Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

SEBASTIAN GÖTSCHEL, MARTIN WEISER, ANTON SCHIELA

Solving Optimal Control Problems with the Kaskade 7 Finite Element Toolbox

ZIB-Report 10-25 (December 2010)

Solving Optimal Control Problems with the Kaskade 7 Finite Element Toolbox

Sebastian Götschel Martin Weiser Anton Schiela

December 14, 2010

Abstract

This paper presents concepts and implementation of the finite element toolbox Kaskade 7, a flexible C++ code for solving elliptic and parabolic PDE systems. Issues such as problem formulation, assembly and adaptivity are discussed at the example of optimal control problems. Trajectory compression for parabolic optimization problems is considered as a case study.

AMS MSC 2010: 65N30, 65M60, 65K10, 65Y99, 68U20

Keywords: partial differential equations, optimal control, finite elements, generic programming, adaptive methods

1 Introduction

Kaskade 7 is a general-purpose finite element toolbox for solving systems of elliptic and parabolic PDEs. Design targets for the Kaskade 7 code have been *flexibility*, *efficiency*, and *correctness*. One possibility to achieve these, to some extent competing, goals is to use C++ with a great deal of template metaprogramming [13]. This generative programming technique uses the C++ template system to let the compiler perform code generation. The resulting code is, due to static polymorphism, at the same time type and const correct and, due to code generation, adapted to the problem to be solved. Since all information relevant to code optimization is directly available to the compiler, the resulting code is highly efficient, of course depending on the capabilites of the compiler. In contrast to explicit code generation, as used, e.g., by the FEniCS project [9], no external toolchain besides the C++ compiler/linker is required. Drawbacks of the template metaprogramming approach are longer compile times, somewhat clumsy template notation, and hard to digest compiler diagnostics. Therefore, code on higher abstraction levels, where the performance gains of inlining and avoiding virtual function calls are negligible, uses dynamic polymorphism as well.

The Kaskade 7 code is heavily based on the DUNE libraries [2, 1, 3], which are used in particular for grid management, numerical quadrature, and linear algebra.

As a guiding example at which to illustrate features of Kaskade 7 we will use the all-at-once approach to the following simple optimal control problem. For a desired state y_d defined over a domain $\Omega \subset \mathbb{R}^d$, $d \in \{1,2,3\}$, and $\alpha > 0$ we consider the tracking type problem

$$\min_{u \in L^2(\Omega), y \in H_0^1(\Omega)} \frac{1}{2}\|y - y_d\|_{L^2(\Omega)}^2 + \frac{\alpha}{2}\|u\|_{L^2(\Omega)}^2 \quad \text{s.t.} \quad -\Delta y = u \quad \text{in } \Omega.$$

The solution is characterized by the Lagrange multiplier $\lambda \in H_0^1(\Omega)$ satisfying the Karush-Kuhn-Tucker system

$$\begin{bmatrix} I & & \Delta \\ & \alpha & I \\ \Delta & I & \end{bmatrix} \begin{bmatrix} y \\ u \\ \lambda \end{bmatrix} = \begin{bmatrix} y_d \\ 0 \\ 0 \end{bmatrix}.$$

For illustration purposes, we will discretize the system using piecewise polynomial finite elements for y and λ and piecewise constant functions for u, even though this is not the best way to approach this particular type of problems [7, 15].

2 Kaskade 7 Structure and Implementation

The foundation of all finite element computation is the approximation of solutions in finite dimensional function spaces. In the next section, we will discuss the representation of functions in Kaskade 7 before addressing the problem formulation.

