WCCM V
Fifth World Congress on
Computational Mechanics
July 7–12, 2002, Vienna, Austria
Eds.: H.A. Mang, F.G. Rammerstorfer,
J. Eberhardsteiner

Generic programming for mesh algorithms: Implementing universally usable geometric components

Guntram Berti

C&C Research Laboratories
NEC Europe Ltd., Rathausallee 10, 53757 St. Augustin, Germany
e-mail: berti@ccrl-nece.de

Key words: Generic programming, reusable software components, geometric algorithms, mesh data structure abstractions

Abstract
Geometric functionality is crucial for a variety of application domains, including computational mechanics. Typically, geometric tasks are embedded into a larger problem frame. Due to the diversity of tasks, geometric tools must often be combined to achieve the desired solution. As implementing geometric algorithms is difficult and time-consuming, reusing them is highly desirable. Unfortunately, traditional implementations are intimately tied to the underlying representations of the geometric data, and hence are not directly usable in a different context. Conventional approaches to implementing geometric tools are thus limited to copying the data via an API (or to a file), and calling an external routine (or application) application implementing the desired functionality or even implementing an ad-hoc solution, and have obvious drawbacks in terms of efficiency, composability, scalability or quality. Here, we present a radically different approach, concentrating on the case of algorithms working on *cellular structures*, for example meshes (tesselations, grids) of surfaces and solids. Exploiting the common underlying mathematical structure we define an abstract interface capturing this mathematical notion. Algorithms are implemented *generically* on top of this interface, thus making the implementations independent of any concrete representation issues. Special emphasis is placed on a separation of combinatorial and geometric aspects and on a general framework for associating arbitrary data to mesh entities of any dimension. A wide variety of reusable geometric tools is implemented in the open source C++ library GRAL. They can be combined and nested in very flexible ways to solve geometric task arising in computational mechanics, independently of the underlying data structures. We conclude by discussing some practical issues of the generic approach, including the quite competitive efficiency of generic components.

1 Introduction

Geometric algorithms play an important role in computational mechanics. Starting with the preparation of a geometric model, a suitable mesh (e. g. for finite element computations) has to be generated, and its quality must be assured. Specification of boundary conditions may involve complex computations like detecting mesh intersections for contact problems. For Lagrangian approaches, the mesh has to be updated and its quality continuously monitored. Besides these standard tasks, there occur a lot of special geometric problems requiring ad-hoc solutions.

Geometric software *per se* is widely available, both free and commercial. However, the traditional design of geometric (and also other scientific) software libraries severely constrains their use in a *different context*, for instance in computational mechanics. Why? Implementations of algorithms must make some assumptions on the data they are operating on. The more complex the data, the more assumptions are made, and the stronger the ties to a specific representation. Now geometric data, such as meshes, are rather complex entities, and therefore geometric libraries either define their own set of data structures (e. g. CGAL [1]), or they operate on some "standard" – often low-level array-based – representations, for example the code of graphics gems (e. g. [2]).

Now, simulation applications in general have their own geometric representations, often tuned towards computational needs, such as FEM meshes. If a geometric library such as discussed above are to be used on these representations, one has to copy the data. This leads to programming overhead, performance penalties and also to memory bottlenecks, as data sets in these simulations tend to be large. For local or incremental algorithms, e. g. point location, the overhead may be prohibitive. As a consequence, geometric algorithms are re-implemented over and over, with all the well-known drawbacks for productivity and code quality.

To overcome these problems, we propose a radically different approach for implementing geometric libraries, which is applicable to other algorithmic libraries, too. Algorithm implementations are no longer accessing low-level data layout details directly, but through a concise abstract interface. Then, algorithms can work generically on a broad range of data structures supporting the interface. In the following, we show how to design such an interface for cellular structures like finite element meshes, and how to define algorithms on top of this interface. By using the template feature of C++, we provide the compiler with sufficient information to optimize out most of the potential abstraction overhead. In this way, we achieve a complete decoupling of algorithms from data structures while maintaining a very competitive performance.

