A Generic Discretization Library

Michael Spevak
Institute for Microelectronics
Disastrous 27-29
1040 Vienna, Austria
spevak@iue.tuwien.ac.at

René Heinzl
Christian Doppler Laboratory
Gusshausstrasse 27-29
1040 Vienna, Austria
heinzl@iue.tuwien.ac.at

Philipp Schwaha
Christian Doppler Laboratory
Gusshausstrasse 27-29
1040 Vienna, Austria
schwaha@iue.tuwien.ac.at

ABSTRACT
We present a generic library which provides means to specify partial differential equations using different discretization schemes, dimensions, and topologies. Due to the common interfaces for simulation domains as well as numerical algebra we have an overall high inter-operability.

1. INTRODUCTION
One of the major topics in the field of scientific computing is the solution of differential equations. The field of differential equations covers various sub-fields of varying complexity and has different requirements on the underlying simulation domain as well as the mathematical formalism. In the most complex cases we face a system of coupled partial differential equations.

As mathematical structures such as scalar fields on a simulation domain do not have a direct mapping to data structures of a computer, discretization schemes and numerical methods have to be employed. During the last decades a vast number of different tools for the solution of differential equations has been developed. In general, the methods that need to be performed have not changed. Some of them have to be used together with other techniques. Based on a data structure which represents a cell complex, an equation system is assembled. After the solution of the equation system is computed the data are mapped back to the cell complex.

The main part of our work is the re-factoring and separation of the program structures needed for the discretization as well as the assembly of differential equations. By investigating numerous tools we found that various parts of code have been re-implemented in each of these tools. The tediously implemented domain-specific improvements introduced by each of the tools were not reusable in any way and had to be recoded repeatedly.

Typical programs operate on two different structures, namely a simulation domain and a matrix data structure. In the process of re-factoring and re-organizing of available code it is crucial to define all the external interfaces explicitly. The formalization of topological mechanisms allows the implementation of different discretization schemes independently from the actual representation of the topological data structure. The generic topology library (GTL) [7] provides an implementation of such a data structure which can be parametrized for arbitrary dimensions and cells.

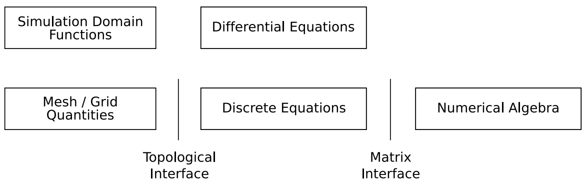

Figure 1: Interface dependencies of discretization schemes, solving algorithms and matrix interfaces and the simulation domain

The interface for the simulation domain requires access to the topological structure as well as the defined functions. The GTL models these interfaces and provides the necessary functionality to store values on the simulation domain, which are given discretely on the single topological elements. The topological functionality is mostly needed to fulfill the requirements of the discretization schemes such as the finite element method [14] or the finite volume method [12]. All of these schemes need a set of neighboring elements based on the topological property of **incidence**. Two elements are incident if one of the sets is a subset of the other.

The formalization of matrix access mechanisms for the use in numerical algebra provides the inter-operability and exchangability needed for different solver mechanisms [8]. This allows a comparison of different numeric algebra software packages under the same circumstances, which is usually not possible without a considerable amount of manual work.

There are many different interfaces [1, 8] available for the efficient assembly of equation systems. In general the interface can be reduced to only a few requirements. Even if there are solvers available which support highly specific storage structures such as band matrices, symmetric matrices or diagonal matrices, using different compressed matrix formats, the basic operations of the solvers are the same. The main problem of all of these solution mechanisms is the lack of a high level standard interface which reduces the detail of knowledge required by the end user.

We present a simple interface which makes the internal matrix structure completely interchangeable and due to orthogonality of the concepts we can reduce this effort of interfacing from $O(m \times n)$ to $O(m + n)$. The generic discretization library (GDL) introduces interfaces and makes them applicable to a huge number of problems. Its main aims are to formalize the way of discrete mathematical formula specification and to provide a formalized way of coding mathematical expressions.

The intended range of applications covers typical PDE methods as encountered in electrodynamics, mechanics, diffusion processes, fluid mechanics, and chemical reactions as well as typically discrete phenomena such as particle dynamics or even graph based problems.

