# KASKADE7 Finite Element Toolbox
## *Programmers Manual*

M. Weiser, A. Schiela, S. Götschel, L. Lubkoll
B. Erdmann, L. Weimann, M. Moldenhauer, F. Lehmann, J. Schneck, J. Ullrich

June 10, 2021

# Contents

# 1 Introduction

KASKADE7 is a general-purpose finite element toolbox for solving systems of elliptic and parabolic PDEs. Design targets for the KASKADE7 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 [14]. 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 capabilities of the compiler. In contrast to explicit code generation, as used, e.g., by the FENICS project [12], no external toolchain besides the C++ compiler/linker is required. Drawbacks of the template metaprogramming approach are longer compile times, somewhat cumbersome 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 KASKADE7 code is heavily based on the DUNE libraries [6, 5, 7, 2], which are used in particular for grid management, numerical quadrature, and linear algebra.

TODO In Section 2 we describe the design and structure of KASKADE7 , also presenting some details of the implementation. The next section presents more practical advices to use the code. In particular, we give hints how to install the code and a set of third-party software needed in KASKADE7 .

Following the guideline of the sections 3, 4, and 5 the user can provide all the technical requirements necessary to start his/her own programming. This should be accompanied by the tutorial including a set of examples in Section 6.

Subsequent to that *Getting started* chapter we bring more and more details about programming of certain classes and modules helping the user of KASKADE7 to extend his knowledge and get familiar to all the topics interesting for developers. We close with a gallery of projects dealt with KASKADE7 , some notes on the history of development of the code, and finally a list of publications using simulation results provided by KASKADE7 .

# 2 Structure and Implementation

As a guiding example at which to illustrate features of KASKADE7 we will use in this section 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

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

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 $\lambda$ and piecewise constant functions for $u$, even though this is not the best way to approach this particular type of problems [11, 15].

The foundation of all finite element computation is the approximation of solutions in finite dimensional function spaces. In this section, we will discuss the representation of functions in KASKADE7 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 $a_{I_T} \in \mathbb{R}^k$ is the subvector of $a$ containing the coefficients of all finite element ansatz functions which do not vanish on the element $T$, $K \in \mathbb{R}^{m \times k}$ is a matrix describing the linear combination of shape functions $\phi_i$ to ansatz functions $\varphi_j$, $\Phi(\xi) \in \mathbb{R}^{s \times m}$ is the matrix consisting of the shape functions' values at the reference coordinate $\xi$ 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 permutation 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 KASKADE7 , 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:

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

The type aliases for common FE spaces as above are predefined in `fem/spaces.hh` for convenience. They are parametrized over the grid type (and scalar type, with double as default). The space definition above could thus be written as follows.

```
#include "fem/spaces.hh"
...
H1Space<Grid> h1Space(gridManager,view,order);
L2Space<Grid> 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 KASKADE7 core addresses variational functionals of the type

$$\min_{u_i \in V_i} J(u) = \int_\Omega F(x, u_1, \ldots, u_n, \nabla u_1, \ldots, \nabla u_n)\, \mathrm{d}x + \int_{\partial\Omega} G(x, u_1, \ldots, u_n)\, \mathrm{d}S. \quad (1)$$

with $u = (u_1, \ldots, u_n)^T$. In general, this problem is nonlinear. Therefore we formulate the Newton iteration in order to find the solution $u$:

$$J''(u^k)\delta u = -J'(u^k), \qquad u^{k+1} = u^k + \delta u^k \quad (2)$$

Hence, starting with an initial guess $u^0$ for $u$ we compute the Newton update $\delta u$ by

$$\int_\Omega F''(u^0)[\delta u, v]\, \mathrm{d}x + \int_{\partial\Omega} G''(u^0)[\delta u, v]\, \mathrm{d}S = -\int_\Omega F'(u^0)v\, \mathrm{d}x - \int_{\partial\Omega} G'(u^0)v\, \mathrm{d}S \quad \forall v \in V \quad (3)$$

with the Fréchet derivatives (directional derivatives in direction of $v$ and $\delta u$) of $F$ and $G$ of first and second order. The approximation after one step is

$$u^1 = u^0 + \delta u.$$

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 [1] 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 in arbitrary order.

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

Besides this data, a problem class defines, apart from some static meta information, two mandatory member classes, the `DomainCache` defining $F$ and the `BoundaryCache` defining $G$. 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.derivative;
  if (i==1) return alpha*u*vi.value - lambda*vi.value;
  if (i==2) return dy*vi.derivative - 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.derivative*wj.derivative;
  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.derivative*wj.derivative;
  if (i==2 && j==1) return -vi.value*wj.value;
}
```

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 $\lambda$ 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 Pos, class Evaluators>
void evaluateAt(Pos 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.derivative(at_c<0>(evaluators));
    dlambda = lambdaFunc.derivative(at_c<0>(evaluators));
}
```

**Hint.** Usage of `at_c<int>`

The function `at_c<int>()` allows one to access the elements of a heterogeneous array, that is an array, with possibly different data types (as opposed to the `std::array`, where the data type is fixed at initialisation to one type). An example for this is above in the method `evaluateAt()`. The data type `evaluators` contains evaluators of different type for the continuous H1 space used for state y and the discontinuous L2 space used for the control u. Having different types, the evaluators cannot be stored in a homogeneous array such as `std::vector` or `std::array`. Hence the at_c method allows to access both evaluators. The call `at_c<0>(evaluators)` reaches the first element of `evaluators`, while `at_c<1>(evaluators)` accesses the second and so forth.

## 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 applicable, 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 through an interface to DUNE-ISTL. For instance, the DUNE `AssembledLinearOperator` interface is provided by the KASKADE7 class `AssembledGalerkinOperator`. After the assembly, and some more initializations (`rhs`, `solution`), e.g. a direct solver `directType` can be applied:

```
AssembledGalerkinOperator A(as);
directInverseOperator(A, directType).applyscaleadd(-1.,rhs,solution
    );
```

## 2.4 Adaptivity

KASKADE7 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 $\mathrm{diag}(A_{qq})x_q = -(b_q + A_{ql}x_l)$ is solved [9]. A global error estimate can be obtained by evaluating the scalar product $\langle x_q, b_q \rangle$.

In KASKADE7 , 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 assembly and 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 [1]. 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. On that account, 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 Time-dependent Problems

KASKADE7 provides an extrapolated linearly implicit Euler method for integration of time-dependent problems $B(y)\dot{y} = f(y)$, [10]. Given an evolution equation `Equation eq`, the corresponding loop looks like

```
Limex<Equation> limex(gridManager,eq,variableSet);
for (int 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.

## 2.6 Nonlinear Solvers

A further aspect of KASKADE7 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. This 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. 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 can then be plugged together, and decisions are made, which type of discretization, linear solver, adaptivity, etc. is used together with the nonlinear algorithm. In this respect, KASKADE7 provides a couple of standard components, but of course users can write their own specialized components.

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.

In order to apply an algorithm to the solution of a nonlinear problem, one can in principle derive from these abstract classes and implement their purely virtual methods. However, for the interaction with the other components of KASKADE7 , bridge classes are provided, which are derived from the abstract base classes, and own an implementation.

We explain this at the following example which shows a simple implementation of the damped Newton method:

```
for(int 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(evalTrialIterate(*trialIter,*correction,*lin)==Failed);
  *iterate = *trialIter;
  if(convergenceTest(*correction,*iterate)==Achieved) return 1;
}
```

While `regularityTest`, `updateDampingFactor`, `evalTrialIterate`, and `convergenceTest` are implemented within the algorithm, `functional`, `lin`, and `linearSolver`, used within the subroutines are instantiations of de-

rived 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. It may either be a direct or an iterative solver on a fixed discretization, or solve this operator equation adaptively, until a prescribed relative accuracy is reached. In the latter case, 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.

The object `lin` is of type `AbstractLinearization`, which is implemented by the bridge class `Bridge::KaskadeLinearization`. This bridge class is a template, 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. From the client side, only the variational functional has to be defined and an appropriate set of variables has to be given.

Several algorithms are currently implemented. Among them there is a damped Newton method [8] 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 [**?**].

### 2.7 Module interaction

Figure 1 shows the interaction between the described modules.

## 3 Installation and code structure

### 3.1 Obtaining KASKADE7 and third-party software

KASKADE7 is currently maintained by the free and open source distributed version control system Git and is hosted on the server `https://git.zib.de/`. There are two branches: the master branch and the stable Kaskade7.4branch. Current development of KASKADE7 code happens on the master branch. For using the KASKADE7 code for own projects without troubles or bugs, please consider checking out the stable Kaskade7.4branch. If you don't mind working with the latest version which is under development, you can checkout the master branch. To

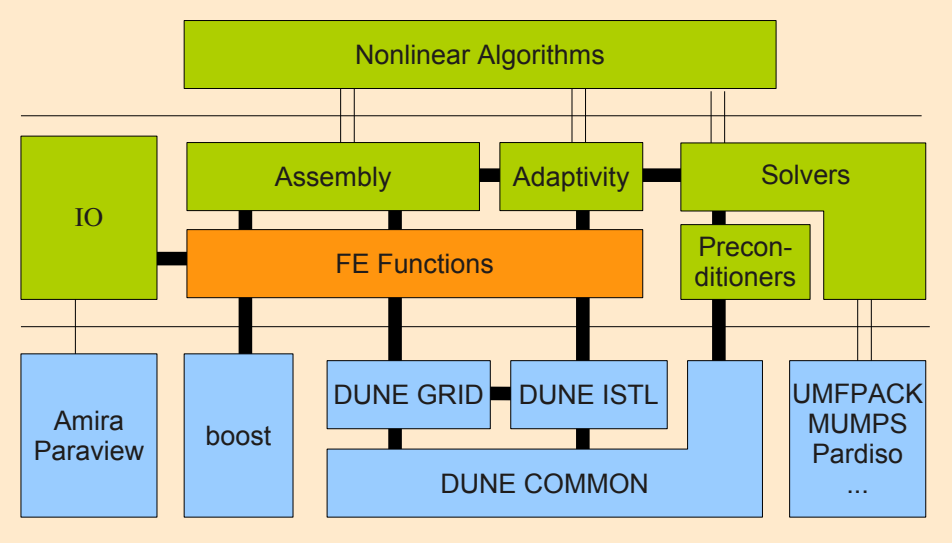


Figure 1: Module interaction

obtain a working copy of KASKADE7 from the ZIB Git repository you first need to be granted access rights to the repository by the admin. Then, open a terminal and change to the directory wherein you want to create the working copy subdirectory. Clone the KASKADE7 repository with SSH via

```
git clone git@git.zib.de:numerical-mathematics/...
       ...computational-anatomy-and-physiology/kaskade7.git
```

or clone with HTTPS via

```
git clone https://git.zib.de/numerical-mathematics/...
       ...computational-anatomy-and-physiology/kaskade7.git
```

You have now successfully cloned the master repository to your directory. If you want to work on a branch, i.e. the Kaskade7.4branch with the current version number, switch to it by performing

```
git checkout Kaskade<version-number>
```

which updates the index and the files in the working tree and points the HEAD at the branch.