The approach owes a lot to the C++ STL [3], which provided the initial idea. Related work has been accomplished e. g. in the field of graphs [4] or image processing [5]. In the field of computational geometry, the CGAL library [1] comes closest to our approach, yet many of its algorithms are not as generic as they could be.

The outline of this paper is as follows: First, we introduce the generic programming approach for mesh data structures. Next, we show how typical geometric tasks arising in computational mechanics can be solved by using generic geometric tools. Finally, we discuss some practical aspects of the generic approach.

2 The Generic Programming Approach for Cellular Structures

A standard interface "sitting" between algorithms and data structures is subject to several constraints: First, it must satisfy the functionality requirements of a (hopefully very large) class of algorithms. As high-level descriptions of algorithms are generally referring to a common underlying mathematical structure – in our case, cellular complexes, which we will describe below – the aim to provide a faithful representation for this structure turns out to be a useful guiding principle for interface design. As data structures are in general compromises between efficiency and completeness, the interface must be fine-grained enough to cope with diverging capabilities of mesh representations. On the other hand, it should be simple enough to be easy to learn and implement. Last but not least, *efficient* generic implementations of algorithms must be possible on top of the interface.

There are two reasons in favor of a *layered* grid interface: First, the fact that many algorithms can do with a very simple interface, while some need a more sophisticated one. And second, the hierarchy of mathematical concepts beneath the most general definitions. For example, consider point location which can be implemented much more efficiently on a Cartesian than on a general grid. But it has to exploit special structure not accessible through the general interface. The generic approach offers a natural way for dealing with this situation: We may *specialize* the algorithm for the concrete special data structure and make use of the additional information. Thus, specialization – a key ingredient of generic programming! – reflects the ramifications of the underlying mathematical concepts. The next logical step is to develop an additional interface layer capturing the essence of Cartesian grids, and to make the specialization work on *all* representations for this abstract concept. This *partial specialization* is directly supported by C++.

In the following, we give a very short overview of a simple abstract interface for geometric meshes (cellular complexes), together with two extension layers: First, additional information on local structure using *cell archetypes* and the *switch* operator introduced in [6], and second, coarse-grained mutating primitives for grids. We begin by defining the underlying mathematical concepts.

Definition 1 (Abstract complex) *An abstract finite complex $\mathcal{C}$ of dimension d is a set of* elements e*, together with a mapping* $\dim : \mathcal{C} \mapsto \{0, \ldots, d\} \subset \mathbb{N}$*, (*$\dim(e)$ *is called the dimension of* e*), and a partial order* $<$ *(*side-of *relation) with* $e_1 < e_2 \Rightarrow \dim(e_1) < \dim(e_2)$*. Elements are named according to table 1. A* morphism *between abstract complexes $\mathcal{C}_1, \mathcal{C}_2$ is a mapping $\Phi : \mathcal{C}_1 \mapsto \mathcal{C}_2$ with $e < f \Rightarrow \Phi(e) < \Phi(f)$.*

An abstract complex is a purely combinatorial entity, also known as *poset* (which in many important cases is even a *lattice*). We need the notion of a geometric complex, too:

Definition 2 (Geometric realization of an abstract complex) *A geometric realization (or embedding) Γ of an abstract complex $\mathcal{C}$ is a mapping into a Hausdorff space $\|\mathcal{C}\|$ such that*

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

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

In the following, we will use the terms complex, grid and mesh interchangeably. An analysis of a number of mesh algorithms reveals a recurring pattern: The required functionality can be classified as *combinatorial*, relating to the abstract complex, *geometric*, and *data association*, i. e. storing data on grid elements

Table 1: Combinatorial grid entities

Element	dim	codim	Sequence Iterator
Vertex	0	d	VertexIterator
Edge	1	$d-1$	EdgeIterator
Face	2	$d-2$	FaceIterator
Facet	$d-1$	1	FacetIterator
Cell	d	0	CellIterator

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

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

(grid functions). Fig. 1 is a simple but typical example. Furthermore, there is functionality related to modifying a grid, discussed afterwards. We will not discuss the interface in technical detail, see fig. 2 for an example and [7] for detailed documentation.

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

Figure 1: A simple algorithm ...