2.1 Finite Element Spaces

On each reference element T_0 there is a set of possibly vector-valued shape functions $\phi_i : T_0 \to \mathbb{R}^s$, $i = 1, \ldots, m$ defined. Finite element functions are built from these shape functions by linear combination and transformation. More precisely, finite element functions defined by their coefficient vectors $a \in \mathbb{R}^N$ are given as

$$u(x)|_T = \psi_T(x)(\Phi(\xi) K_T a_{I_T}),$$

where $a_{I_T} \in \mathbb{R}^l$ is the subvector of a containing the coefficients of all finite element ansatz functions which do not vanish on the element T, $K \in \mathbb{R}^{m \times l}$ is a matrix describing the linear combination of shape functions ϕ_i to ansatz functions φ_j, $\Phi(\xi) \in \mathbb{R}^{s \times m}$ is the matrix consisting of the shape functions' values at the reference coordinate ξ corresponding to the global coordinate x as columns, and $\psi_T(x) \in \mathbb{R}^{s \times s}$ is a linear transformation from the values on the reference element to the actual element T.

The indices I_T and efficient application of the matrices K_T and $\psi_T(x)$ are provided by local-to-global-mappers, in terms of which the finite element spaces are defined. The mappers do also provide references to the suitable shape function set, which is, however, defined independently. For the computation of the index set I_T the mappers rely on the Dune index sets provided by the grid views on which the function spaces are defined.

For Lagrange ansatz functions, the combiner K is just a permutation matrix, and the converter $\psi(x)$ is just 1. For hierarchical ansatz functions in 2D and 3D, nontrivial linear combinations of shape functions are necessary. The implemented over-complete hierarchical FE spaces require just signed permuation matrices [16]. Vectorial ansatz functions, e.g. edge elements, require nontrivial converters $\psi(x)$ depending on the transformation from reference element to actual element. The structure in principle allows to use heterogeneous meshes with different element topology, but the currently implemented mappers require homogeneous meshes of either simplicial or quadrilateral type.

In Kaskade 7, finite element spaces are template classes parameterized with a mapper, defining the type of corresponding finite element functions and supporting their evaluation as well as prolongation during grid refinement, see Sec. 2.4. Assuming that `View` is a suitable Dune grid view type, FE spaces for the guiding example can be defined as:

```
typedef FEFunctionSpace<ContinuousLagrangeMapper<double,View> > H1Space;
typedef FEFunctionSpace<DiscontinuousLagrangeMapper<double,View> >
   L2Space;
H1Space h1Space(gridManager,view,order);
L2Space l2Space(gridManager,view,0);
```

Multi-component FE functions are supported, which gives the possibility to have vector-valued variables defined in terms of scalar shape functions. E.g., displacements in elastomechanics and temperatures in the heat equation share the same FE space. FE functions as elements of a FE space can be constructed using the type provided by that space:

```
H1Space::Element<1>::type y(h1Space), lambda(h1Space);
L2Space::Element<1>::type u(l2Space);
```

FE functions provide a limited set of linear algebra operations. Having different types for different numbers of components detects the mixing of incompatible operands at compile time.

During assembly, the ansatz functions have to be evaluated repeatedly. In order not to do this separately for each involved FE function, FE spaces define `Evaluators` doing this once for each involved space. When several FE functions need to be evaluated at a certain point, the evaluator caches the ansatz functions' values and gradients, such that the remaining work is just a small scalar product for each FE function.

2.2 Problem Formulation

For stationary variational problems, the Kaskade 7 core addresses variational functionals of the type

$$\min_{u_i \in V_i} \int_\Omega f(x, u_1, \ldots, u_n, \nabla u_1, \ldots, \nabla u_n)\, dx + \int_{\partial\Omega} g(x, u_1, \ldots, u_n)\, ds \qquad (1)$$

The problem definition consists of providing f, g, and their first and second directional derivatives in a certain fashion. First, the number of variables, their number of components, and the FE space they belong to have to be specified. This partially static information is stored in heterogeneous, statically polymorphic containers from the Boost Fusion [4] library. Variable descriptions are parameterized over their space index in the associated container of FE spaces, their number of components, and their unique, contiguous id.

```
typedef boost::fusion::vector<H1Space*,L2Space*> Spaces;
Spaces spaces(&h1Space,&l2Space);
typedef boost::fusion::vector<VariableDescription<0,1,0>,
                              VariableDescription<0,1,1>,
                              VariableDescription<1,1,2> > VarDesc;
```