**Installation** If you intend to run your KASKADE7 copy on a system different from the 64-bit Linux system at ZIB, you have to install the required compilers and external libraries. A collection of shell installer script can be found in the `InstallDependencies/` subdirectory. ome information can be found in `InstallDependencies/README.md`. You must inspect and change the shell script to your needs. Start by editing `install.sh` and define versions, paths and URLs. Start the installation process by

```
cd InstallDependencies; sh install.sh
```

If anything goes wrong, inspect and change the corresponding script. They are intended to be modified as needed, and therefore mostly simple and readable.
If you are a ZIB member, an installation of required libraries is maintained and you can skip the library installation step.

Next modify `Makefile.Local` to reflect the installation paths of KASKADE7 and the required libraries, i.e. change the path of `KASKADE7 = ...` to your installation directory. If you installed the third-party libraries manually, you'll have replace the path

```
include /data/numerik/software/KaskadeDependencies/...
      ...Kaskade7.5Dependencies-10.2/installed/Makefile.Local
```

by the path to your installation. Then,

```
make kasklib
```

builds the library. You can run the test suite with `make test` and the tutorials with `make tutorial`. With a successful run of the above commands you can now start working with the KASKADE7 library.

## 3.2 Structure of the program

After checking out the program to directory KASKADE7 we have the following structure of the source code stored into corresponding subdirectories.

- KASKADE7/algorithm
- KASKADE7/doc
- KASKADE7/fem
- KASKADE7/io
- KASKADE7/linalg

- KASKADE7/mg
- KASKADE7/tests
- KASKADE7/timestepping
- KASKADE7/tools
- KASKADE7/tutorial
- KASKADE7/utilities

In these subdirectories we find `.hh-` and `.cpp-` files, some auxiliary files and sometimes further subdirectories. In directories including `.cpp-` files there is a `Makefile` generating object files or libraries which are stored into `KASKADE7/libs`.

### 3.3 Compiler und Libraries

Before calling `make` in KASKADE7 directory or in one of the subdirectories the user should make sure that the correct compiler (corresponding to the selection in the Makefiles) is available. In the installation at ZIB this can be provided by modifying the PATH variable, e.g. using the bash shell

```
export PATH="/home/data/numerik/archiv/software/linux64/gcc-$VERSION_GCC/gcc/bin:$PATH"
```

or similar in csh shell:

```
setenv PATH /home/data/numerik/archiv/software/linux64/gcc-$VERSION_GCC/gcc/bin:$PATH
```

where we have to set $*VERSION\_GCC*$ to "7.1.0" or "7.2.0" in version Kaskade7.4(note that the given paths (`/home/data/numerik/...`) are adjusted to htc?? /bin/sh: /datanumerik/... htc use).
In order to assure that all needed shared libraries are found we have to set the `LD_LIBRARY_PATH` variable to the shared libraries libmpfr.so, libmpc.so, libgmp.so (used when the compiler is called):

```
export LD_LIBRARY_PATH="/home/data/numerik/archiv/software/linux64/lib"
```

or similar in a csh shell:

```
setenv LD_LIBRARY_PATH "//home/data/numerik/archiv/software/linux64/lib"
```

Note: on MacOS X machines we have to set the variable `DYLD_LIBRARY_PATH` with the corresponding paths.

## 3.4 Using the make command

When working with KASKADE7 you will frequently encounter the `make` command, i.e., when compiling and linking a program or cleaning up a subdirectory. A brief introduction to `Makefiles` can be found under the link
`http://matt.might.net/articles/intro-to-make/`
Now let us take a look at a few KASKADE7 specific `make` calls.

**Makefile.Local.** In the root directory the file `Makefile.Local` contains paths to third-party software, compilers, as well as flags for compiler and debugger. In particular, the full path of the `kaskade7` directory has to be defined, as described above in the installation procedure.

**Makefile.Rules** This file contains a set of compiler flags which are set when compiling a KASKADE7 program.

**make in the KASKADE directory.** In the root directory `kaskade7` the Makefile may be called with different parameters. The functionality of those parameters can be set in the Makefile directly.

- **make clean** removes all object files, executables, graphics out, and some other files
- **make cleantutorial** removes all object files, executables, and some other files in the `subdirectories` stated in the Makefile
- **make depend** generates the `Makefile` in the KASKADE7 subdirectories from the `Makefile.gen` template, which includes dependencies to all (possibly nested) included c++ headerfiles of KASKADE.
- **make kasklib** generates object files and builds the library `libkaskade.a`.
- **make install** combines the two commands *make depend* and *make kasklib*.
- **make tutorial** builds and executes the examples in the `tutorial` , and builds the pdf version of the manual.
- **make test** builds and executes the examples in the `benchmarking` subdirectories.
- **make distribution** generates tar.gz - files after executing **make clean**, ignoring SVN information, documentation, and particular subdirectories

- **make doc** creates the Doxygen documentary
- **make manual** compiles the `.tex` files and creates the manual as pdf

For installing the complete KASKADE7 use the three top make commands from above in the order specified: **clean**, **depend**, **kasklib**. Note, that first some shell variables (i.e., *PATH* and *LD_LIBRARY_PATH*) have to be defined as described in the paragraph 3.3.

**make in subdirectory.** Once you have a complete installation of KASKADE7 it may be necessary (after changing a file) to recompile in a subdirectory and update the KASKADE7 library. This is done by using the file `Makefile` in the corresponding subdirectory. You just have to type `make`.

Note, that after checking out the code from the repository there are no files `Makefile` in the subdirectories but only files called `Makefile.gen`. Such a `Makefile.gen` can be used to generate a corresponding `Makefile` by typing:

- make -f Makefile.gen depend

Thus the KASKADE7 header files dependencies are detected and registered in the `Makefile`. Each `Makefile.gen` includes a `depend` and a `clean` option. Note that a call **make depend** in the KASKADE directory also generates the local `Makefile` from the local `Makefile.gen` in each of the subdirectories mentioned in the `Makefile`.

## 3.5 Tutorial and examples

Applications of the KASKADE7 software can be found in the subdirectories

- KASKADE7/tests
- KASKADE7/tutorial

Each of these subdirectories needs a `Makefile.gen` with the properties mentioned above. In the subdirectory `KASKADE7/tutorial` you find examples which are described in detail in this manual starting with Section 6. Switch to your KASKADE7 directory. Now, all tutorials are executed with the command

```
make tutorial
```

If `make tutorial` fails after `make clean`, try running `make kasklib` before.

To run these tutorials individually, you can edit the Makefile at `TUTORIALMODULES = laplacian` and comment out the other tutorials, e.g. `# stationary_heattransfer`

by the use of #. Otherwise, from the `kaskade7` directory in your shell you can run this tutorial by creating an executable file with the command `make tutorial/laplacian/laplace` followed by the command `tutorial/laplacian/laplace` to execute. Or just switch to the directory where `laplace` is located and type

```
make laplace
```

Some calculations create `.vtu` files, which store graphical data such as mesh and solution data. These files can be visualized by various tools, e.g. *paraview*.

## 3.6 Testing

The directory `KASKADE7/tests` provides a set of test examples. In addition to the examples in the tutorial we investigate here not only whether the computation is running but also whether it computes the correct results. Like the tutorial examples you can choose which tests are to be executed by editing the `Makefile.Local`. Note that running the `AmiraIO` test requires a license for *Amira*, see chapter 4. The testing is started in the KASKADE7 root directory by typing

```
make test
```

The results will be summarized in the file `testResult.txt`.

## 3.7 Communication with Git repository

Above in this section we described how to get a copy from the KASKADE7 Git repository. This copy can be used for arbitrary applications. Any change by the user is allowed. Git provides a set of commands to communicate between the local copy of a user and the current state of the repository. Thus there are Git commands to add a new file into the repository, to delete a file from the repository, to update local files or to commit local changes to the repository.

Enter the command

```
git help
```

in your shell to get a first idea of the options offered by Git:

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
[--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
[-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
<command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
```

```
clone      Clone a repository into a new directory
init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
add        Add file contents to the index
mv         Move or rename a file, a directory, or a symlink
reset      Reset current HEAD to the specified state
rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
bisect     Use binary search to find the commit that introduced a bug
grep       Print lines matching a pattern
log        Show commit logs
show       Show various types of objects
status     Show the working tree status

grow, mark and tweak your common history
branch     List, create, or delete branches
checkout    Switch branches or restore working tree files
commit     Record changes to the repository
diff       Show changes between commits, commit and working tree, etc
merge      Join two or more development histories together
rebase     Reapply commits on top of another base tip
tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
fetch      Download objects and refs from another repository
pull       Fetch from and integrate with another repository or a local branch
push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

The correct syntax of these Git commands can easily be found in the internet, e.g. `https://git-scm.com/docs/git#_git_commands`. Some more ressources are provided in the appendix B.1 or simply perform a search in your preferred search engine.

A simple workflow example is presented here. Change into your cloned KASKADE7 directory. To update e.g. the manual files after a review, first type

```
git status
```

and you will see the working tree status. If you made some changes to a file, Git notices differences between the working tree and the index file and will display this file here. If you created a new file, this file is yet not known to Git (i.e. not tracked) and will be displayed somewhere below. If there are other files staged for the next commit, they are displayed here as well.

Next, you want to add your changed file and maybe a new file to the index before committing with

```
git add <file-name>
```

and have another look at `git status`. The added files are now shown on top. Now, create a commit containing the current contents of the index and write a short log message describing your changes:

```
git commit -m <a few updates to the manual>
```

Afterwards, your file content is ready to be pushed (i.e. load) from your local repository into the remote repository of the master branch. This is the last step:

```
git push
```

Great, you're done and updated your first file!

# 4 External libraries

TODO PARDISO??

