stored in an accessor. For a vertex `v` at the interface of segments with indices 0 and 1, the values `1.0` for the segment with index 1 and `2.0` for the segment with index 2 are stored:

```
std::vector<double> segment_0_values;
std::vector<double> segment_1_values;

viennagrid::result_of::field<std::vector<double>, VertexType>::type
    segment_0_values_accessor(segment_0_values);
viennagrid::result_of::field<std::vector<double>, VertexType>::type
    segment_1_values_accessor(segment_1_values);

segment_0_values_accessor(v) = 1.0;
segment_1_values_accessor(v) = 2.0;
```

The quantity is added to the VTK writer as

```
add_scalar_data_on_vertices(my_vtk_writer, segment_0, segment_0_values, "
    segment_data");
add_scalar_data_on_vertices(my_vtk_writer, segment_1, segment_1_values, "
    segment_data");
```

A tutorial code using a VTK writer for discontinuous data at segment boundaries can be found in `examples/tutorial/multi-segment.cpp`.

Chapter 9

Library Internals

Details about the internals of ViennaGrid will be given in the following. They should aid developers to extend the library with additional features and to understand the internal data structures used. Nevertheless, the information provided might be of interest for new users of ViennaGrid as well.

9.1 Recursive Inheritance

ViennaGrid extensively relies on recursive inheritance to build the individual element types. The element are of type element_t<ElementTag, ConfigType>, where ElementTag is a tag identifying the topological shape of the element and ConfigType is the configuration class. The class element_t itself is almost empty, but inherits the information about its boundary elements from a boundary_element_layer. In addition, the class inherits from a class with the sole purpose of providing an ID mechanism

```
template <typename ElementTag,
          stypename ConfigType>
class element_t :
   public viennagrid::storage::id_handler<...>,
   public boundary_element_layer<ElementTag,
       BoundaryElementContainerTypelist>,
{ ... }
```

The second template parameter of boundary_element_layer is a typelist where all boundary element types are stored. The boundary_element_layer inherits from a boundary_element_layer with the last element type of that typelist.

```
template<typename element_tag, typename bnd_cell_container_type_, typename
   orientation_container_type_, typename tail>
class boundary_element_layer<element_tag, meta::typelist_t< meta::
   static_pair<bnd_cell_container_type_, orientation_container_type_>, tail
    > > :
    public boundary_element_layer<element_tag, tail>
```

where the additional technical template arguments customizing the behavior of each boundary layer are omitted. The recursion is terminated at vertex level by a partial template specialization of the class.

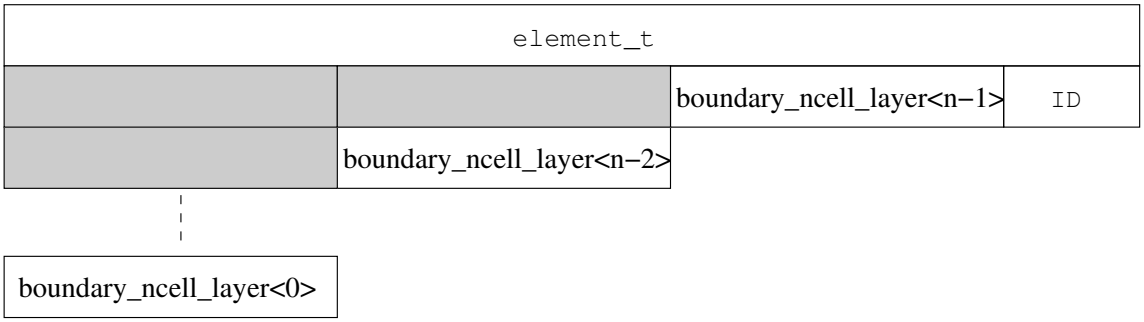


Figure 9.1: Illustration of recursive inheritance for the element class `element_t`. The `boundary_element_layer` class is abbreviated as `bnd_layer`. The widths of the boxes refer to the sizes of a single class object.