The main focus of the design of the GDL was spent on specifying the mathematical formalisms. There have been approaches towards domain specific languages in this field [9, 2, 3, 11] but most of them are very specific and only work in special cases such as finite element or finite volume discretization schemes. We construct the base framework of topological operations, which covers different discretization schemes as well as purely discrete phenomena.

2. REQUIREMENTS RESULTING FROM DISCRETIZATION SCHEMES

The main aim of discretization schemes is to yield a numeric representation of a differential equation by projecting it onto a discrete simulation domain. Typically, the discretization of differential equations is achieved on elements of an underlying cell complex, e.g. on vertices. This results in an algebraic equation being assembled for each of the discrete elements of the complex. The assembled algebraic equations do not only involve values on single vertices, but also depend on neighboring elements. It is therefore impossible to solve the equations at each vertex locally and a set of coupled equations is obtained.

The discretization schemes we have investigated are the finite element method, the finite volume method, and as specialized case the finite difference method [13]. In particular, we show the different iteration mechanisms necessary for implementation.

The finite element method uses shape functions on the highest dimensional elements. Each global shape function is located on a vertex. Therefore shape-functions have non-zero values in all cells containing this vertex. The main aim of this discretization scheme is to find a weighted residual formulation. For the formulation of this discretization scheme the following operations have to be performed.

```
On all cells (c) in Neighborhood(v)
 On all vertices (w) in Neighborhood(c)
  ...
```

Finite volume schemes and finite difference schemes as well as the required topological iteration mechanisms are discussed in [4].

3. EXTERNAL INTERFACES

The main topological requirements on the simulation domain is an iteration over incident elements. A typical example of such an iteration is to find all incident edges of a vertex or vice versa. These operations use iterator-like structures [7] for all permutations of different topological elements. Apart from incidence and iteration, discretization schemes need the property of orientation. All elements have a standard orientation, e.g. an edge provides a source and a sink vertex. We define an orientation function $\mathcal{O}(a, b)$ between an edge and a vertex which returns $+1$ if a vertex coincides with the source and -1 if the vertex coincides with the sink (Figure 2). A full reference of the topological iteration possibilities can be found in [7].

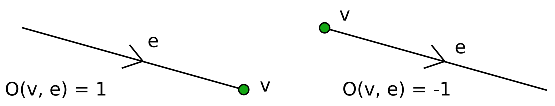

Figure 2: Orientation of an edge. The orientation function returns either $+1$ or -1 depending on whether the vertex is the sink or the source of the edge.

An interface to linear solvers has to provide several operations like the solution of a linear equation system, the inversion of a matrix or the retrieval of eigenvalues. Non-linear solver mechanisms are usually based on linear solvers and therefore can be matched to this interface.

All of these methods have in common that they are capable of handling matrices covering an arbitrary number of entries. At the time of initialization of the matrix data structure, the number of rows and columns of the matrix as well as the number of right hand side vectors has to be specified. In order to define the 10×10 matrix structure for a linear solver with three right hand side vectors we call the following constructor.

```
matrix_t msi(10, 3);
```

All matrix elements can be accessed by a function object. This interface can be used in order to obtain the values of the right hand side, the solution, eigenvalues or eigenvectors.

```
matrix_t::entry_accessor entry(msi);
matrix_t::rhs_accessor rhs(msi);

entry(0, 1) = 12.5;
rhs(2, 2) = 3.4;
```

Algorithms and data structures for linear algebra can be formulated independently. While simulation tools fill the entries of the matrix using properties of the simulation domain and the discretized differential equations, solver mechanisms operate on the matrix in order to provide the solution of the discretized problem.

4. A FUNCTIONAL CALCULUS FOR DISCRETIZATION

We have already listed the operations which are necessary to formulate discretization schemes. Based on the example of the Laplace operator we introduce the notions necessary for a fast as well as efficient implementation.

Due to space considerations we omit the means of matrix assembly. However we have to state that differential equations can be formulated using the associated differential operators. All differential equations $\mathcal{L}(f) = 0$ can be represented by their operator $\mathcal{L}(f)$. The underlying matrix mechanisms can be employed to determine all entries of the discretized differential equation and assemble the differential equations.

For the following considerations we only show the application of the differential operator.