- ALBERTA: Directory alberta-3.0.1/lib (optional) C
  - **libalberta_3d.a**
    ALBERTA 3D-grid routines.
  - **libalberta_2d.a**
    ALBERTA 2D-grid routines.
  - **libalberta_1d.a**
    ALBERTA 1D-grid routines.
  - **libalberta_utilities.a**
    ALBERTA common utilities routines.
- AMIRAMESH: Directory libamira/lib (license restrictions apply!) C
  - **libamiramesh.a**
    Reading and writing Amiramesh files.
- BOOST: Directory boost-1.70.0/lib C
  - **libboost_signals.SUFFIX**
    Managed signals and slots callback implementation.
  - **libboost_program_options.SUFFIX**
    The program_options library allows program developers to obtain program options, that is (name, value) pairs from the user, via conventional methods such as command line and config file.
  - **libboost_program_system.SUFFIX**
    Operating system support, including the diagnostics support that will be part of the C++0x standard library.

  - **libboost_program_timer.SUFFIX**
    Event timer, progress timer, and progress display classes.
  - **libboost_program_thread.SUFFIX**
    Portable C++ multi-threading.
  - **libboost_program_chrono.SUFFIX**
    Useful time utilities.
  - **SUFFIX** is **so** under Linux and **dylib** under MacOS X (Darwin)

- DUNE: Directory dune-2.6.0/lib C
  - **libdunecommon.a**
    DUNE common modules.
  - **libdunegeometry.a**
    DUNE geometry modules.
  - **libdunegrid.a**
    DUNE grid methods.
  - **libdunealbertagrid_3d.a**
    DUNE interface to ALBERTA 3D-grid routines. (optional)
  - **libdunealbertagrid_2d.a**
    DUNE interface to ALBERTA 2D-grid routines. (optional)
  - **libdunealbertagrid_1d.a**
    DUNE interface to ALBERTA 1D-grid routines. (optional)
  - **libdunegridglue.a**
    DUNE library for contact-problems (optional)

- HYPRE: Directory hypre-2.11.2/lib C
  - **libHYPRE.a**

- ITSOL: Directory itsol-2/lib C
  - **libitsol.a**

- MUMPS: Directory mumps-5.1.2/lib
  Direct sparse linear solver library. C
  - **libdmumps.a**
  - **libmpiseq.a**
  - **libmumps_common.a**

  - **libpord.a**
  - **libpthread.a**

- TAUCS: Directory taucs-2.0/lib
  Preconditioner library. C
  - **libtaucs.a**

- UG for DUNE: Directory dune-2.6.0/lib
  UG for DUNE, FEM grid-library. C
  - **libugS3.a**
  - **libugS2.a**
  - **libugL3.a**
  - **libugL2.a**
  - **libdevS.a**
  - **libdevX.a**

- UMFPACK: Directory umfpack-5.4.0/lib
  Direct sparse linear solver library. C
  - **libumfpack.a**
  - **libamd.a**

# 5 Documentation online

In subdirectory `Kaskade7/doc` there is a script `makeDocu` for generating a documentation of the source code. Necessary is the program `doxygen` and a `Latex` installation. Outline and shape of the documentation is steered by the `doxygen` parameter file called `Doxyfile`.

By default ( $GENERATE\_HTML = YES$ ) the generation of HTML pages is selected. The source files to be analysed are defined via `INPUT` variable.

If the auxiliary program `dot` of the `GraphViz` software is available, we recommend to change the preset $HAVE\_DOT = NO$ to $HAVE\_DOT = YES$ in the file `Doxyfile`.

After generating the documentation ( by command `makeDocu` ) the pages may be considered in the browser by specifying the full path $.../kaskade7/doc/html/index.html$.

# 6 Getting started

In this chapter we present a set of examples which enables a user wihout any experience with KASKADE7 to get started. In particular, the first example specifies all the steps needed to understand the technical handling of KASKADE7 independent of any know-how about numerics and implementation. The following examples give more and more details, bringing together the mathematical formulation of a problem and its implementation.

## 6.1 A very first example: Laplacian, the simplest stationary heat transfer equation

### 6.1.1 Compile and execute this example

This example is implemented in subdirectory

KASKADE7/tutorial/laplacian

Files: **laplace.cpp**, **laplace.hh**, **Makefile**

Once appropriate changes are made to the code or the executable does not exist, the user has to compile and generate the executable by calling the `Makefile` (recall that the `Makefile` was already generated during the installation process as introduced in Section 3) while being in the KASKADE7 main directory. Just enter the command

```
make tutorial/laplacian/laplace
```

in the terminal.
If this `make` procedure works without errors you get the executable file `laplace` which can be run by the command

```
tutorial/laplacian/laplace
```

or, while being in the respective folder, by

```
./laplace
```

in the terminal. The program sends some messages to the terminal (about progress of the calculation) and writes graphical information (mesh and solution data) to a file *temperature.vtu*. This file can be visualized by any tool (e.g., *paraview*) which can interprete *vtk* format.

The shell command

```
make clean
```

deletes the files *laplace*, *laplace.o*, *temperature.vtu*, and *gccerr.txt*. The last one is only generated if errors or warnings are discovered by the compiler. It includes all the error messages and warnings in a single file which offers a more comfortable way to analyse the errors than to get all error messages at once on the terminal.

We summarize: In order to get a running code which computes a finite element soluton of the Lapacian problem, the following three commands, while in the main KASKADE7 directory, have to be entered

```
make clean
make tutorial/laplacian/laplace
tutorial/laplacian/laplace
```

### 6.1.2 Problem formulation

We search for the solution $u$ of the Laplacian or Poisson equation

$$\begin{aligned} -\Delta u = -\nabla\cdot(\nabla u) = 1 \qquad & \text{x } \in \Omega \\ u = 0 \qquad & \text{on } \Gamma \end{aligned} \tag{4}$$

on the two-dimensional open unit square $\Omega$ under homogeneous boundary conditions on $\Gamma = \partial\Omega$. These equations may describe stationary heat transfer caused by a constant heat source (value 1 on the right-hand side) and constant temperature ($0^oC$) on the boundary, e.g. by cooling.

Resolving the $\nabla$ - operator in the equation (18), we can also write it in Cartesian coordinates $x$ and $y$

$$\begin{aligned} -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = 1 \qquad & (x,y) \in (0,1)^2 \\ u = 0 \qquad & \text{on } \Gamma \end{aligned} \tag{5}$$

The treatment of this problem in context of finite element methods as used in KASKADE7 is based on a variational formulation of the equation (18). It is necessary to provide a triangulation of the domain $\bar{\Omega}$ (including the boundary), a set of ansatz functions (order 1: linear elements, order 2: quadratic elements,...), functions for evaluating the integrands in the weak formulation, assembling of the stiffness matrix and right-hand side. All this is to be specified in the files *laplace.cpp* and *laplace.hh* using the functionality of the KASKADE7 library. Shortly, we will explain the details and possibilities to change the code.

Now we explain the fundamentals of treating this problem in KASKADE7 . Since test and trial space are the same, we can reformulate the variational equation and get a minimization problem: Using the notation of functional $J(u)$ from

Section 2.2 we define for this example

$$F(u) := \frac{1}{2}\nabla u^T \nabla u - fu$$

and

$$G(u) := \gamma\frac{1}{2}(u-u_0)^2, \quad u_0 = 0$$

with a (large, e.g., $10^9$) penalty factor $\gamma$ ensuring sufficiently accurate satisfaction of the Dirichlet boundary condition. The penalty method is widely used in numerical optimization to incorporate constraints. In this context, we want to minimize $J(u)$ not over $H_0^1$ (i.e. the space of functions in $H^1(\Omega)$ which vanish on $\Gamma$), but over $H^1$. The penalty method enables the use of $H^1(\Omega)$ by imposing the constraint (i.e. the Dirichlet boundary condition) into a penalty term in the minimization formulation (see [4], [13]). Hence, the minimizer $u$ of $J(u)$ is a solution of equation 5.
We compute a solution of the minimization problem

$$\min_{u_i \in V_i} J(u) = \int_\Omega F(x, u_1, \ldots, u_n, \nabla u_1, \ldots, \nabla u_n)\, \mathrm{d}x + \int_{\partial\Omega} G(x, u_1, \ldots, u_n)\, \mathrm{d}S$$

as in (1) by a Newton iteration, solving in each step the following problem:

Find $u \in V \subset H^1(\Omega)$ such that

$$\int_\Omega F''(\cdot)[u,v]\, \mathrm{d}x + \int_{\partial\Omega} G''(\cdot)[u,v]\, \mathrm{d}S = -\int_\Omega F'(\cdot)v\, \mathrm{d}x - \int_{\partial\Omega} G'(\cdot)v\, \mathrm{d}S \quad \forall v \in V \quad (6)$$

where $u$ is the Newton update (being $\delta u$ in section 2.2) and consider for implementation

$$\begin{array}{llll} d1^\Omega(v) & = F'(\cdot)v, & d1^\Gamma(v) & = G'(\cdot)v, \\ d2^\Omega(u,v) & = F''(\cdot)[u,v], & d2^\Gamma(u,v) & = G''(\cdot)[u,v]. \end{array}$$

In our context $V$ is always a finite element space spanned by the base functions $\{\varphi_i\}_{1,\ldots,N}$,with $N$ as the dimension of the space. That means, we have to solve the following system of $N$ equations in order to find the minimum in the space $V$:

$$\int_\Omega d2^\Omega(u,\varphi_i)\, dx + \int_\Gamma d2^\Gamma(u,\varphi_i)\, ds = \int_\Omega d1^\Omega(\varphi_i) + \int_\Gamma d1^\Gamma(\varphi_i)\, ds \qquad \text{for} \quad \varphi_i,\ i = 1,\ldots,N. \quad (7)$$

Here we have

$$d0^{\Omega}() = \frac{1}{2}\nabla u^T \nabla u - fu \tag{8}$$
$$d1^{\Omega}(\varphi_i) = \nabla u^T \nabla \varphi_i - f\varphi_i \tag{9}$$
$$d2^{\Omega}(\varphi_i, \varphi_j) = \nabla \varphi_i^T \nabla \varphi_j \tag{10}$$

in the region $\Omega$, and

$$d0^{\Gamma}() = \gamma \frac{1}{2}(u - u_0)^2 \tag{11}$$
$$d1^{\Gamma}(\varphi_i) = \gamma (u - u_0)\varphi_i \tag{12}$$
$$d2^{\Gamma}(\varphi_i, \varphi_j) = \gamma \varphi_i \varphi_j \tag{13}$$

on the boundary $\Gamma$.

These functions have to be defined in two mandatory classes in the problem class (called HeatFunctional): the *DomainCache* defining $d0^{\Omega}()$, $d1^{\Omega}()$, and $d2^{\Omega}()$ for the region $\Omega$, and the *BoundaryCache* defining $d0^{\Gamma}()$, $d1^{\Gamma}()$, and $d2^{\Gamma}()$ on the boundary $\Gamma = \partial\Omega$.