An illustration of the resulting class layout is given in Fig. 9.1. Since each layer is configured by respective metafunctions that return the specified user settings, the class size of each layer varies depending on the storage of boundary cells and on the use of local reference orientations.

A problem of recursive inheritance is name hiding. For example, a member function `fun()` defined for one layer will be repeated in every layer and thus become inaccessible from the outside. The resolve this issue, member function overloading of the form

```
void fun(boundary_element_tag) { }
```

for boundary element with tag `boundary_element_tag` is used. In this way, every layer is accessible from the outside, which is used for the free functions such as `elements<>()`.

Domains and segments are set up in essentially the same way with a few internal storage facilities adjusted. Instead of an ID handler, the segment inherits from a class providing a reference to the underlying domain.

9.2 Element Storage in Domain and Segment

For storing the elements inside a domain, the natural approach is to use a `std::vector<>` for that purpose. However, a drawback of this datastructure is that the number of elements should be known in advance in order to avoid costly reallocations. This is especially a problem for mesh file formats which do not contain the total number of elements in an explicit way. `ViennaGrid` uses a `std::deque<>` (double ended queue) as container for vertices and cells, because it does not suffer form costly reallocations if the number of elements is a-priori unknown.

For non-vertices and non-cells, unique representations of the respective elements are required. For example, the edge connecting vertices 1 and 2 must not lead to an edge $[1, 2]$ and an edge $[2, 1]$ in the domain. While such a distinction is simple for vertices, this is harder to achieve for e.g. quadrilateral facets. In `ViennaGrid`, the global vertex IDs of each element are used as a tuple for the identification of the respective element. Thus, the internal storage inside the domain for each non-vertices and non-cells is given by

```
std::map<TupleType, ElementType>
```

where `TupleType` refers to the tuple of sorted global vertex IDs, and `ElementType` is the type of the element.

For segments, only handles to the global element objects in the domain are stored. Since uniqueness of elements is required in segments as well, an internal storage scheme of type `std::set<ElementHandleType>` is chosen for non-cells, where `ElementType` denotes the type of the elements.

Finally, it should be noted that future versions of `ViennaGrid` may provide additional flexibility in customizing the internal storage scheme for domain and segments. In particular, users may be interested in replacing the `std::map<TupleType, ElementType>` used for non-vertices and non-cells with a `std::vector<>` after the setup phase for reasons of memory consumption, faster (random) access and/or better caching possibilities.

Chapter 10

Design Decisions

In this chapter the various aspects that have lead to ViennaGrid in the present form are documented. The discussion focuses on key design decisions mostly affecting usability and convenience for the library user, rather than discussing programming details invisible to the library user. Since the design decisions also reflect the history of ViennaGrid and the individual preferences of the authors to a certain degree, a more vital language is chosen in the following.

10.1 Iterators

Consider the iteration over all vertices of a domain. Clearly, the choice

```
for (VertexIterator vit = domain.begin(); vit != domain.end(); ++vit) {}
```

is not sufficiently flexible in order to allow for an iteration over edges. Nevertheless, the STL-like setup and retrieval of iterators is appealing.

A run-time dispatched iterator retrieval in the spirit of

```
for (VertexIterator vit  = domain.begin(ElementTag());
                    vit != domain.end(ElementTag());
                  ++vit) {}
```

will sacrifice efficiency especially for loops with a small number of iterations, which is not an option for high-performance environments. Therefore, iterators should be accessed using a compile time dispatch.

Since hard-coding function names is not an option for an parametrized traversal, the next choice is to add a template parameter to the begin and end member functions:

```
for (VertexIterator vit  = domain.begin<ElementTag>();
                    vit != domain.end<ElementTag>();
                  ++vit) {}
```

This concept would be sufficiently flexible to allow for an iteration over edges, facets, etc. in a unified way. In fact, this approach is also chosen by DUNE [2] and was also used in an early prototype of ViennaGrid. However, there is one pecularity with this approach when it comes to template member functions: According to the C++ standard, the syntax needs to be supplemented with an additional **template** keyword, resulting in

```
for (VertexIterator vit  = domain.template begin<ElementTag>();
                    vit != domain.template end<ElementTag>();
                  ++vit) {}
```

Apart from the fact that the `template` keyword for member functions is probably unknown to a wide audience, weird compiler messages are issued if the keyword is forgotten. Since a high usability is one of the design goals of ViennaGrid, we kept searching for better ways with respect to our measures.