4.1 Specification of Mathematical Formulae

The simplest expressions of our calculus are data which are stored on topological elements. Each of the discrete formulae has to access different values which are associated with the topological elements.

We assume a simulation domain where values are stored on vertices. Each vertex has one value ψ stored on it. If a function evaluation of the expression ψ has to be performed in a distinct vertex, we obtain this stored value.

If we limit the calculus to simple evaluations of the values of data, it unnecessarily impoverishes the resulting calculus without leading to a significant simplification of expressions. However, if we can combine these expressions using operators, we obtain a huge variety of combinations which cover a very broad range of expressions.

So far, the operations used do not differ very much from a typical functional approach such as the Boost Lambda Library or Boost Phoenix [5]. The major difference of mathematical expressions occurring in discretization to typical functional expressions is that the location of evaluation does not change in functional expressions. In the following example that shows the evaluation of the Laplace operator we see that for the calculation we do not only need values on the vertex of interest but we also have to access values within its neighborhood. We investigate the typical finite volume formulation of this differential operator.

$$\mathrm{div}(\mathrm{grad}(\psi_i)) = \sum_j \frac{\psi_j - \psi_i}{d_{i,j}} \cdot A_{i,j} \qquad (1)$$

Even though this formulation is common for finite volume formulations, it is not directly implementable in a computer. First and most importantly, the ranges of the sum as well as the indices i and j are not defined explicitly. A lot of information is implicit and is only valid within the framework of the specific discretization scheme. For the sake of generality, however, we can not keep domain specific notations.

There is only one iteration operation to adjacent vertices, there are data $A_{i,j}$ which seem to be evaluated on both of these vertices. More precisely, the formulation states: Doubly indexed data are stored on an edge (which is defined by the vertices i and j), singly indexed data fields are located on vertices. Even though the summation index j is specified, the kind of iteration (namely vertex-vertex adjacency over edges) is assumed implicitly. From the implementation point of view this specification can only be implemented using further assumptions.

Expression (1) can be re-organized in order to show a consistent iteration scheme which allows a formulation free of domain specific notational abbreviations. We use a sum as well as a difference based on topological iteration. The indices `ve` and `ev` denote a local neighborhood iteration from a vertex to an edge and vice versa (3, 4). We explicitly name the occurring topological elements, the initial vertex is called v, the edges are called e and the vertices derived by iteration are called w.

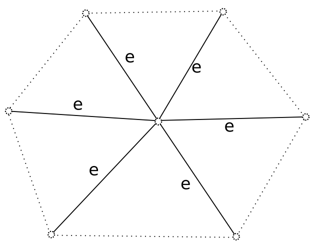

Figure 3: Exterior iteration loop. Starting from the base vertex, the incident edges are determined and traversed. The values of A as well as d are evaluated on the edges.

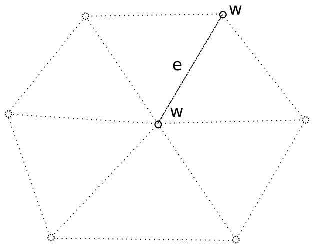

Figure 4: The interior iteration loop. The vertices which are incident with the edges are used for data access.

$$\mathrm{div}(\mathrm{grad}(\psi(v))) = \sum_{\mathrm{ve}(v,e)} \frac{A(e)}{d(e)} \Delta_{\mathrm{ev}(e,w)} \psi(w) \ . \qquad (2)$$

Even though this formulation provides explicit naming of the topological elements we can see that this is not necessary for most of them. In the outer sum the data accessed are located only on e, in the difference the data accessed are located only on w. Even though the accessed data are not always located on the actually traversed element, this holds true in most of the cases. Therefore we implicitly access the element that is actually being traversed in the innermost loop. This allows us to reduce our formulation without losing generality.

$$\mathrm{div}(\mathrm{grad}(\psi)) = \sum_{\mathrm{ve}} \frac{A}{d} \Delta_{\mathrm{ev}} \psi \ . \qquad (3)$$

However, the difference of elements is directly connected to the order in which the elements are traversed. Due to the fact that the implementation of the cell complex is based purely on topological neighborhood information, the order is completely free and the subtractions may be performed in an arbitrary order. We overcome this problem by generalizing the difference to more general summation processes using appropriate multiplicative factors to determine if a value has to be added or subtracted.