In general the functional to be minimized depends nonlinearly on the solution $u$. However, in this example it is linear, hence the first step of Newton's method already provides the solution and is implemented in *main()* as shown below. In case of a nonlinear functional we have to write a complete Newton loop controlling the size of the update and stopping if it is small enough. We present such an example (from elasticity) later in this chapter.

Now, we focus on the specific details of implementation for our Laplacian problem which can be found in the files *laplace.cpp* and *laplace.hh*. We are not trying to explain everything. Just some hints to the essentials in order to get a first feeling for the code.

The code in the main program (file *laplace.cpp*)

```
int main()
{
  std::cout << "Start Laplacian tutorial program" << std::endl;

  constexpr int dim = 2;
  int refinements = 5,
      order       = 2;

  using Grid = Dune::UGGrid<dim>;
  using LeafView = Grid::LeafGridView;
```

```
  using H1Space = FEFunctionSpace<ContinuousLagrangeMapper<double,
      LeafView> >;
  using Spaces = boost::fusion::vector<H1Space const*>;
  using VariableDescriptions = boost::fusion::vector<Variable<
      SpaceIndex<0>,Components<1>,VariableId<0> > >;
  using VariableSetDesc = VariableSetDescription<Spaces,
      VariableDescriptions>;
  using Functional = HeatFunctional<double,VariableSetDesc>;
  using Assembler = VariationalFunctionalAssembler<LinearizationAt
      <Functional> >;
  using Operator = AssembledGalerkinOperator<Assembler>;
  using CoefficientVectors = VariableSetDesc::
      CoefficientVectorRepresentation<0,1>::type;

  ...
}
```

sets some parameters and *using* statements and comprises the following essential parts:

- definition of a triangulation of the region $\Omega$

  ```
  GridManager<Grid> gridManager( createUnitSquare<Grid>() );
  gridManager.globalRefine(refinements);
  ```

  The parameter *refinements* defined in top of the main program determines how often the coarse grid defined here has to be refined uniformly. The resulting mesh is the initial one for the following computation. In context of adaptive mesh refinement it might be object of further refinements, see example in subsection 6.5.

- definition of the finite element space

  ```
  // construction of finite element space for the scalar
      solution u.
  H1Space temperatureSpace(gridManager,gridManager.grid().
      leafView(),order);
  Spaces spaces(&temperatureSpace);
  VariableSetDesc variableSetDesc(spaces,{ "u" });
  ```

  Here we define the finite element space underlying the discretization of our equation (7) corresponding to the introduction in Section 2.1. The parameter *order* defined in top of the main program specifies the order of the finite element space in the statement

```
H1Space temperatureSpace(gridManager, gridManager.grid().
    leafView(), order);
```

- definition of the variational functional

```
Functional F;
```

The code defining the functional can be found in the file *laplace.hh*. The corresponding class is called *HeatFunctional* and contains the mandatory members *DomainCache* and *BoundaryCache* each of them specifying the functions $d0()$, $d1()$, and $d2()$ described above. The static member template class $D2$ defines which Hessian blocks are available what is of major interest in case of systems of equations, here we only have one block. $D1$ provides information about the structure of the right-hand side, e.g., if it is non-zero (present = true, i.e. it is present) or if it is not constant (constant = false). In the member function *integrationOrder* the order of the integration formula used in the assembling is specified.

```
template <class RType, class VarSet>
class HeatFunctional : public FunctionalBase<
    VariationalFunctional>
{
public:
  using Scalar = RType;
  using OriginVars = VarSet;
  using AnsatzVars = VarSet;
  using TestVars = VarSet;

  class DomainCache : public CacheBase<HeatFunctional,
      DomainCache>
  {
    ...
  }

  class BoundaryCache : public CacheBase<HeatFunctional,
      BoundaryCache>
  {
    ...
  }

  template <int row>
  struct D1 : public FunctionalBase<VariationalFunctional>::
      D1<row>
  {
```

```
    static bool const present   = true;
    static bool const constant  = false;
  };


  template <int row, int col>
  struct D2 : public FunctionalBase<VariationalFunctional>::
      D2<row,col>
  {
    static bool const present = true;
    static bool const symmetric = true;
    static bool const lumped = false;
  };


  template <class Cell>
  int integrationOrder(Cell const& /* cell */,
                       int shapeFunctionOrder, bool boundary)
                           const
  {
    if (boundary)
      return 2*shapeFunctionOrder;
    else
    {
      int stiffnessMatrixIntegrationOrder = 2*(
          shapeFunctionOrder-1);
      int sourceTermIntegrationOrder = shapeFunctionOrder;
          // as rhs f is constant, i.e. of




      return std::max(stiffnessMatrixIntegrationOrder,
          sourceTermIntegrationOrder);
    }
  }
};
```

The DomaineCache provides member functions $d0$, $d1\_impl$, and $d2\_impl$ evaluating $F(\cdot)$, $F'(\cdot)\varphi_i$, $F''(\cdot)[\varphi_i,\varphi_j]$, respectively. The function $u$ is specified on construction of the caches, and can be evaluated for each quadrature point in the member function *evaluatedAt()* using the appropriate one among the evaluators provided by the assembler

```
  class DomainCache : public CacheBase<HeatFunctional,
      DomainCache>
```

```
{
public:
  DomainCache(HeatFunctional const&,
              typename AnsatzVars::Representation const&
                  vars_,
              int flags=7):
    data(vars_)
  {}


  template <class Position, class Evaluators>
  void evaluateAt(Position const& x, Evaluators const&
      evaluators)
  {
    u  = component<uIdx>(data).value(boost::fusion::at_c<
        uSpaceIdx>(evaluators));
    du = component<uIdx>(data).derivative(boost::fusion::
        at_c<uSpaceIdx>(evaluators));
    f = 1.0;
  }

  Scalar
  d0() const
  {
    return sp(du,du)/2 - f*u;
  }

  template<int row>
  Scalar d1_impl (VariationalArg<Scalar,dim,TestVars::
      template Components<row>::m> const& arg) const
  {
    return sp(du,arg.derivative) - f*arg.value;
  }

  template<int row, int col>
  Scalar d2_impl (VariationalArg<Scalar,dim,TestVars::
      template Components<row>::m> const &argTest,
                  VariationalArg<Scalar,dim,AnsatzVars::
                      template Components<row>::m> const
                      &argAnsatz) const
  {
    return sp(argTest.derivative,argAnsatz.derivative);
  }

private:
  typename AnsatzVars::Representation const& data;
  Dune::FieldVector<Scalar,AnsatzVars::template
      Components<uIdx>::m> u, f;
  Dune::FieldMatrix<Scalar,AnsatzVars::template
```

```
        Components<uIdx>::m, dim> du;
    LinAlg::EuclideanScalarProduct sp;
  };
```

The *BoundaryCache* is defined analogously. More details are presented in the next examples. We now continue with the *laplace.cpp* file.

- assembling and solution of a linear system

```
//construct Galerkin representation
Assembler assembler(spaces);
VariableSetDesc::Representation u(variableSetDesc);
assembler.assemble(linearization(F,u));

Operator A(assembler);
CoefficientVectors solution(VariableSetDesc::
    CoefficientVectorRepresentation<>::init(spaces));
CoefficientVectors rhs(assembler.rhs());

directInverseOperator(A).applyscaleadd(-1.0,rhs,solution);
component<0>(u) = component<0>(solution);
```

The *assembler* is the kernel of each finite element program. It evaluates the integrals of the weak formulation based on the member functions of the Heat-Functional class and the finite element element space defined in the *H1Space* from above, of course closely aligned to the *Grid*.

- output for graphical device and end of program

```
writeVTK(u,"temperature",IoOptions().setOrder(order).
    setPrecision(7));

std::cout << "graphical output finished, data in VTK format
     is written into file temperature.vtu \n";
std::cout << "End Laplacian tutorial program" << std::endl;
```

KASKADE7 offers two formats to specify output of mesh and solution data for graphical devices. The VTK format is used in this example and can be visualized by *Paraview* software, for example. In particular for 3D geometries the format of the visualization package *amira* is recommended.

**Exercises**:

1. Use the parameter *refinements* to generate grids of different refinement levels. Write the grid and solution data into a file and compare the results by a visualization tool, e.g., *Paraview*.

2. In the example above we use quadratic finite elements of Lagrange type (i.e., *order* = 2). Consider increase of the number of degrees of freedom for *order* = 1,2,3,...

3. Change the value of $f$ in the header file (i.e. in the *evaluateAt* member function) and observe the changes in the output. (If there is no visible change, also look at the color legend.)

**Remark**: This first example has been updated in March 2015 (to which this tutorial is adjusted). The following examples have not been updated so far which leads to discontinuities in coding/notation (e.g. *typedef* instead of *using*). In general the coding style of this example is to be followed.

## 6.2 Stationary heat transfer

This example is implemented in subdirectory

KASKADE7/tutorial/stationary_heattransfer

Files: **ht.cpp**, **ht.hh**, **Makefile**
We consider another stationary heat transfer problem but with some extensions compared to the preceding example:

- the region $\Omega$ may be two- or three-dimensional,
- the diffusion coefficient $\kappa$ may depend on $x \in \Omega$,
- the operator may include a mass term with coefficient $q(x), x \in \Omega$,
- the heat source (right-hand side of heat transfer equation) may depend on $x \in \Omega$,
- the user may select between direct and iterative solution of the linear systems,
- the user gets support to handle parameters.

The corresponding scalar partial differential equation is still linear and given by

$$\begin{aligned} -\nabla \cdot (\kappa(x)\nabla u(x)) + q(x)u(x) &= f(x) && \text{x } \in \Omega \\ \kappa \frac{\partial u}{\partial n}(x) &= 0 && \text{on } \Gamma_1 \\ u(x) &= u_b(x) && \text{on } \Gamma_2 \end{aligned} \tag{14}$$

$\Gamma_1$ and $\Gamma_2$ denote parts of the boundary $\partial\Omega = \Gamma_1 \dot\cup \Gamma_2$ where we have to define boundary conditions. On $\Gamma_1$ we assume a homogeneous Neumann and on $\Gamma_2$ we prescribe the values of the solution $u$ (Dirichlet condition).

In KASKADE7 , the solution of this problem is calculated via the Finite Element Method (FEM). Therefore it is necessary to consider the system (14) in its weak formulation, see appendix A.1. In particular, we want to treat the problem in its minimization formulation as in the example before and as described in Section 2.2 for the general case:

Find the solution $u$ of the minimization problem (in the finite element space) as described in (1) using

$$F(v) = \frac{1}{2}(\kappa \nabla v^T \nabla v + qvv) - fv$$

and

$$G(v) = \frac{1}{2}(v - u_0)^2, \quad u_0 = u_b(x)$$

We remark that the homogeneous Neumann boundary conditions provide no contribution in the weak formulation.