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

Figure 2: ... and its generic implementation

2.1 The Basic Combinatorial Interface

At a very basic level, a grid is a set of sequences of its *elements*: A sequence of its vertices, of its edges, and so on. We can model this property by introducing *grid sequence iterators* which just have the standard (STL) iterator interface. The naming of elements with respect to either dimension of codimension (see table 1) allows for dimension-independent implementations of some algorithms, e. g. figure 2. A minimal representation of an element of a fixed grid is called *element handle*, which may be simply an integer. Handles are useful e. g. for representing subranges.

In order to access the incidence relationship, we need *incidence iterators* (table 2). These allow for example to access the sequence of all vertices of a cell (VertexOnCellIterator), see fig. 3. A related concept are *adjacency iterators*, which relate elements of the same dimension. We define them only for vertices and cells, lacking a "natural" definition for the intermediate dimensions.

This interface is fine-granular enough to cater for different capabilities of grid data structures: If a particular incidence relationship is not readily available, just skip the corresponding iterator. Also, it does not impose that anything at all is stored, which is important for implicitly given grids like Cartesian ones.

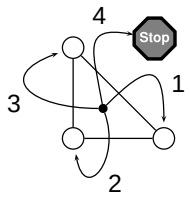

Figure 3: Action of a VertexOnCellIterator (*Incidence iterator*)

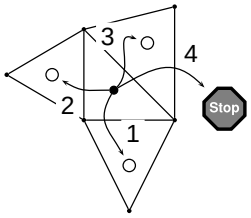

Figure 4: Action of a CellOnCellIterator (*Adjacency iterator*)

2.2 The Grid-Geometry Interface

Grid geometries represent geometric realizations (embeddings) of combinatorial grids. Thus, they map combinatorial to geometric entities of corresponding dimensions: Vertices to points, edges to arcs, etc. The grid geometry interface is open for additional properties or measures, for example lengths of edges, thus entailing better encapsulation of geometric decision: If edge lengths are computed in client code under the implicit assumption of linear segments, it would fail to profit from pre-calculated edge lengths, or would break for, say, isoparametric elements of higher order.

The separation from the combinatorial grid layer has a number of practical advantages. It allows to reuse the same combinatorial grid data structure with different embeddings (and vice versa!), for example 2D domain and 3D surface grids. We can also maintain different geometries for the same grid simultaneously, for instance use straight edges for FEM computation, and curved edges on the boundary for grid refinement, or we can employ exact arithmetic just for some geometric algorithms requiring it.

2.3 The Grid Function Interface

Almost every algorithm needs to store and access data on grid elements of any dimension. This is supported by *grid functions*. Grid functions can be *total* (dense), that is, explicitly stored on each element, or *partial* (sparse), that is, having a default value for most elements. Partial grid functions are essential for the efficient implementation of local algorithms. For both variants exist generic default implementations in GRAL, see also [8].

In these implementations, the storage of this application-dependent data is decoupled completely from the combinatorial grid data structure, such that arbitrary types of data can be stored on a given grid. This decoupling is crucial to avoid data structures needing to know about the algorithms using them – which would be even worse than the inverse coupling we overcame with the generic approach! Thus, it is important to recognize grid functions as a separate concept. Yet, one can often find this coupling in object-oriented implementations of grid data structures, where for instance state information is stored in vertex objects.

2.4 Mutating Primitives

The interface presented so far allows only read access to meshes. Sometimes, however, we need to change the grid: The simplest case occurs if we read the grid from some file, or copy it from another grid. Also, for refinement, coarsening or optimization, the mesh has to be changed.

In search of a general solution which allows efficient implementations for a large class of data structures, we found that in virtually all cases we investigated it proved sufficient to use *coarse grained* mutating primitives (in contrast to *atomic* primitives like Euler operators [9]). We can do with just three of them: Copying grids, enlarging (gluing) grids, and removing parts of a grid. These primitives maintain a *grid isomorphism* between the source and the copy, in order to allow the transfer of additional information, such as grid functions. Mutating primitives are discussed in more detail in [10] and [11].