Besides this data, a problem class defines, apart from some static meta information, two member classes, the DomainCache defining f and the BoundaryCache. The domain cache provides member functions d0, d1, and d2 evaluating $f(\cdot)$, $f'(\cdot)v_i$, and $f''(\cdot)[v_i, w_j]$, respectively. For the guiding example with

$$f = \frac{1}{2}(y - y_d)^2 + \frac{\alpha}{2}u^2 + \nabla\lambda^T\nabla y - \lambda u,$$

the corresponding code looks like

```
double d0() const {
  return (y-yd)*(y-yd)/2 + u*u*alpha/2 + dlambda*dy - lambda*u;
}
template <int i, int d>
double d1(VariationalArg<double,d> const& vi) const {
  if (i==0) return (y-yd)*vi.value + dlambda*vi.gradient;
  if (i==1) return alpha*u*vi.value - lambda*vi.value;
  if (i==2) return dy*vi.gradient - u*vi.value;
}
template <int i, int j, int d>
double d2(VariationalArg<double,d> const& vi,
          VariationalArg<double,d> const& wj) const {
  if (i==0 && j==0) return vi.value*wj.value;
  if (i==0 && j==2) return vi.gradient*wj.gradient;
  if (i==1 && j==1) return alpha*vi.value*wj.value;
  if (i==1 && j==2) return -vi.value*wj.value;
  if (i==2 && j==0) return vi.gradient*wj.gradient;
  if (i==2 && j==1) return -vi.value*wj.gradient;
}
```

A static member template class D2 defines which Hessian blocks are available. Symmetry is auto-detected, such that in d2 only $j \leq i$ needs to be defined.

```
template <int row, int col>
class D2 {
  static int present = (row==2) || (row==col);
};
```

The boundary cache is defined analogously. The functions for y, u, and λ are specified (for nonlinear or instationary problems in form of FE functions) on construction of the caches, and can be evaluated for each quadrature point using the appropriate one among the evaluators provided by the assembler:

```
template <class Position, class Evaluators>
void evaluateAt(Position const& x, Evaluators const& evaluators) {
    y = yFunc.value(at_c<0>(evaluators));
    u = uFunc.value(at_c<1>(evaluators));
    lambda = lambdaFunc.value(at_c<0>(evaluators));

    dy = yFunc.gradient(at_c<0>(evaluators));
    dlambda = lambdaFunc.gradient(at_c<0>(evaluators));
}
```

2.3 Assembly

Assembly of matrices and right hand sides for variational functionals is provided by the template class `VariationalFunctionalAssembler`, parameterized with a (linearized) variational functional. The elements of the grid are traversed. For each cell, the functional is evaluated at the integration points provided by a suitable quadrature rule, assembling local matrices and right hand sides. If applicape, boundary conditions are integrated. Finally, local data is scattered into global data structures. Matrices are stored as sparse block matrices with compressed row storage, as provided by the Dune `BCRSMatrix<BlockType>` class. For evaluation of FE functions and management of degrees of freedom, the involved spaces have to be provided to the assembler. User code for assembling a given functional will look like the following:

```
boost::fusion::vector<H1Space*,L2Space*> spaces(&h1space,&l2space);
VariationalFunctionalAssembler<Functional> as(spaces);
as.assemble(linearization(f,x));
```

For the solution of the resulting linear systems, several direct and iterative solvers can be used. An interface to Dune-ISTL is provided. E. g. the class `AssembledGalerkinOperator` provides the Dune `AssembledLinearOperator` interface for the Kaskade 7 class `VariationalFunctionalAssembler`. After the assembly, and some more initializations (`rhs`, `solution`), a direct solver `solverType` can be applied:

```
AssembledGalerkinOperator A(as);
directInverseOperator(A,solverType).applyscaleadd(-1.0,rhs,solution);
```

2.4 Adaptivity

Kaskade 7 provides several means of error estimation.