In order to define the sign, we use the orientation function $\mathcal{O}(a, b)$, (see Section 2) which is passed to the edge as well as the vertex (Figure 2). As a consequence we have to access the edge and pass it to the orientation function. As the edge orientation function uses the edge which is not the element of the innermost loop, we have to use an explicit name e.

$$\mathrm{div}(\mathrm{grad}(\psi)) = \sum_{\mathrm{ve}} \frac{A}{d} \sum_{\mathrm{ev}(e,w)} \psi \cdot \mathcal{O}(e, w) \ . \qquad (4)$$

With this calculus we can handle a wide range of mathematical formulations which orthogonally use the methods of topological iteration and conventional functional expressions.

5. APPLICATIONS

This section describes the basic structure of a generic discretization library as well as its application to differential equations using different discretization schemes. Based on the functional calculus we establish a set of functional expressions to specify arithmetic operations and function application to data in combination with topological iteration methods. We provide a few basic examples which show the application of the mentioned discretization schemes.

5.1 Finite Volumes

We show a possible procedural implementation of the Laplace operator first. It uses iterators which are gradually replaced by a functional approach in order to obtain a formulation which is close to the expression.

Using a purely generic approach with functional elements for data access applied to (4) yields:

```
double laplacian;
vertex_edge veit(v);
for(; veit.valid(); ++veit) {
  double inter = 0;

  edge_vertex evit(*veit)
  for(; evit.valid(); ++evit) {
    inter += psi(*evit) * Orient(*evit, *veit);
  }

  inter *= A(*veit) / d(*veit);
  laplacian += inter;
}
```

Conventional formula assembly using loops and accumulation implies the use of loop counters or iterators, as well as intermediate results, which are avoided in the mathematical formulation. We explicitly need to name the iterators `evit`, `veit` as well as the variable `inter`. This does not only force us to find names for these variables in each implementation but also introduces redundant information.

Apart from this fact, we can see that in most cases we only need the iterator of the innermost loop explicitly. This enables us to condense the formulation using function objects which provide accumulation in combination with topological iteration. For the sake of simplicity we omit the orientation function at this point.

```
sum<vertex_edge>(ZERO)[
  A(_1) / d(_1) * sum<edge_vertex>(ZERO)[
    psi(_1) ]]
(v);
```

Although this formulation does not cover the complete information, the expression already contains the semantic information of the original formula. In contrast to the Phoenix 2 library we use the unnamed function object `_1` for the element of the innermost loop. This means, that in the outer sum `_1` denotes the edge, whereas in the inner sum `_1` denotes the vertex.

Indeed, there are two significant problems with this formulation: First, the return value of functions is hard to determine, because it is not given explicitly. In addition, more general accumulation routines require some kind of neutral value to start the accumulation. For this reason, we explicitly insert the neutral element, in our case `ZERO`. As there are many kinds of accumulation operations (sums, products, all, exist) the value can not be coupled to the type but has to be specified explicitly.

The second problem arises when we introduce the orientation function. Even though such a function can be made available as a functional expression, the second argument, namely the edge is not available directly. In analogy to the Phoenix library we use a named variable `_e` to keep the local element available in the inner loops. These local variables are passed to the function objects using the local calling stack.

```
laplacian = sum<vertex_edge>(ZERO)[
  A(_1) / d(_1) * sum<edge_vertex>(ZERO, _e)[
    psi(_1) * Orient(_1, _e) ]]
(v);
```

This functional expression is equivalent to the mathematical formulation (4) and transforms the semantics into the C++ programming language.

5.2 Finite Elements

The finite element scheme uses an integral formulation in order to assemble partial differential equations. For each two points belonging to a common cell $\mathcal{C}$ (e.g. a triangle) an integral is evaluated. This integral determines a local summand for the differential operator.

```
sum<vertex_cell>(ZERO, _v)[
  sum<cell_vertex>(ZERO, _c)[
    psi(_1)*int(_c, _1, _v) ]]
(v);
```

All finite element formulations using shape functions which are located on the vertices can be re-formulated in this manner. For higher order finite elements, this method can be easily generalized to general neighborhood operations.