Having high-performance environments in mind, one must not forget about the advantage of index-based for loops such as

```
for (std::size_t i=0; i<3; ++i) { ... }
```

for the iteration over e.g. the vertices of a triangle. The advantage here stems from the fact that the compiler is able to unroll the loop, which is much harder with iterators. Consequently, we looked out for a unified way of both iterator-based traversal as well as an index-based traversal.

In order to stick with a simple iterator-based loop of the form

```
for (VertexIterator vit  = something.begin();
                    vit != something.end();
                  ++vit) {}
```

where `something` is some proxy-object for the vertices in the domain, one finally ends up with the current `viennagrid::ncells<>` approach. Writing the loop in the form

```
for (VertexIterator vit  = elements<ElementTag>(domain).begin();
                    vit != elements<ElementTag>(domain).end();
                  ++vit) {}
```

readily expresses the intention of iterating over 0-cells and does not suffer from the problems related to the `template` keyword. Thanks to the rich overloading rule-set for free functions, an extension to `elements<ElementTag>(segment)` or `elements<ElementTag>(cell)` is immediate. As presented in Chapter 5, the `elements<>()` approach also allows for an index-based iteration in most cases.

In retrospective, the ranges returned by `elements<>()` would have come up with coboundary iterators (which were added later), since one then has to pass the enclosing cell complex anyway.

10.2 Default Behavior

A delicate question for a highly configurable library such as ViennaGrid is the question of the default behavior. We decided to provide the full functionality by default, even if the price to pay is a possibly slow execution at first instance.

Our decision is based on mostly psychological reasoning as follows: If ViennaGrid were tuned for low memory consumption and high speed by default, then a substantial set of functionality would be unavailable by default and causing compilation errors for users that just want to try a particular feature of ViennaGrid. It is unlikely that these users will continue to use ViennaGrid if they are not able to compile a simple application without

digging through the manual and searching for the correct configuration. On the contrary, if the desired feature works essentially out of the box and is fast enough, users will not even have to care about the further configuration of `ViennaGrid`. However, if additional speed is required, there are plenty of configuration options available, each potentially leading to higher speed or lower memory footprint and thus resulting in a feeling of success.

The bottomline of all this is that we consider it more enjoyable to tune a slower default configuration for maximum speed than to fight with compiler error messages and being forced to work through the manual. We hope that our decision indeed reflects the preferences of our users.

10.3 Segments

Operating on a subset Ω_i of a domain Ω is a common demand. This could be well achieved by what is known as a *view*, i.e. a selection of elements from a domain. However, a view typically possesses a considerably shorter lifetime compared to the domain, thus it is hard to store any data for the view itself. For this reason, segments in `ViennaGrid` are part of the domain, thus having a comparable lifetime in typical situations. In particular, meta information can be stored on a segment during the domain setup stage already.

Appendix A

Reference Orientations

The order of the vertices of a n-cell implicitly determines the boundary k-cells, thus it is crucial to provide the vertices in the correct order. While one- and two-dimensional objects provide little space for variations of reference orientations, the situation changes in higher dimensions. The reference orientations of the boundary k-cells provides ample of variations, while only a few choices satisfy requirements such as consistent cell normals.