Embedded error estimator. Given a FE function u, an approximation of the error can be obtained by projecting u onto the ansatz space with polynomials of order one less. The method `embeddedErrorEstimator()` then constructs (scaled) error indicators, marks cells for refinement and adapts the grid with aid of the `GridManager` class, which will be described later.

```
error = u;
projectHierarchically(variableSet, u);
error -= u;
accurate = embeddedErrorEstimator(variableSet,error,u,scaling,tol,
   gridManager);
```

Hierarchic error estimator. After discretization using a FE space S_l, the minimizer of the variational functional satisfies a system of linear equations, $A_{ll}x_l = -b_l$. For error estimation, the ansatz space is extended by a second, higher order ansatz space, $S_l \oplus V_q$. The solution in this enriched space satisfies

$$\begin{bmatrix} A_{ll} & A_{lq} \\ A_{ql} & A_{qq} \end{bmatrix} \begin{bmatrix} x_l \\ x_q \end{bmatrix} = - \begin{bmatrix} b_l \\ b_q \end{bmatrix} .$$

Of course the solution of this system is quite expensive. As x_l is essentially known, just the reduced system $\operatorname{diag}(A_{qq})x_q = -(b_q + A_{ql}x_l)$ is solved [5]. A global error estimate can be obtained by evaluating the scalar product $\langle x_q, b_q \rangle$.

In Kaskade 7, the template class `HierarchicErrorEstimator` is available. It is parameterized by the type of the variational functional, and the description of the hierarchic extension space. The latter can be defined using e. g. the `ContinuousHierarchicExtensionMapper`. The error estimator then can be assembled and solved analogously to the solution of the original variational functional.

Grid transfer. Grid transfer makes heavy use of the signal-slot concept, as implemented in the Boost.Signals library [4]. Signals can be seen as callback functions with multiple targets. They are connected to so-called slots, which are functions to be executed when the signal is sent. This paradigm allows to handle grid modifications automatically, ensuring that all grid functions stay consistent.

All mesh modifications are done via the `GridManager<Grid>` class, which takes ownership of a grid once it is constructed. Before adaptation, the grid manager triggers the affected FE spaces to collect necessary data in a class `TransferData`. For all cells, a local restriction matrix is stored, mapping global degrees of freedom to local shape function coefficients of the respective father cell. After grid refinement or coarsening, the grid manager takes care that all FE functions are transfered to the new mesh. Since the construction of transfer matrices from grid modifications is a computationally complex task, these matrices are constructed only once for each FE space. To this extent, FE spaces listen for the `GridManager`'s signals. As soon as the transfer matrices are constructed, the FE spaces emit signals to which the associated FE functions react by updating their coefficient vectors using the provided transfer matrix. Since this is just an efficient linear algebra operation, transfering quite a lot of FE functions from the same FE space is cheap.

After error estimation and marking, the whole transfer process is initiated in the user code by:

```
gridManager.adaptAtOnce();
```

The automatic prolongation of FE functions during grid refinement makes it particularly easy to keep coarser level solutions at hand for evaluation, comparison, and convergence studies.

2.5 Nonlinear Solvers

A further aspect of Kaskade 7 is the solution of nonlinear problems, involving partial differential equations. Usually, these problems are posed in function spaces, which reflect the underlying analytic structure, and thus algorithms for their solution should be designed to inherit as much as possible from this structure.

Algorithms for the solution of nonlinear problems of the form (1) build upon the components described above, such as discretization, iterative linear solvers, and adaptive grid refinement. A typical example is Newton's method for the solution of a nonlinear operator equation. Algorithmic issues are the adaptive choice of damping factors, and the control of the accuracy of the linear solvers.

The control of accuracy includes requirements for iterative solvers, but also requirements on the accuracy of the discretization.