All equation-specific properties of the formulation can be encapsulated in the term `int(_c, _1, _v)`. The integral term returns the value of an integral, where $\mathcal{L}$ is the given differential operator

$$\mathrm{int}(\mathcal{C}, i, j) := \int_{\mathcal{C}} \mathcal{L}\psi_i(x) \cdot \psi_j(x) \cdot dV \ . \qquad (5)$$

The arguments passed to this term are the cell which is the domain of integration. This integral formulation can be evaluated either using an analytical formula or numerically. After the integrals are evaluated and the resulting linear equation is evaluated, it is entered into the global matrix.

6. IMPLEMENTATION

The GDL is based on the Phoenix2 library and excessively uses the interfaces provided there. The GDL provides three different kinds of function objects: Data accessors, functions and accumulators. We briefly show how accumulation is implemented using the Phoenix2 library. Data accessors are used in formulae in order to access values which are stored with respect to a topological element. These accessors can be adapted to the underlying property map and do not use topological features.

In the accumulation objects, topological information is combined with information of the data stored. We show the implementation of the following line.

```
sum<IteratorTag>(Initial)[Summand]
```

First, we use several object generators in order to beautify the code and to save manual effort for explicitly coding data types. For the implementation as Phoenix2 objects, we provide a class which implements a function `eval` as well as a meta-function `apply` which returns the return type of `eval` depending on the arguments passed.

The evaluation function requires the so called environment, which contains function arguments, as well as further objects for the summand and for the initial value, which are also implemented as Phoenix 2 data structures. In the following implementation, the types of the variables are omitted due to space considerations.

We obtain the first element of the passed environment and construct a GTL style iterator. Then we initialize the result value with the result value of the function object `init`. In the next snippet, we perform the iteration combined with the evaluation of the summand.

```
eval(Env & env, Initial & init, Summand & summand)
{
  base_elem(at<0>(env.args));
  Iterator iter(base_elem);
  result = init.eval(env);

  while(iter.valid()){
     result += summand.eval(newenv(env, *iter));
     ++iter;
  }
```

For the evaluation of the summand, we have to pass the value of the iterator. If we use a vertex on cell iterator of the GTL, this function is passed a cell. The summand, however, has to be passed a vertex. For this reason we introduce a function `newenv` which transforms the environment. All other variables of the environment are preserved, only the first variable is changed.

We briefly measure the loss of performance due to the achieved level of abstraction. This was tested for the calculation of a finite volume difference approximation of a Laplace operator. A three dimensional mesh is used and compared with respect to compile time as well as run time for the functional implementation as well as for its imperative analogon.

We found that the run time for evaluation of functional expressions was within the specified range of Phoenix2 which is about 1 per cent. The abstraction penalty was under one per cent. However resource use for the compilation of large functional expressions is not negligible. The evaluation of large functional data structures also requires large amounts of RAM. For a more in-depth benchmark of functional structures we refer to [6].

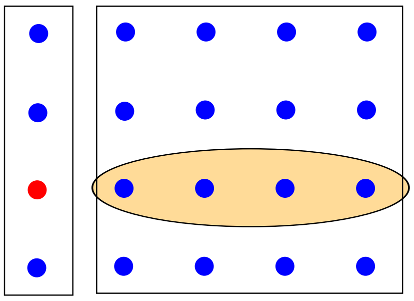

Figure 5: Two dimensional iteration. A two dimensional field of vector is iterated. The GDL provides general access mechanisms.

7. OUTLOOK

The GDL is not restricted to the operations used in scientific computing. Almost every branch of computer science deals with data structures with a more complicated underlying topology, and therefore can be reformulated to use an interface which is provided by the GTL. We show applications of this library which were not intended at the beginning but can also be performed using the GDL.

As a typical example we show an array data type (e.g. `std::vector`). If we can specify GTL style iterators which can be easily provided for each type of array we have the ability to access the underlying data structure via the GDL interfaces. Even though a vector can be specified by means of the GTL explicitly, one might have an implementation which relies on `std::vector`.