The reference orientations in `ViennaGrid` are chosen such that the tuples of vertex IDs for all boundary cells are in ascending order. This is accomplished in a generic way for a unit-n-cell from the simplex and hypercube families in the n-dimensional space as follows:

1. Start with a 1-cell at the points $(1, 0, \ldots)$ and $(0, 1, 0, \ldots)$, enumerate the points and set $k = 2$.
2. Prolongate the $k-1$-cell to a k-cell in the k-dimensional subspace induced by the first k unit vectors.
3. Enumerate the new vertices.
4. If $k = n$, stop. Otherwise, go back to 2.

The families of simplices and hypercubes differ in the way the the prolongation is carried out. Details are given in the following subsections.

Boundary k-cells are ordered with respect to the less-than operator acting on the tuple of vertices in ascending order. The first entry has precedence over the second entry, etc.

A.1 Simplices

The prolongation from a $k-1$-simplex to a k-simplex is straightforward, since only one vertex is added. Resulting reference orientations for a triangle and a tetrahedron are given in Fig. A.1.

Boundary k-cell orientations are chosen such that the tuple of vertices is sorted in increasing order. Note that this does not lead to a consistent orientation of n-cell normals. In particular, the boundary k-cells of a triangle and a tetrahedron are thus ordered as follows:

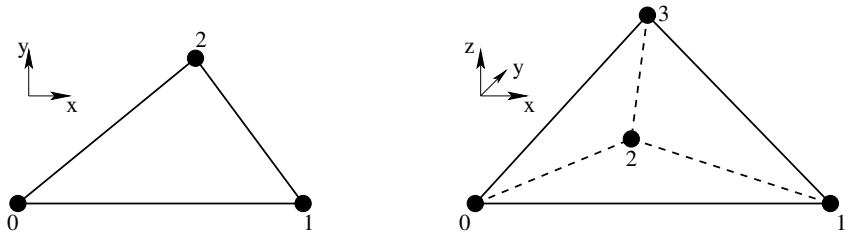


Figure A.1: Reference orientations of a triangle (left) and a tetrahedron (right).

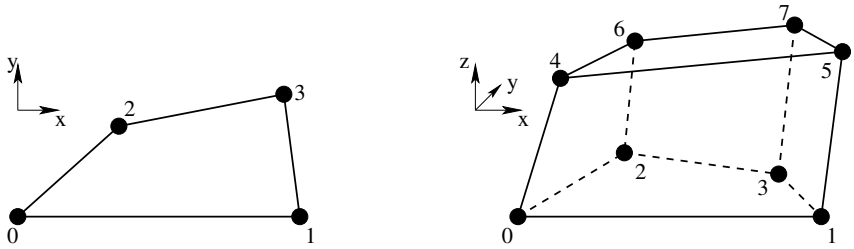


Figure A.2: Reference orientations of a quadrilateral (left) and a hexahedron (right).

	Triangle	Tetrahedron
1-cells	$[0,1]$, $[0,2]$, $[1,2]$	$[0,1]$, $[0,2]$, $[0,3]$, $[1,2]$, $[1,3]$, $[2,3]$
2-cells	$[0,1,2]$	$[0,1,2]$, $[0,1,3]$, $[0,2,3]$, $[1,2,3]$

A.2 Hypercube

For the prolongation from a unit-$k-1$-hypercube to a unit-k-hypercube, the standard tensor construction is used. This results in a second $k-1$-hypercube shifted along the k-th unit vector, with the new vertices enumerated in the same orientation as the initial $k-1$-hypercube. This procedure can also be seen in Fig. A.2, where the two 1-cells used for the prolongation to the hexahedron are $[0,1,2,3]$ and $[4,5,6,7]$.

For reference, the edges and faces of a quadrilateral and a hexahedron are ordered as follows:

	Quadrilateral	Hexahedron
1-cells	$[0,1]$, $[0,2]$, $[1,3]$, $[2,3]$	$[0,1]$, $[0,2]$, $[0,4]$, $[1,3]$, $[1,5]$, $[2,3]$, $[2,6]$, $[3,7]$, $[4,5]$, $[4,6]$, $[5,7]$, $[6,7]$
2-cells	$[0,1,2,3]$	$[0,1,2,3]$, $[0,1,4,5]$, $[0,2,4,6]$, $[1,3,5,7]$, $[2,3,6,7]$, $[4,5,6,7]$

Mind that the the reference orientations of a `ViennaGrid` quadrilateral and a `ViennaGrid` hexahedron coincide with those of the VTK types `VTK_PIXEL` and `VTK_VOXEL`, but differ from the orientations of the VTK types `VTK_QUAD` and `VTK_HEXAHEDRON`.