The interface between nonlinear solvers and supporting routines is rather coarse grained, so that dynamic polymorphism is the method of choice here. This makes it possible to develop and compile complex algorithms independently of the supporting routines, and to reuse the code for a variety of different problems. In client code the components are then plugged together, and decisions are made, which type of discretization, linear solver, adaptivity, etc. is used together with the nonlinear algorithm.

Core of the interface are abstract classes for a mathematical vector, which supports vector space operations, but no coordinatewise access, abstract classes for norms and scalar products, and abstract classes for a nonlinear functional and its linearization (or, more accurately, its local quadratic model). Further, an interface for inexact linear solvers is provided. These concepts form a framework for the construction of iterative algorithms in function space, which use discretization for the computation of inexact steps and adaptivity for error control.

The following shortened example code shows a simple implementation of the damped Newton method:

```
for(step=1; step <= maxSteps; step++) {
  lin = functional->getLinearization(*iterate);
  linearSolver->solve(*correction,*lin);
  do {
    *trialIter = *iterate;
    trialIter->axpy(dampingFactor,*correction);
    if(regularityTest(dampingFactor)==Failed) return -1;
    updateDampingFactor(dampingFactor);
  } while(evaluateTrialIterate(*trialIter,*correction,*lin)==Failed);
  *iterate = *trialIter;
  if(convergenceTest(*correction,*iterate)==Achieved) return 1;
}
```

While `regularityTest`, `updateDampingFactor`, `evaluateTrialIterate`, and `convergenceTest` are implemented within the algorithm, `functional`, `lin`, and `linearSolver`, used within the subroutines are instantiations of derived classes, provided by client code. By

```
linearSolver->solve(*correction,*lin);
```

a linear solver is called, which has access to the linearization `lin` as a linear operator equation. One may then either use a direct or iterative solver on a fixed discretization, or solve this operator equation adaptively, until a prescribed relative accuracy is reached. Then, the adaptive solver calls in turn a linear solver on each refinement step. There is a broad variety of linear solvers available, and moreover, it is not difficult to implement a specialized linear solver for the problem at hand.

For convenient implementation via Kaskade 7 concepts, bridge classes are provided, which are parametrized by Kaskade 7 types, and provide implementations for the abstract base classes needed by the algorithms. For example, the class `Bridge::KaskadeLinearization` is parametrized by a variational functional and a vector of type `VariableSet::Representation`. It uses the assembler class to generate the data needed for step computation, and manages generated data.

Several algorithms are currently implemented. Among them there is a damped Newton method with affine covariant damping strategy, a Newton path-following algorithm, and algorithms for nonlinear optimization, based on a cubic error model. This offers the possibility to solve a large variety of nonlinear problems involving partial differential equations. As an example, optimization problems with partial differential equations subject to state constraints can be solved by an interior point method combining Newton path-following and adaptive grid refinement [12].

3 State Trajectory Compression

As a case study we consider a 3D parabolic model optimal control problem

$$\min \frac{1}{2} \|y - y_d\|^2_{L^2(\Omega\times(0,T))} + \frac{\alpha}{2} \|u\|^2_{L^2(\partial\Omega\times(0,T))} ,$$

subject to

$$By_t - \Delta y = f \text{ in } \Omega \times (0, T), \quad \partial_\nu y + y = u \text{ on } \partial\Omega \times (0, T), \quad y(\cdot, 0) = 0 \text{ in } \Omega.$$

To avoid 4D discretization, methods working on the reduced objective functional are often employed. For the computation of the reduced gradient, one forward solve of the state equation as well as one backward solve of the adjoint equation is needed, see e. g. [7] and the references therein. As the state enters into the adjoint equation, the forward solution, a 4D data set, has to be stored. In this section, we will focus on the implementation of a compression scheme in Kaskade 7.