The copy primitive can be seen as a generalized constructor, and it can be used to implement transparent file I/O or data structure conversion, necessary for using traditional libraries with an API or file coupling. To this end, an input adapter having a (minimal) grid interface and an output adapter is implemented for each file format. Reading the file is achieved by copying from the input adapter, and writing is equivalent to copying to the output adapter. An example in GRAL is the output adapter for the OFF format used by GEOMVIEW [12].

A generic copy operation is not as straightforward as it might seem: For instance, different applications often have different conventions for numbering 3D cell vertices. In order to copy from one numbering to another, we need to calculate a grid isomorphism between the two cell representations (or more precisely, their *archetypes*, see below). There is a generic algorithm in GRAL for this task.

2.5 Enhanced Combinatorial Interface

Incidence iterators provide enough combinatorial information for a surprisingly large class of algorithms. However, there is no ordering relationship between different incidence iterators; for example in 2D, vertices and edges incident to a cell can be ordered independently.

If we need such relationships, we can use the *switch* operator, which exploits the lattice structure of a grid's poset. This operator allows e. g. to traverse a connected component of a grid's boundary (see [10] for details), or to investigate a corner of a cell, see section 3.1 below. Second, the boundary of a cell (its *archetype*) can be accessed as a grid of dimension $d-1$, thus making local vertex numbering explicit. A corresponding *archetype geometry* defines local coordinates on a cell. In a typical FEM mesh, there are only few different archetypes, meaning that calculations which can be performed om archetypes (such as isomorphisms) can be reused for a large number of cells. The interface for switch and archetypes is still work in progress.

3 Reusable Geometric and Topological Components

Geometric or topological (combinatoric) problems occur over and over in computational mechanics, for instance related to boundary conditions or mesh handling. Often, these tasks require rather ad-hoc solutions, such that no general library can provide ready-made solutions. What a library *can* provide, however, is a well-designed, extensible set of *standard* tools which can be combined to solve these *non-standard* tasks. The generic approach we have introduced supplies a good framework for developing combinable tools; it forms the nucleus of a high-level language which can greatly simplify problem solution. The framework not only guarantees to a certain extent a common language of tools to ensure their syntactic composability, it also ensures that the iterative composition of tools still results in efficient components, which is a problem with standard approaches.

In the following, we discuss a few examples of tasks arising in computational mechanics and show how a solution can be built up from more general components. Among standard tools implemented in GRAL

are algorithms for generating incidence information (cell-neighbor search), traversal of boundary components (also for inner boundaries), grid subranges and their closures (e. g. vertices of a cell range), see [11] for more information. As these tools adhere to the language of grid(ranges) and the corresponding iterators sketched before, they are thus quite natural to use.

3.1 Mesh Quality Checking

Monitoring (and enhancing) mesh quality can be used to check the output of mesh generators, for ill-shaped and invalid cells, and is of special importance for Lagrangrian approaches, where the mesh deforms and gradually degrades.

A number of mesh quality measures are in use; here, we follow an approach introduced in [13], where also the relationship to other quality measures is discussed. The measure evaluates the condition number of matrices formed by the directions of edges around a vertex of a cell. Thus, it works for cell types which are *simple* polytopes (like hexahedrons and tetrahedrons), i. e. each vertex has degree 3 on the cell's surface mesh. For tetrahedrons, a weighting matrix has to be included, as the ideal corner is that of a regular tetrahedron.

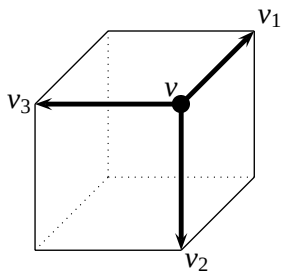

Figure 5: directions forming $A(v)$

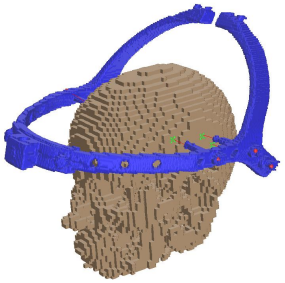

Figure 6: Halo positioning task

More formally, let v be a vertex of a cell c, and v_1, v_2, v_3 be the 3 vertices of c adjacent to v. We set $e_i = \Gamma(v_i) - \Gamma(v)$, Γ being the grid geometry, and $A(v) = (e_1, e_2, e_3)$ (cf. figure 5). Then we want to calculate the quantity

$$K_c(v) = \kappa(A(v)) \tag{1}$$

where $\kappa(A) = \|A\|\|A^{-1}\|$ is the condition number.