Appendix B

Versioning

Each release of ViennaGrid carries a three-fold version number, given by

```
ViennaGrid X.Y.Z.
```

For users migrating from an older release of ViennaGrid to a newer one, the following guidelines apply:

- X is the *major version number*, starting with 1. A change in the major version number is not necessarily API-compatible with any versions of ViennaGrid carrying a different major version number. In particular, end users of ViennaGrid have to expect considerable code changes when changing between different major versions of ViennaGrid.
- Y denotes the *minor version number*, restarting with zero whenever the major version number changes. The minor version number is incremented whenever significant functionality is added to ViennaGrid. The API of an older release of ViennaGrid with smaller minor version number (but same major version number) is *essentially* compatible to the new version, hence end users of ViennaGrid usually do not have to alter their application code. There may be small adjustments in the public API, which will be extensively documented in the change logs and require at most very little changes in the application code.
- Z is the *revision number*. If either the major or the minor version number changes, the revision number is reset to zero. The public APIs of releases of ViennaGrid, which only differ in their revision number, are compatible. Typically, the revision number is increased whenever bugfixes are applied, performance and/or memory footprint is improved, or some extra, not overly significant functionality is added.

Always try to use the latest version of ViennaGrid before submitting bug reports!

Appendix C

Change Logs

Version 1.1.0

- New storage layer, re-wrote most of the internals
- Using tags instead of topologic dimension to identify elements
- Added support for dynamic elements: polygon and PLCs
- Added support for neighbour iteration
- Addes support for multiple segmentations
- New algorithms: intersection, seed point segmenting

Version 1.0.0

First release

Appendix D

License

Bibliography

[1] “CGAL - Computational Geometry Algorithms Library.” [Online]. Available: http://www.cgal.org/

[2] “DUNE - Distributed and Unified Numerics Environment.” [Online]. Available: http://www.dune-project.org/

[3] “GrAL - Grid Algorithms Library.” [Online]. Available: http://gral.berlios.de/

[4] “libmesh.” [Online]. Available: http://libmesh.sourceforge.net/

[5] “OpenMesh.” [Online]. Available: http://www.openmesh.org/

[6] “trimesh2.” [Online]. Available: http://gfx.cs.princeton.edu/proj/trimesh2/

[7] “VCGlib.” [Online]. Available: http://vcg.sourceforge.net/

[8] “ViennaData.” [Online]. Available: http://viennadata.sourceforge.net/

[9] “CMake.” [Online]. Available: http://www.cmake.org/

[10] “Xcode Developer Tools.” [Online]. Available: http://developer.apple.com/technologies/tools/xcode.html

[11] “Fink.” [Online]. Available: http://www.finkproject.org/

[12] “DarwinPorts.” [Online]. Available: http://darwinports.com/

[13] “MacPorts.” [Online]. Available: http://www.macports.org/

[14] “Boost C++ Libraries.” [Online]. Available: http://www.boost.org/

[15] “OpenMP.” [Online]. Available: http://openmp.org

[16] “VTK - Visualization Toolkit.” [Online]. Available: http://www.vtk.org/

[17] “VTK File Formats.” [Online]. Available: http://www.vtk.org/VTK/img/file-formats.pdf

[18] “Netgen Mesh Generator.” [Online]. Available: http://sourceforge.net/projects/netgen-mesher/

[19] “Paraview - Open Source Scientific Visualization.” [Online]. Available: http://www.paraview.org/

[20] “OpenDX.” [Online]. Available: http://www.opendx.org/