3.1 A Lossy Compression Algorithm

Assume a nested family $\mathcal{T}_0 \subset \cdots \subset \mathcal{T}_l$ of triangulations, constructed from a coarse grid $\mathcal{T}_0$. With $\mathcal{N}_j$ denoting the set of nodes on level j, let $\{\varphi_k | k \in \mathcal{N}_j\}$ be the piecewise linear nodal basis functions over the triangulation $\mathcal{T}_j$. Thus, the solution of the state equation is given by $y(x, t_i) = \sum_{k\in\mathcal{N}_l} y_{k,i}\varphi_k(x)$.

At time t_i we make use of the mesh hierarchy. Initially, the finite element coefficients for the coarse grid are predicted as zero. Given an approximation of $y_{k,i}, k = 0, \ldots, |\mathcal{N}_j|-1$ on grid level j, a prolongation of these values to the next grid level $j + 1$ is computed, e.g. by linear interpolation. As the exact nodal values are known, the prediction error can be evaluated, quantized keeping an error bound, and stored. The reconstruction of the nodal values then is used as input for the next prediction step, allowing the decoder to mirror the prediction process during the adjoint integration. As no random access to the state values is needed, differential encoding is applied to the quantized values to reduce temporal correlations. See [14] for details.

3.2 Time-dependent Problem Formulation

Kaskade 7 provides an extrapolated linearly implicit Euler method for integration of time-dependent problems $B(y)\dot{y} = f(y)$, [6]. Given an evolution equation `Equation eq`, the corresponding loop looks like

```
Limex<Equation> limex(gridManager,eq,variableSet);
for (steps=0; !done && steps<maxSteps; ++steps) {
  do {
    dx = limex.step(x,dt,extrapolOrder,tolX);
    errors = limex.estimateError(/*...*/);
    // choose optimal time step size
  } while( error > tolT );
  x += dx ;
}
```

Step computation makes use of the class `SemiImplicitEulerStep`. Here, the stationary elliptic problem resulting from the linearly implicit Euler method is defined. This requires an additional method `b2` in the domain cache for the evaluation of B. For the simple scalar model problem with $B(x)$ independent of y, this is just the following:

```
template<int i, int j, int d>
Dune::FieldMatrix<double, TestVars::Components<i>::m,
                          AnsatzVars::Components<j>::m>
  b2(VariationalArg<double,d> const& vi,
     VariationalArg<double,d> const& wj) const {
    return bvalue*vi.value*vj.value;
  }
```

Of course, `bvalue` has to be specified in the `evaluateAt` method.

3.3 Implementation

Implementation of the lossy compression algorithm makes use of the existing infrastructure for adaptivity, see Sec. 2.4. For the prolongation step, a FE space over the current `LevelGridView` is constructed. With help of the `TransferData` class, a transfer matrix is generated, which applied to a suitable coefficient vector `coeff` performs the transfer to the next grid level. This differs from the usual grid transfer, where the coefficient vector is prolongated to the *leaf* grid. For code reusability, this difference is taken care of by policy classes `AdaptationCoarseningPolicy` and `MulitlevelCoarseningPolicy`, which control the acceptance of cells during the creation of transfer matrices:

```
MultilevelCoarseningPolicy policy(level) ;
levelView = grid.levelView(level);
Space space(gridManager, levelView, order);
TransferData<Space,MultilevelCoarseningPolicy> transferData(space,
   policy);
levelView = grid.levelView(level+1);
space.mapper().update();
std::auto_ptr<TransferData<Space,
              MultilevelCoarseningPolicy>::TransferMatrix>
   tm = transferData.transferMatrix();
newCoeff = tm->apply(coeff) ;
```

Evaluation of the prediction error is straightforward, as the FE solution on the leaf grid is at hand. For quantization, the range of prediction errors on the current grid level is divided into N subintervals, where $N = \text{range}/(2\delta)$

is determined by the required quantization error tolerance δ. The subinterval indices are stored using a range coder [10].

For decoding the state values during the solution of the adjoint equation, the corresponding adaptively refined grids have to be stored as well. An efficient algorithm can be found in [8].