### 6.2.1 A walk through the main program

For solving the problem we have to do both defining the attributes of the equations (14) and defining the details of the method. In order to keep the example simple we choose a constant diffusion coefficient $\kappa = 1$, a constant mass coefficient $q = 1$. The right-hand side $f$ is determined so that $u = x(x-1)exp(-(x-0.5)^2)$ is solution of equation (14) for all $(x,y,z) \in \Omega$. The domain $\Omega$ is the square unit or unit cube respectively. On the boundary we have $u_b = 0$ for $x = 0$ and $x = 1$, elsewhere homogeneous Neumann boundary conditions.

**Preliminaries.** We write some important parameters in the top of the main program, e.g., *order* defining the order of the finite element ansatz, or *refinements* specifying the number of refinements of the initial grid, or `verbosity` for selecting the level (possible values 0,1,2,3) of verbosity of certain functions (e.g. iterative solver).

```
int verbosityOpt = 1;
bool dump = true;
std::unique_ptr<boost::property_tree::ptree> pt =
getKaskadeOptions(argc, argv, verbosityOpt, dump);

int refinements = getParameter(pt, "refinements", 5),
```

```
    order   = getParameter(pt, "order", 2),
    verbosity  = getParameter(pt, "verbosity", 1);
std::cout << "original mesh shall be refined : " << refinements <<
     " times" << std::endl;
std::cout << "discretization order : " << order << std::endl;
std::cout << "output level (verbosity): " << verbosity << std::
    endl;

int  direct, onlyLowerTriangle = false;

DirectType directType;
MatrixProperties property = SYMMETRIC;
PrecondType precondType = NONE;
std::string empty;
```

In this example we use the function *getKaskadeOptions* to create a *property_tree pt* for storing a set of parameters in tree structure. This tree is filled by predefinitions in a file *default.json* and by arguments in *argc*, *argv*. Based on this *property_tree* the call of $getParameter(..,s,..)$ provides a value corresponding to the string $s$. The result might be another string or a value of any other type, e.g. *integer* or *double*. In particular, a parameter may be specified in the input line via the *argc*, *argv* when starting the executable, e.g.,

- Let's have the following call of the executable `heat`

  `./heat --order 1`

  then call of getParameter(...) in the statement

  `order = getParameter(pt, "order", 2);`

  assign the value 1 to the variable *order*.

- Let's start the executable without parameter list then the call of getParameter(...) in the statement

  `order = getParameter(pt, "order", 2);`

  checks for a string `order` in the file `Kaskade7/default.json`. If there is none the default value of the getParameter(...,2) is assigned to the variable *order*.

- Let's consider the following call of the executable

  `./heat --solver.type direct`

  then call of getParameter(...) in the statement

  `getParameter(pt, "solver.type", empty);`

  reveals the string "direct" as return value which is used to build a string $s$ by

```
std::string s("names.type.");
s += getParameter(pt, "solver.type", empty);
```

As result we get "names.type.direct" as value of *s*. The call of

```
int direct = getParameter(pt, s, 0);
```

looks for the value of the string *s* in the file `default.json` and finds the value 1 assigning it to the integer variable `direct`.

Note that there are default values for *verbosityOpt* and *dump* in the parameter list of *getKaskadeOptions*(...). More details are given in Section 7.

**Defining the grid.** We define a two-dimensional grid on a square $\Omega = [0,1] \times [0,1]$ with two triangles and refine it `refinements` times. The grid is maintained using the UG library [3] which will allow adaptive refinement.

```
#if SPACEDIM==2
  //   two-dimensional space: dim=2
  constexpr int dim=2;
  using Grid = Dune::UGGrid<dim>;
  GridManager<Grid> gridManager( createUnitSquare<Grid>() );
  gridManager.globalRefine(refinements);
  std::cout << std::endl << "Grid: " << gridManager.grid().size(0)
       << " triangles, " << std::endl;
#else
  //  three-dimensional space: dim=3
  constexpr int dim=3;
  using Grid = Dune::UGGrid<dim>;
  GridManager<Grid> gridManager( createUnitCube<Grid>(0.5) );
  gridManager.globalRefine(refinements);
  std::cout << std::endl << "Grid: " << gridManager.grid().size(0)
       << " tetrahedra, " << std::endl;
  std::cout << "       " << gridManager.grid().size(1) << "
      triangles, " << std::endl;
#endif
  std::cout << "       " << gridManager.grid().size(dim-1) << "
      edges, " << std::endl;
  std::cout << "       " << gridManager.grid().size(dim) << "
      points" << std::endl;
```

Here we use as grid type `Dune::UGGrid<dim>`.

**Function spaces and variable sets.** We use quadratic continuous Lagrange finite elements for discretization on the mesh constructed before. The defined `variableSet`

contains the description of out finite element space

```
using LeafView = Grid::LeafGridView;
using H1Space = FEFunctionSpace<ContinuousLagrangeMapper<double,
    LeafView> >;
// using H1Space = FEFunctionSpace<ContinuousHierarchicMapper<
    double,LeafView> >;
using Spaces = boost::fusion::vector<H1Space const*>;
using VariableDescriptions = boost::fusion::vector<Variable<
    SpaceIndex<0>,Components<1>,VariableId<0> > >;
using VariableSet = VariableSetDescription<Spaces,
    VariableDescriptions>;
using Functional = HeatFunctional<double,VariableSet>;
using Assembler = VariationalFunctionalAssembler<LinearizationAt
   <Functional> >;
constexpr int neq = Functional::TestVars::noOfVariables;
using CoefficientVectors = VariableSet::
    CoefficientVectorRepresentation<0,neq>::type;
using LinearSpace = VariableSet::CoefficientVectorRepresentation
    <0,neq>::type;

// construction of finite element space for the scalar solution
    T.
H1Space temperatureSpace(gridManager,gridManager.grid().leafView
    (),order);

Spaces spaces(&temperatureSpace);

// construct variable list.
// VariableDescription<int spaceId, int components, int Id>
// spaceId: number of associated FEFunctionSpace
// components: number of components in this variable
// Id: number of this variable

std::string varNames[1] = { "u" };

VariableSet variableSet(spaces,varNames);
```

A finite element space `H1Space` is constructed with Lagrange elements of order 2 on our grid or to be more precise on the set of leaves of our grid, the `leafIndexSet`.

Due to generality we administrate the finite element space in a vector `Spaces` of spaces though in case of a scalar heat transfer we only have one equation, i.e., the vector `spaces` has only one component, the `temperatureSpace`. We use the boost fusion vector type to get a container which may hold different variable

sets e.g. linear and quadratic element descriptions.

The template parameters for the class `Variable` are

- `SpaceIndex<>` the number of associated FEFunctionSpace (0, we have only the H1Space in our example),
- `Components<>` the number of components in this variable (1, the temperature), and
- `VariableId<>` the number of this variable (0)

in arbitrary order. Alternatively, the class `VariableDescription` can be used, where the respective numbers can be specified directly in the correct order.

**Using functionals.** `HeatFunctional` denotes the functional to be minimized. An object of this class using the material parameters $\kappa$ and $q$ is constructed by

```
Functional F(kappa,q);
```

The implementation will be discussed in Section **??**. The Galerkin operator types `Assembler` and `Rhs` are defined. The two variables `u` and `du` will hold the solution, respectively a Newton correction. (Here we need only one Newton step because the is linear in $u$.) The corresponding code:

```
// construct vatiational functional
typedef HeatFunctional<double,VariableSet> Functional;
double kappa = 1.0;
double q = 1.0;
Functional F(kappa,q);
// ... print out number of variables, equations and degrees of
    freedom ...
//construct Galerkin representation
typedef VariationalFunctionalAssembler<LinearizationAt<
    Functional>> Assembler;

typedef VariableSet::CoefficientVectorRepresentation<0,1>::type
     CoefficientVectors;
Assembler assembler(gridManager,spaces);
VariableSet::Representation u(variableSet);
VariableSet::Representation du(variableSet);
```

**Assembling.**

```
property = SYMMETRIC;
if ( (directType == DirectType::MUMPS)||(directType == DirectType
    ::PARDISO) || (precondType == PrecondType::ICC) )
{
  onlyLowerTriangle = true;
  std::cout <<
    "Note: direct solver MUMPS/PARADISO or ICC preconditioner ===>
        onlyLowerTriangle is set to true!"
     << std::endl;
}
```

The finite element discretization in this example leads to a symmetric linear system which we have to assemble and to solve. Therefore, we should assign SYMMETRIC to the variable *property* describing the property of the matrix in order to save computing time where possible. If the matrix is symmetric some solver offer the possibility to work only on the lower triangle, e.g., the direct solver MUMPS and PARADISO, or the incomplete Cholesky preconditioner ICC. In these cases we may (in case of ICC we even have to) set the parameter `onlyLowerTriangle` used when providing the matrix from the Galerkin operator, see below.

```
  size_t nnz = assembler.nnz(0,1,0,1,onlyLowerTriangle);
  std::cout << "number of nonzero elements in the stiffness
      matrix: " << nnz << std::endl << std::endl;

 boost::timer::cpu_timer assembTimer;
  CoefficientVectors solution(VariableSet::
      CoefficientVectorRepresentation<0,1>::init(variableSet));
  solution = 0;

  // UG seems to admit concurrent reads while claiming not to be
      thread safe.
  // In this case we enforce multithreading during assembly.
  gridManager.enforceConcurrentReads(std::is_same<Grid,Dune::
      UGGrid<dim> >::value);
  assembler.setNSimultaneousBlocks(blocks);
  assembler.setRowBlockFactor(rowBlockFactor);
  assembler.assemble(linearization(F,u),assembler.MATRIX|
      assembler.RHS|assembler.VALUE,nthreads,
```

```
  CoefficientVectors rhs(assembler.rhs());
  AssembledGalerkinOperator<Assembler,0,1,0,1> A(assembler,
      onlyLowerTriangle);
std::cout << "computing time for assemble: " << boost::timer::
    format(assembTimer.elapsed()) << "\n";
```