The algorithm for computing $K(v)$ can be broken down into the following steps, given v and c:

1. get the vertices v_i
2. set up the matrix $A(v) = (e_1, e_2, e_3)$
3. calculate the condition number $\kappa(A(v))$

Step 2 is straightforward. Step 3 can be solved in a generic way by generic algebraic components, which will not be discussed here. The challenging part for a *generic* implementation is step 1, because we cannot rely on a a-priori known way of numbering the vertices of a cell. For hex cells, there are $8!/24$ ways of numbering the vertices, unique up to a rotation. Here, cell archetypes and the switch operator

play a crucial role: Using switch, we can find the adjacent vertices for all vertices of a cell. By doing this in a preprocessing step on the cell archetypes, we need to perform this search only once for each archetype, and not for each cell.

As step 1 is a potentially useful building block also for other algorithms, we encapsulate it into a component which delivers the ordered sequence of adajacent vertices of any vertex of a given cell. More generally, this component also gives access to the corresponding sequence of edges and faces incident to both c and v. In lattice terminology, this set of edges and facets is called the *interval* $[v, c]$; it could be used e. g. for determining the solid angle at v.

3.2 Geometric Specification of Boundary Conditions

Handling boundary conditions often is a tedious task involving lots of manual interaction. In the specific example at hand occurring in maxillo-facial surgery within the SimBio project [14], we want to calculate the stress distribution of a human skull to which a so-called *halo device* is fixed with screws (see fig. 6). Both halo and head data are given as voxel images. The task involves finding the locations where the halo screws penetrate the skull. For this purpose, we model the screws as cylinders and their axes as rays pointing towards the skull, leading to the following steps:

1. Determine the screw data (ray and thickness) from the voxel data
2. extract the skull surface from the volume mesh built from the voxel data
3. Find the intersection of rays with the skull surface
4. Find the vertices or cells in vincinity of the intersections which are intersected by the corresponding cylinder

Again, each step in itself does not pose insurmountable problems. However, programming everything from scratch would be very tedious, so we want to find a way of combining standard tools to achieve the desired results.

Step 1 has been performed manually, as the halo geometry is considered stable, in contrast to the skull data taken from individual patients. Step 2 requires to find facets at a material boundary, and so facet-cell incidence information is needed. The latter can be calculated using a generic cell neighbor search algorithm which is a standard component of GRAL. Step 3 can be performed by a generic algorithm calculating intersections of rays (or lines) with a grid. Such a tool is useful in a broad range of applications, for instance computer graphics. A basic building block of this routine is a ray-triangle (or ray-polygon) intersection component, which is part of generic geometric primitives, not discussed here.

Finally, step 4 requires the ability to perform a traversal of the neighborhood of a surface mesh cell, which is possible if the surface mesh has cell-cell adjacencies. Instead of performing this traversal explicitly in the application code, we use a *view* which gives us access to a maximal connected component that includes some starting cell (here: the intersection cell) and all cells of which fulfil some predicate (here: at least one vertex lies within the corresponding screw cylinder). The same component is also useful when we decide to statically refine the neighborhood of the screw-skull intersection in the 3D volume grid: Due to the dimension-independent implementation, we can reuse the same generic tool!

4 Practical Issues: Questions and Answers

The viability of any software development approach has to be ultimately gauged by its practical usability, no matter what its theoretical merits are. As a rather young technique, generic programming for data-intensive scientific computing inavoidably still has some rough corners, but overall its practical value appears to be considerable. We discuss a few decisive issues in a question & answer style:

Q: How much work is it to (re)use a generic component?

A: If generic components are to be used with one's own data structures, one has first to create and adaptation layer implementing the kernel interface as far as necessary. Afterwards, *all* generic algorithms can be used without further effort. Be aware that the learning curve is quite steep!