We apply the compression scheme to the model optimal control problem with data $\alpha = 10^{-5}$, $y_d(x,t) = t|x|^2$, $f(x,t) = |x|^2 - 4t$, discretized in space with linear finite elements on a uniform mesh. For minimization, a simple gradient-descent algorithm with an Armijo stepsize rule is used (see e. g. [11]). Fig. 1 shows the error in the reduced gradient and control induced by the lossy compression scheme, after 100 iterations of the optimization algorithm. For comparison, we estimate the discretization error for the reduced gradient and control using the quantities computed on a finer grid as reference.

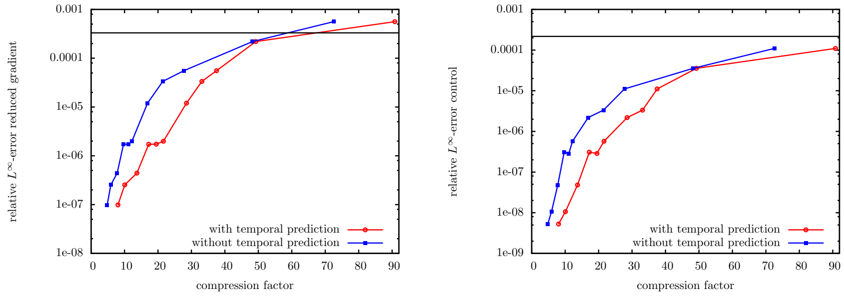

Figure 1: Relative error vs. compression rate for reduced gradient (left) and control (right) after 100 gradient steps. The horizontal line shows the approximated discretization error.

Acknowledgement Partial funding by the DFG Research Center MATHEON, projects A1, A17, and F9, is gratefully acknowledged.

References

[1] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in dune. computing. *Computing*, 82(2–3):121–138, 2008.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *Computing*, 82(2–3):103–119, 2008.

[3] M. Blatt and P. Bastian. The iterative solver template library. In B. Kåström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Scientific Computing*, pages 666–675. Springer, 2007.

[4] Boost. C++ libraries. `http://www.boost.org/`.

[5] P. Deuflhard, P. Leinen, and H. Yserentant. Concepts of an adaptive hierarchical finite element code. *IMPACT Comp. Sci. Eng.*, 1(1):3–35, 1989.

[6] P. Deuflhard and U. Nowak. Extrapolation integrators for quasilinear implicit ODEs. In P. Deuflhard and B. Engquist, editors, *Large Scale Scientific Computing*, volume 7 of *Progress in Scientific Computing*, pages 37–50. Birkhäuser, 1987.

[7] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich. *Optimization with PDE constraints*. Springer, Berlin, 2009.

[8] F. Kälberer, K. Polthier, and C. von Tycowicz. Lossless compression of adaptive multiresolution meshes. In *Proc. Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, volume 22, 2009.

[9] A. Logg. Automating the finite element method. *Arch Comput Methods Eng*, 14:93–138, 2007.

[10] G.N.N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. Presented at Video & Data Recording Conference, Southampton, 1979.

[11] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 2006.

[12] A. Schiela and A. Günther. Interior point methods in function space for state constraints – Inexact Newton and adaptivity. ZIB Report 09-01, 2009.

[13] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.

[14] M. Weiser and S. Götschel. State trajectory compression for optimal control with parabolic PDEs. ZIB Report 10-05, 2010.

[15] M. Weiser, T. Gänzler, and A. Schiela. Control reduced primal interior point methods. *Comput Optim Appl*, 41(1):127–145, 2008.

[16] G. Zumbusch. Symmetric hierarchical polynomials and the adaptive h-p-version. In A.V. Ilin and L.R. Scott, editors, *Proc. of the Third Int. Conf. on Spectral and High Order Methods, ICOSAHOM '95*, Houston Journal of Mathematics, pages 529–540, 1996.