**Solving.** A direct method (e.g., third-party software MUMPS or UMFPACK ) or an iterative solver ( e.g., the cg or the bicgstab method with a suitable preconditioner) may be used for solving the assembled linear system. The variable *property* describing some matrix property should be set to SYMMETRIC. The parameter *onlyLowerTriangle* used for constructing the matrix has to be choosen carefully because not every solver or preconditioner works correctly if *onlyLowerTriangle* is set to true. In particular, preconditioners made for nonsymmetric problem expect that $onlyLowerTriangle = false$.

```
if (direct)
{
  boost::timer::cpu_timer directTimer;
  directInverseOperator(A,directType,property).applyscaleadd(-1.0,
      rhs,solution);
  u.data = solution.data;
  std::cout << "computing time for direct solve: " << boost::timer
      ::format(directTimer.elapsed()) << "\n";
}
else
{
  boost::timer::cpu_timer iteTimer;
  int iteSteps = getParameter(pt, "solver.iteMax", 1000);
  double iteEps = getParameter(pt, "solver.iteEps", 1.0e-10);
  typedef VariableSet::CoefficientVectorRepresentation<0,1>::type
      LinearSpace;
  Dune::InverseOperatorResult res;

  switch (precondType)
  {
    case NONE:
    {
      TrivialPreconditioner<AssembledGalerkinOperator<Assembler
          ,0,1,0,1> > trivial;
      Dune::CGSolver<LinearSpace> cg(A,trivial,iteEps,iteSteps,
          verbosity);
      cg.apply(solution,rhs,res);
    }
    break;
    case ADDITIVESCHWARZ:
```

```
  {
    std::cout << "selected preconditioner: ADDITIVESCHWARZ" <<
        std::endl;
    std::pair<size_t,size_t> idx = temperatureSpace.mapper().
        globalIndexRange(gridManager.grid().leafIndexSet().
        geomTypes(dim)[0]);
    AdditiveSchwarzPreconditioner<AssembledGalerkinOperator<
        Assembler,0,1,0,1> > addschwarz(A,idx.first,idx.second,
        verbosity);
    Dune::CGSolver<LinearSpace> cg(A,addschwarz,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case ILUT:
  {
    int lfil = getParameter(pt, "solver.ILUT.lfil", 140);
    double dropTol = getParameter(pt, "solver.ILUT,dropTol",
        0.01);
    ILUTPreconditioner<AssembledGalerkinOperator<Assembler
        ,0,1,0,1> > ilut(A,lfil,dropTol);
    Dune::BiCGSTABSolver<LinearSpace> cg(A,ilut,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case ILUK:
  {
    int fill_lev = getParameter(pt, "solver.ILUK.fill_lev", 3);
    ILUKPreconditioner<AssembledGalerkinOperator<Assembler
        ,0,1,0,1> >
    iluk(A,fill_lev,verbosity);
    Dune::BiCGSTABSolver<LinearSpace> cg(A,iluk,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case ICC:
  {
    std::cout << "selected preconditioner: ICC" << std::endl;
    if (property != SYMMETRIC)
    {
      std::cout << "ICC preconditioner of TAUCS lib has to be
          used with matrix.property==SYMMETRIC\n";
      std::cout << "i.e., call the executable with option --
          solver.property SYMMETRIC\n\n";
    }
    double dropTol = getParameter(pt, "solver.ICC.dropTol",
        0.01);
```

```
    ICCPreconditioner<AssembledGalerkinOperator<Assembler
        ,0,1,0,1> > icc(A,dropTol);
    Dune::CGSolver<LinearSpace> cg(A,icc,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case ICC0:
  {
    std::cout << "selected preconditioner: ICC0" << std::endl;
    ICC_0Preconditioner<AssembledGalerkinOperator<Assembler
        ,0,1,0,1> > icc0(A);
    Dune::CGSolver<LinearSpace> cg(A,icc0,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case HB:
  {
    std::cout << "selected preconditioner: HB" << std::endl;
    HierarchicalBasisPreconditioner<Grid,
        AssembledGalerkinOperator<Assembler,0,1,0,1>::range_type
        ,
     AssembledGalerkinOperator<Assembler,0,1,0,1>::range_type >
         hb(gridManager.grid());
    Dune::CGSolver<LinearSpace> cg(A,hb,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case ARMS:
  {
    int lfil = getParameter(pt, "solver.ARMS.lfil", 140);
    int lev_reord = getParameter(pt, "solver.ARMS.lev_reord", 1)
        ;
    double dropTol = getParameter(pt, "solver.ARMS.dropTol",
        0.01);
    double tolind = getParameter(pt, "solver.ARMS.tolind", 0.2);
    ARMSPreconditioner<AssembledGalerkinOperator<Assembler
        ,0,1,0,1> > iluk(A,lfil,dropTol,lev_reord,tolind,
        verbosity);
    Dune::CGSolver<LinearSpace> cg(A,iluk,iteEps,iteSteps,
        verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case BOOMERAMG:
  {
    int steps = getParameter(pt, "solver.BOOMERAMG.steps",
```

```
        iteSteps);
    int coarsentype = getParameter(pt, "solver.BOOMERAMG.
        coarsentype", 21);
    int interpoltype = getParameter(pt, "solver.BOOMERAMG.
        interpoltype", 0);
    int cycleType = getParameter(pt, "solver.BOOMERAMG.cycleType
        ", 1);
    int relaxType = getParameter(pt, "solver.BOOMERAMG.relaxType
        ", 3);
    int variant = getParameter(pt, "solver.BOOMERAMG.variant",
        0);
    int overlap = getParameter(pt, "solver.BOOMERAMG.overlap",
        1);
    double tol = getParameter(pt, "solver.BOOMERAMG.tol", iteEps
        );
    double strongThreshold = getParameter(pt, "solver.BOOMERAMG.
        strongThreshold",dim==2)?0.25:0.6);
    BoomerAMG<AssembledGalerkinOperator<Assembler,0,1,0,1> >
        BoomerAMGPrecon(A, steps, coarsentype, interpoltype, tol,
        cycleType, relaxType, strongThreshold, variant, overlap,1,
        verbosity);
    Dune::LoopSolver<LinearSpace> cg(A,BoomerAMGPrecon,iteEps,
        iteSteps, verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  // ...
  case EUCLID:
  {
    std::cout << "selected preconditioner: EUCLID" << std::endl;
    int level      = getParameter(pt, "solver.EUCLID.level",1);
    double droptol = getParameter(pt, "solver.EUCLID.droptol"
        ,0.01);
    int printlevel = 0;
    if (verbosity>2) printlevel=verbosity-2;
    printlevel = getParameter(pt,"solver.EUCLID.printlevel",
        printlevel);
    int bj = getParameter(pt, "solver.EUCLID.bj",0);
    Euclid<AssembledGalerkinOperator<Assembler,0,1,0,1> >
        EuclidPrecon(A, level, droptol, printlevel, bj, verbosity);
    Dune::CGSolver<LinearSpace> cg(A, EuclidPrecon, iteEps,
        iteSteps, verbosity);
    cg.apply(solution,rhs,res);
  }
  break;
  case JACOBI:
  default:
  {
    std::cout << "selected preconditioner: JACOBI" << std::endl;
```

```
      JacobiPreconditioner<AssembledGalerkinOperator<Assembler
          ,0,1,0,1> > jacobi(A,1.0);
      Dune::CGSolver<LinearSpace> cg(A,jacobi,iteEps,iteSteps,
          verbosity);
      cg.apply(solution,rhs,res);
    }
    break;
  }

  solution *= -1.0;
  u.data = solution.data;
  std::cout << "iterative solve eps= " << iteEps << ": "
            << (res.converged?"converged":"failed") << " after "
            << res.iterations << " steps, rate="
            << res.conv_rate << ", computing time=" << (double)(
                iteTimer.elapsed().user)/1e9 << "s\n";
}
```

**Output.**

```
// output of solution in VTK format for visualization,
// the data are written as ascii stream into file temperature.vtu,
// possible is also binary

writeVTK(u,"temperature",IoOptions().setOrder(order).setPrecision
    (7));

// output of solution for Amira visualization,
// the data are written in binary format into file temperature.am,
// possible is also ascii

// IoOptions options;
// options.outputType = IoOptions::ascii;
// LeafView leafView = gridManager.grid().leafView();
// writeAMIRAFile(leafView,variableSet,u,"temperature",options);
```

Default values (ascii format) will be set if `options` are omitted in the parameter list.

You may also produce instant graphical output, using the *GnuplotWriter*. Usage of this type of output needs that the program *gnuplot* is installed on the machine where you run your KASKADE7 application, and X11 is installed on the machine which controls your display. For using the *GnuplotWriter*, add in the headers-section of your program the line

```
#include "io/gnuplot.hh"
```

At the point of the ht.cpp source where you wish to output the solution, e.g. near the VTK output, insert the following lines:

```
  IoOptions gnuplotOptions{};
//     gnuplotOptions.info = IoOptions::none; // or IoOptions::
    summary or IoOptions::detail
// or use the two lines below to be able to set gnuplotOptions.
    info using the parameter --gnuplotinfo:
// s = "names.gnuplotinfo." + getParameter(pt, "gnuplotinfo",
    empty);
// gnuplotOptions.info = static_cast<IoOptions::Info>(getParameter
    (pt, s, 0));
// the parameter --gnuplotinfo may be set to one of the following
    values: none or summary or detail
// LeafView leafView = gridManager.grid().leafView();
writeGnuplotFile(u,"temperature",gnuplotOptions);
```

Set the environment variable `DISPLAY` to *your-workstation-name*:0, and run on your workstation the command

```
xhost +htcnnn
# substitute nnn by the proper three digits, for example 026 for htc026
```

to permit htcnnn to send X11-output to your workstation.
Furthermore, in the above example, you have also to supply a file named `temperature.gnu` with appropriate *gnuplot* commands, like the following:

```
set terminal x11
set dgrid3d 50,50 splines
set title "temperature"
set style line 1 lw 0
set pm3d implicit at s
splot "temperature.data" with dots
pause 15
set style fill solid
set pm3d map
splot "temperature.data"
pause 15
```

The above command makes *gnuplot* to display first a landscape graphics of the solution for 15 seconds and after this a colormap graphics for 15 seconds.
There is also a source `ht_gnuplot.cpp` available in the `stationary_heattransfer` directory, which already includes *gnuplot* output, and which may be used to build the program `heat_gnuplot` by just typing in

```
make heat_gnuplot
```

### 6.2.2 Defining the functional

The definition of the functional is the actual interface to a user who is not interested in algorithmic details. Here one has to specify the parameters of the problems, e.g., the material properties (in `DomainCache`) and the boundary conditions (in `BoundaryCache`).

Further informationabout the order of integration and D1 and D2 are expected.