Such an expression is listed in the following snippet. A vector is traversed and all its elements are summed up. This is still possible using standard algorithms which is shown for comparison. For this reason we use some of the GTL functionality to represent the topological structure of the vector.

```
vector<int> vec;
sum<vector>(ZERO)[_1](vec);
accumulate(vec.begin(), vec.end(), ZERO, _1 + _2);
```

The following operations are not possible with standard algorithms but can be specified using functional environments like FC++ or Phoenix2 [10, 5]. We give a GDL as well as a Phoenix2 implementation. An iteration over a two-dimensional field is performed. (Figure 5)

```
vector<vector<int> > vec2;

sum<vector>(ZERO)[sum<vector>(ZERO)[_1]](vec2);

std::accumulate(vec2.begin(), vec2.end(), 0,
  _1 + phoenix::accumulate(_2, 0.0,
    lambda()[_1 + _2]))
```

In the following we show a simple example which exceeds the power of available functional frameworks. We perform an iteration over a container. During this iteration we accumulate the product of the values stored in the N elements which are topologically closest to the initial element. The set of these N elements is called a meta-cell (Figure 6). A the `meta_cell<N>` iterator of the GTL which provides the required functionality.

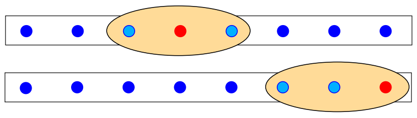

Figure 6: Metacell iteration. In this iteration the the 2 closest elements of a given element are traversed.

```
vector<int> vec;

sum<vector>(ZERO)[
  product<meta_cell<3> >(ZERO)[_1]]
(vec);
```

Even though FC++ and Phoenix2 provide container access, such operations can not be performed in an easy manner without rewritings of some components for this special case. Using the iteration data structures of the GTL, one can use arbitrary subsets for accumulation or iteration.

The applicability of the GDL strongly depends on the availability of topological iteration mechanisms on the underlying data structures. In most cases it is possible to establish such a layer. If data structures model the GTL interfaces it is also possible to specify general functional behavior via the GDL.

8. CONCLUSION

We have shown that the presented library closes the gap between the field of discretized equations and scientific application development. Apart from syntactic difficulties of C++, which complicate the formulation, the specified formulae are identical. Compiler error messages with a higher semantic level could even help the application designer to detect problematic code.

The library offers a possibility of very compact and minimalistic formulation. Even though some expressions can still be shortened, the use of the GDL reduces the effort of specification enormously. This does not only increase the speed of specification, but it also reduces the probability of typical errors.

The consequent use of the library does not only lead to a minimal effort of specification, but it also makes the programmer aware of the topological structures which are required for the discretization schemes. For this reason the framework supports direct implementation of mathematical formulations. Programmers and mathematicians have proved with well defined interfaces and the functional losses resulting from the explanation of formalisms is greatly reduced.

9. ADDITIONAL AUTHORS

Tibor Grasser, email: grasser@iue.tuwien.ac.at
Siegfried Selberherr, email: selberherr@iue.tuwien.ac.at

10. REFERENCES

[1] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. *Proc. Supercomp. '90*, pages 2–11, 1990.

[2] W. Bangerth, R. Hartmann, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. http://www.dealii.org.

[3] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – A General Purpose Object Oriented Finite Element Library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University, 2006.

[4] G. Berti. *Generic Software Components for Scientific Computing*. PhD thesis, Technische Universität Cottbus, 2000.

[5] Boost. *Boost Phoenix2*, 2006. http://spirit.sourceforge.net/.

[6] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser. Performance Aspects of a DSEL for Scientific Computing with C++. In *Proc. of the POOSC Conf.*, Nantes, France, July 2006.

[7] R. Heinzl, M. Spevak, P. Schwaha, and S. Selberherr. A Generic Topology Library. In *Library Centric Sofware Design, OOPSLA, accepted*, Portland, OR, USA, October 2006.

[8] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. K., R. B. L., K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Trans. on Math. Software*, 2005. For TOMS special issue on the ACTS Collection.

[9] Kevin Kramer, W. Nicholas, G. Hitchon, University of Wisconsin. *Semiconductor Devices, a Simulation Approach*. Prentice Hall Professional Technical Reference, 1997.

[10] B. McNamara and Y. Smaragdakis. Functional Programming in C++ using the FC++ Library. *SIGPLAN*, 36(4):25–30, Apr. 2001.

[11] C. Prud'homme. A Domain Specific Embedded Language in C++ for Automatic Differentiation, Projection, Integration and Variational Formulations. *Sci. Comp.*, page submitted, 2005.

[12] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer, Wien–New York, 1984.

[13] J. C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Chapman and Hall, 1989.

[14] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*. McGraw-Hill, Berkshire, England, 1987.