Q: What restrictions does using generic components pose on the client (i. e. application) program?

A: Generic components can be used directly from C++ programs. For use in C/Fortran programs, we have to create an additional wrapper layer around generic library routines. No changes to data structures are needed in either case.

Q: How difficult is it to create generic library components with respect to conventional components?

A: In principle, there is no additional difficulty over conventional programming. One has to keep in mind, however, that while implementing a generic component, one reuses a lot of intellectual work spent on the development of the abstract concepts underlying the approach.

Q: What technical restrictions or risks are associated to using generic components?

A: We certainly have a stronger dependency on good tools (i. e. compilers). Generally compile-times grow, error messages can be ugly, and support from 3rd party tools may not be as good as expected. C++ templates do not fit into the traditional C/Unix library concept; solutions to these problems (such as incremental compilers or concept checks) are emerging only slowly.

Q: Which quality can be expected of generic components, in particular efficiency?

A: Concerning efficiency, the overhead (known as *abstraction penalty*) can in theory be optimized out. In practice this is compiler dependent and works for some examples, but for complicated stuff we will notice some overhead with respect to direct implementations. In the vast majority of cases, this overhead is absolutely tolerable, and with respect to API or file coupling discussed in the introduction, generic components perform much better and in particular avoid memory bottlenecks.

In sum, we can say that using generic libaries certainly requires some familarity with C++ template programming – such as can be gained by using the STL – but than it is definitely worth considering.

5 Conclusion

Generic programming for scientific, algorithm-dominated software is an emerging technology with a great potential to boost productivity. The present work shows how to carry over this approach to the field

of geometric algorithms. It is one of the first approaches to achieve *universally* usable algorithms in this area, in the sense that there is no more dependency on arbitrary data representation issues.

In spite of some practical problems addressed in the preceding section, the overall progress achieved by the generic approach is remarkable, and makes it ready for use in production code. One of its great strengths is its seamless integrability into existing bodies of code.

Besides the practical advantages of better reuse of geometric components, generic programming makes available an intellectual framework created during its development, consisting of a set of domain-specific concepts which can guide and shape reasoning about geometric software.

References

[1] The CGAL Consortium, *The CGAL home page*, `http://www.cgal.org` (1999).

[2] A. S. Glassner, ed., *Graphics Gems*, Academic Press (1990).

[3] M. Lee, A. A. Stepanov, *The standard template library*, Tech. rep., Hewlett-Packard Laboratories (1995).

[4] J. Siek, L.-Q. Lee, A. Lumsdaine, *BGL – the Boost Graph Library*, `http://www.boost.org/libs/graph/doc/table_of_contents.html` (2000).

[5] U. Köthe, *VIGRA homepage*, `http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/` (2000).

[6] E. Brisson, *Representing geometric structures in d dimensions: Topology and order*, in *Proc. 5th Annu. ACM Sympos. Comput. Geom.* (1989), pp. 218–227.

[7] G. Berti, *GrAL – the Grid Algorithms Library*, `http://www.math.tu-cottbus.de/~berti/gral` (2001).

[8] G. Berti, *Generic components for grid data structures and algorithms with C++*, in *First Workshop on C++ Template Programming, Erfurt, Germany* (2000).

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

[10] G. Berti, *Generic software components for Scientific Computing*, Ph.D. thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany (2000).

[11] G. Berti, *A generic toolbox for the grid craftsman*, in W. Hackbusch, U. Langer, eds., *Proceedings of the 17th GAMM Seminar on Construction of Grid Generation Algorithms*, Online proceedings at `http://www.mis.mpg.de/conferences/gamm/2001/` (2001).

[12] *The Geomview homepage*, `http://www.geomview.org`.

[13] P. M. Knupp, *Matrix norms & the condition number: A general framework to improve mesh quality via node-movement*, in *Proceedings of 8th International Meshing RoundTable*, Lake Tahoe (1999), pp. 13–22.

[14] *The SimBio EU-IST Project*, `http://www.simbio.de` (2000–2003).