**The template for the functional framework.**

```
template <class RType, class VarSet>
class HeatFunctional
{
    typedef HeatFunctional<RType,VarSet> Self;

  public:
    typedef RType  Scalar;
    typedef VarSet OriginVars;
    typedef VarSet AnsatzVars;
    typedef VarSet TestVars;
    static ProblemType const type = VariationalFunctional;

    class DomainCache
    { ... };

    class BoundaryCache
    { ... };

    template <int row> struct D1
    { ... };

    template <int row, int col> struct D2
    { ... };

    template <class cell>
    int integrationOrder(Cell const& cell, int shapeFunctionOrder,
                         bool boundary) const
    { ... };

  private:
   Scalar kappa, q;

}
```

**The domain cache.** As in the Laplacian problem in Section 6.1 we have to define $d1(v)$, $d2(v)$ evaluating $F'(\cdot)(v)$ and $F''(\cdot)(v,w)$ respectively. Using the notation from Section 2.2 we have in this example

$$F(u) = \frac{1}{2}(\kappa \nabla u^T \nabla u + q u^2) - fu$$

and

$$G(u) = \frac{1}{2}(u - u_b)^2, \quad u_b = 0$$

In the domain $\Omega$ we have

$$d0() = \frac{1}{2}(\kappa \nabla u^T \nabla u + q u^2) - fu \tag{15}$$

$$d1(\varphi_i) = \kappa \nabla u^T \nabla \varphi_i + qu\varphi_i - f\varphi_i \tag{16}$$

$$d2(\varphi_i, \varphi_j) = \kappa \nabla \varphi_i^T \nabla \varphi_j + q\varphi_i\varphi_j \tag{17}$$

```
class DomainCache
{
  public:
    DomainCache(Self const& F_,
                typename AnsatzVars::Representation const& vars_,
                int flags=7): F(F_), data(vars_)
    {}

    template <class Entity>
    void moveTo(Entity const &entity) { e = &entity; }

    template <class Position, class Evaluators>
    void evaluateAt(Position const& x, Evaluators const&
        evaluators)
    {
      using namespace boost::fusion;
      int const uIdx = result_of::value_at_c<typename AnsatzVars::
          Variables,0>::type::spaceIndex;
      xglob = e->geometry().global(x);

      u  = component<0>(data).value(at_c<uIdx>(evaluators));
      du = component<0>(data).derivative(at_c<uIdx>(evaluators))
          [0];

      double v, w, vX, vXX, wX, wXX, uXX;
      v = xglob[0]*(xglob[0] - 1);
      w = exp(-(xglob[0] - 0.5)*(xglob[0] - 0.5));

      vX  =  2*xglob[0] - 1;
```

```
      vXX =  2;
      wX  = -2*(xglob[0] - 0.5)*w;
      wXX = -2*w - 2*(xglob[0] - 0.5)*wX;

      uXX = vXX*w + 2*vX*wX + v*wXX;
      f = -F.kappa*uXX + F.q*v*w;
    }
    Scalar d0() const
    {
      return (F.kappa*du*du + F.q*u*u)/2 - f*u;;
    }

    template<int row, int dim>
    Dune::FieldVector<Scalar, TestVars::template Components<row>::
        m>
    d1 (VariationalArg<Scalar,dim> const& arg) const
    {
      return du*arg.derivative[0] + F.q*u*arg.value - f*arg.value;
    }

    template<int row, int col, int dim>
    Dune::FieldMatrix<Scalar, TestVars::template Components<row>::
        m, AnsatzVars::template Components<col>::m>
    d2 (VariationalArg<Scalar,dim> const &argTest, VariationalArg<
        Scalar,dim> const &argAnsatz) const
    {
      return argTest.derivative[0]*argAnsatz.derivative[0] + F.q*
          argTest.value*argAnsatz.value;
    }

  private:
    Self const& F;
    typename AnsatzVars::Representation const& data;
    typename AnsatzVars::Grid::template Codim<0>::Entity const* e;
    Dune::FieldVector<typename AnsatzVars::Grid::ctype, AnsatzVars
        ::Grid::dimension> xglob;
    Scalar f;
    Scalar u;
    Dune::FieldVector<Scalar, AnsatzVars::Grid::dimension> du;
};
```

**The boundary cache.** Analogously to example 6.1, we determine $d1(v)$, $d2(v)$ as $G'(\cdot)(v)$ and $G''(\cdot)(v,w)$ respectively. These functions deliver value 0 on those parts of the boundary where we have homogeneous Neumann condition.

```
class BoundaryCache
```

```
{
  public:
    static const bool hasInteriorFaces = false;
    typedef typename AnsatzVars::Grid::template Codim<0>::Entity::
        LeafIntersectionIterator FaceIterator;

    BoundaryCache(Self const&, typename AnsatzVars::Representation
         const& vars_, int flags=7): data(vars_), e(0)
    {}

    void moveTo(FaceIterator const& entity)
    {
      e = &entity;
      penalty = 1.0e9;
    }

    template <class Evaluators>
    void evaluateAt(Dune::FieldVector<typename AnsatzVars::Grid::
        ctype, AnsatzVars::Grid::dimension-1> const& x, Evaluators
         const& evaluators)
    {
      using namespace boost::fusion;

      int const uIdx = result_of::value_at_c<typename AnsatzVars::
          Variables,0>::type::spaceIndex;
      xglob = (*e)->geometry().global(x);

      u = component<0>(data).value(at_c<uIdx>(evaluators));
      u0 = 0;
    }

    Scalar d0() const
    {
      return penalty*(u-u0)*(u-u0)/2;
    }

    template<int row, int dim>
    Dune::FieldVector<Scalar, TestVars::template Components<row>::
        m>
    d1 (VariationalArg<Scalar,dim> const& arg) const
    {
      if ( (xglob[0]<=1e-12) || (xglob[0]>=(1-1e-12)) )
        return penalty*(u-u0)*arg.value[0];
      else return 0.0;
    }

    template<int row, int col, int dim>
    Dune::FieldMatrix<Scalar, TestVars::template Components<row>::
        m, AnsatzVars::template Components<col>::m>
```

```
    d2 (VariationalArg<Scalar,dim> const &argTest, VariationalArg<
        Scalar,dim> const &argAnsatz) const
    {
       if ( (xglob[0]<=1e-12) || (xglob[0]>=(1-1e-12)) )
         return penalty*argTest.value*argAnsatz.value;
       else return 0.0;
   }

  private:
    typename AnsatzVars::Representation const& data;
    FaceIterator const* e;
    Scalar penalty, u, u0;
};
```

**The remainings.**

```
 // constructor
HeatFunctional(Scalar kappa_, Scalar q_): kappa(kappa_), q(q_)
{
}

// structure of right-hand side
template <int row> struct D1
{
  static bool const present   = true;
  static bool const constant  = false;
};

// structure of matrix
template <int row, int col> struct D2
{
  static bool const present = true;
  static bool const symmetric = true;
  static bool const lumped = false;
};

// accuracy of integration formulas
template <class Cell>
int integrationOrder(Cell const& cell, int shapeFunctionOrder,
    bool boundary) const
{
  if (boundary)
    return 2*shapeFunctionOrder;
  else
    return 2*shapeFunctionOrder-1;
}
```

**computing time.**

```
int main(int argc, char *argv[])
{
  using namespace boost::fusion;
  boost::timer::cpu_timer totalTimer;
    ...
  std::cout << "total computing time: " << (double)(totalTimer.
      elapsed().user)/1e9 << "s\n";
  std::cout << "End heat transfer tutorial program" << std::endl;
}
```

We use the class *boost::timer::cpu_timer* to measure the computing time. The statement

`boost::timer::cpu_timer totalTimer;`

defines and starts a clock of type *boost::timer::cpu_timer* with initial value equal to zero. The member function *totalTimer.elapsed()* of this class variable provides the (unformatted) time passed since starting.

**Exercises**:

- 1. Let the code run for the 2D- and a 3D-geometries as prepared in the example. Consider the generated graphical output in an appropriate visualization tool, e.g. *Paraview*.
- 2. Compute solutions for different parameters *refinement* and *order* using the dynamical parameter handling *getKaskadeOptions* (changeable options are: refinements, order, verbosity, solver.type, solver.preconditioner, blocks, threads, rowBlockFactor, solver.iteMax, solver.iteEps and others; look through the program to find more).
- 3. Change the static parameters of the problem ($\kappa$ and $q$) using the dynamical parameter handling *getKaskadeOptions* (i.e. include the getParameter option for those two).
- 4. Use the direct solver and different iterative methods for solving the linear system. Investigate the effect of different accuracy requirements `iteEps` in the iterative solvers.

## 6.3 Laplace on a circle area

This example is implemented in subdirectory

KASKADE7/tutorial/geomgrid

Files: **geomgrid.cpp**, **laplace.hh**, **Makefile**
We compute the solution $u$ of the Laplacian or Poisson equation in two space variables on a circle area.

$$\begin{aligned} -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} &= 1 \qquad (x,y) \in \Omega \\ u &= 0 \qquad \text{on } \Gamma \end{aligned} \tag{18}$$

on the two-dimensional closed circle area $\Omega = \{(x,y) \in \mathbb{R}^2 \mid x^2 + y^2 \leq \sqrt{2}\}$ with the origin as the midpoint and with a radius size of $\sqrt{2}$ under homogeneous boundary conditions on $\Gamma = \partial\Omega$. These equations may describe stationary heat transfer caused by a constant heat source (value 1 on the right-hand side) and constant temperature ($0^o C$) on the boundary, e.g. by cooling.

In the following, we take some looks on the main program in **geomgrid.cpp**, focussing only on the parts which significantly differ from the code we considered in the previous example programs.

**Defining the grid.** We are using the Dune::GeometryGrid template class with a Dune:UGGrid on a square $[-1,1]^2 \subset \mathbb{R}^2$ as a base grid, together with a transformation class SquareToCircle, which maps the points from the square to the circle with the origin as the midpoint and a radius of size $\sqrt{2}$. We are starting with the creation of a course grid on the square, consisting of the five points $(-1,-1),(1,-1),(1,1),(-1,1)$ and $(0,0)$. As there is in KASKADE7 no utility-routine available for just the creation of a start-grid without passing it unmodified to a gridmanager instance, we must define the start-grid step by step, using the Dune GridFactory. In the following, we create a GridFactory called factory, and tell the factory by calling the method insertVertex about the five grid-points mentioned before. The desired start-grid we wish to define, consists of five vertices and four triangles, with each triangle defined by its three corner vertices, which we tell the factory by calling the method insertElement. In the next step, we create from the resulting factory a grid. This grid is passed as the base grid to the Dune::GeometryGrid through the first parameter, together with the class SquareToCircle as the second parameter. The resulting GeometryGrid is immediately used to create a KASKADE7 gridmanager instance. Next, the GeometryGrid is refined using the gridmanagers `globalRefine` method.

```
constexpr int dim=2;
using Grid = Dune::UGGrid<dim>;
using GeoGrid = Dune::GeometryGrid<Grid, SquareToCircle>;
...
Dune::GridFactory<Grid> factory;
```

```
  // vertex coordinates v[0], v[1]
  Dune::FieldVector<double,dim> v;
  v[0]=-1; v[1]=-1; factory.insertVertex(v);
  v[0]=1; v[1]=-1; factory.insertVertex(v);
  v[0]=1; v[1]=1; factory.insertVertex(v);
  v[0]=-1; v[1]=1; factory.insertVertex(v);
  v[0]=0; v[1]=0; factory.insertVertex(v);
  // triangles defined by 3 vertex indices
  std::vector<unsigned int> vid(3);
  Dune::GeometryType gt(Dune::GeometryType::simplex,2);
  vid[0]=0; vid[1]=1; vid[2]=4; factory.insertElement(gt,vid);
  vid[0]=1; vid[1]=2; vid[2]=4; factory.insertElement(gt,vid);
  vid[0]=2; vid[1]=3; vid[2]=4; factory.insertElement(gt,vid);
  vid[0]=3; vid[1]=0; vid[2]=4; factory.insertElement(gt,vid);

  Grid* grid( factory.createGrid() );
  GridManager<GeoGrid> gridManager(new GeoGrid(grid,new
      SquareToCircle(radius)));
  gridManager.globalRefine(refinements);
```

We now take a look at the transformation class SquareToCircle:

```
class SquareToCircle: public Dune::AnalyticalCoordFunction< double
    , 2, 2, SquareToCircle >
{
  using Base = Dune::AnalyticalCoordFunction< double, 2, 2,
      SquareToCircle >;

  public:
  using DomainVector = Base::DomainVector;
  using RangeVector = Base::RangeVector;

  SquareToCircle ( double _radius=sqrt(2) ): radius(_radius) {}

  void evaluate ( const DomainVector &u, RangeVector &y ) const
  {
    double enorm = sqrt(u[0]*u[0]+u[1]*u[1]);
    if(enorm > 1.0e-5)
    {
      double radiusFactor = radius*std::max(std::fabs(u[0]),std::
          fabs(u[1]));
      double scaling = radiusFactor/enorm;
      y[ 0 ] = u[ 0 ]*scaling;
      y[ 1 ] = u[ 1 ]*scaling;
    }
    else
    {
      y[ 0 ] = u[ 0 ];
```

```
      y[ 1 ] = u[ 1 ];
    }
  }

  private:
  double radius;
};
```

The transformation is done in the method evaluate of the class, by a adapted scaling of the squares points to the target circle. To avoid the division by small numbers points, which are near the origin, are not scaled, but just copied.

**An alternative method to read command line parameters** In this example program, we use an alternative method to obtain parameters from the command line. The parameters on the command line may be specified in similar way as for the example program 6.2. However, they are read by the program using a different parsing scheme, based on the *boost::program_options* namespace, extended by a class named *options_description*. The advantage of using this programming interface is that with little effort you will get a useful parameter description when you call the program just with the *--help* option, and that this interface interface is easier to handle. The necessary code to read the parameters and to generate the help message is just the following:

```
  int refinements, order, verbosity;
  double radius;

  if (getKaskadeOptions(argc,argv,Options
    ("refinements",       refinements,        5,          "number
       of uniform grid refinements")
    ("order",             order,              2,          "finite
       element ansatz order")
    ("verbosity",         verbosity,          1,          "output
       level")
    ("radius",            radius,  std::sqrt(2),          "radius
       of the circle")))
    return 1;
```

For the specification of each command line parameter, a list of four program parameters, enclosed in parentheses, must be specified, in the following order: the name of the parameter on the command line as a string, the program variable where to read in the value of the parameter, a default value, and a parameter description. Note that, the parameters are just read here in one block, and therefore, this interface is not suitable for situations, where the parameters need to be read at multiple locations of the program, as in the example 6.2.

**The remaining program parts.** The whole program (with the parts already discussed before excluded) follows:

```
int main(int argc, char *argv[])
{
  std::cout << "Start GeometryGrid tutorial program" << std::endl;

  boost::timer::cpu_timer totalTimer;

  ...

  std::cout << "original mesh shall be refined : " << refinements
     << " times" << std::endl;
  std::cout << "discretization order           : " << order << std
     ::endl;
  std::cout << "output level (verbosity)       : " << verbosity <<
      std::endl;

  bool onlyLowerTriangle = false;

  ...

  using LeafView = GeoGrid::LeafGridView;
  using H1Space = FEFunctionSpace<ContinuousLagrangeMapper<double,
     LeafView> >;
  using Spaces = boost::fusion::vector<H1Space const*>;
  using VariableDescriptions = boost::fusion::vector<Variable<
     SpaceIndex<0>,Components<1>,VariableId<0> > >;
  using VariableSetDesc = VariableSetDescription<Spaces,
     VariableDescriptions>;
  using Functional = HeatFunctional<double,VariableSetDesc>;
  using Assembler = VariationalFunctionalAssembler<LinearizationAt
     <Functional> >;
  using Operator = AssembledGalerkinOperator<Assembler>;
  constexpr int neq = Functional::TestVars::noOfVariables;
  using CoefficientVectors = VariableSetDesc::
     CoefficientVectorRepresentation<0,neq>::type;

  ...

  // construction of finite element space for the scalar solution
     u
  H1Space temperatureSpace(gridManager,gridManager.grid().
     leafGridView(),order);
  Spaces spaces(&temperatureSpace);
  std::string varNames[1] = { "T" };
  VariableSetDesc variableSet(spaces,varNames);
```

```
  Functional F(radius);
  constexpr int nvars = Functional::AnsatzVars::noOfVariables;
  std::cout << std::endl << "no of variables = " << nvars << std::
      endl;
  std::cout << "no of equations = " << neq   << std::endl;
  size_t dofs = variableSet.degreesOfFreedom(0,nvars);
  std::cout << "number of degrees of freedom = " << dofs   << std
      ::endl;

 //construct Galerkin representation
  Assembler assembler(spaces);
  VariableSetDesc::VariableSet u(variableSet);

  size_t nnz = assembler.nnz(0,neq,0,nvars,onlyLowerTriangle);
  std::cout << "number of nonzero elements in the stiffness matrix
      : " << nnz << std::endl << std::endl;

  boost::timer::cpu_timer assembTimer;
  assembler.assemble(linearization(F,u));
  std::cout << "computing time for assemble: " << boost::timer::
      format(assembTimer.elapsed());

  Operator A(assembler);
  CoefficientVectors solution(VariableSetDesc::
      CoefficientVectorRepresentation<>::init(spaces));
  CoefficientVectors rhs(assembler.rhs());

  boost::timer::cpu_timer directTimer;
  directInverseOperator(A).applyscaleadd(-1.0,rhs,solution);
  std::cout << "computing time for direct solve: " << boost::timer
      ::format(directTimer.elapsed());
  component<0>(u) = component<0>(solution);

  L2Norm l2Norm;
  std::cout << "L2norm(solution) = " << l2Norm(boost::fusion::at_c
      <0>(u.data)) << std::endl;

  boost::timer::cpu_timer outputTimer;
  writeVTK(u,"temperature",IoOptions().setOrder(order).
      setPrecision(7));
  std::cout << "graphical output finished, data in VTK format is
      written into file temperature.vtu\n";
  std::cout << "computing time for output: " << boost::timer::
      format(outputTimer.elapsed()) << "\n";
  std::cout << "total computing time: " << boost::timer::format(
      totalTimer.elapsed());
  std::cout << "End GeometryGrid tutorial program" << std::endl;
}
```

**The functional.** The implementation of the functional - only the parts, which differ from the first Laplace example 6.1 discussed before in this manual, looks as follows:

```
HeatFunctional(Scalar radius_=std::sqrt(2)): radius(radius_) {}
```

```
class BoundaryCache
{
public:
  using FaceIterator = typename AnsatzVars::Grid::
      LeafIntersectionIterator;

  BoundaryCache(HeatFunctional<RType,AnsatzVars> const&,
                typename AnsatzVars::VariableSet const& vars_,
                int flags=7):
    data(vars_), penalty(1e9), u0(0.)
  {}

  void moveTo(FaceIterator const& entity)
  {
    e = &entity;
  }

  template <class Evaluators>
  void evaluateAt(Dune::FieldVector<typename AnsatzVars::Grid::
      ctype,
                                    AnsatzVars::Grid::dimension
                                      -1>
                  const& x, Evaluators const& evaluators)
  {
    using namespace boost::fusion;

    int const uIdx = result_of::value_at_c<typename AnsatzVars::
        Variables,
                                           0>::type::spaceIndex;
    xglob = (*e)->geometry().global(x);
    squareNorm = sqrt( xglob[0]*xglob[0]+xglob[1]*xglob[1] );

    u = component<0>(data).value(at_c<uIdx>(evaluators));
  }

  Scalar
  d0() const
  {
   if ( squareNorm >= radius )
     return penalty*(u-u0)*(u-u0)/2;
```

```
    else return 0.0;
  }

  template<int row, int dim>
  Dune::FieldVector<Scalar, TestVars::template Components<row>::
      m>
  d1 (VariationalArg<Scalar,dim> const& arg) const
  {
    if ( squareNorm >= radius )
      return penalty*(u-u0)*arg.value[0];
    else return 0.0;
  }

  template<int row, int col, int dim>
  Dune::FieldMatrix<Scalar, TestVars::template Components<row>::
      m,
                    AnsatzVars::template Components<col>::m>
  d2 (VariationalArg<Scalar,dim> const &arg1,
                     VariationalArg<Scalar,dim> const &arg2)
                         const
  {
    if ( squareNorm >= radius )
     return penalty*arg1.value*arg2.value;
    else return 0.0;
  }

private:
  typename AnsatzVars::VariableSet const& data;
  FaceIterator const* e;
  Dune::FieldVector<typename AnsatzVars::Grid::ctype,AnsatzVars
      ::Grid::dimension> xglob;
  Scalar penalty, u, u0, squareNorm, radius;
};
```

**Exercises**:

1. Determine the exact solution of the underlying Laplacian problem 6.3 for a prescribed arbitrary radius of the circle area. Use the parameter *radius* to solve the Laplacian problem on circles with different radiuses. Write the grid and solution data into a file and view them by a visualization tool, e.g., *Paraview*, and verify the correctness of the numerically computed solution.

2. a. Extend the example to solve the Laplacian problem on an ellipse area with a given long half-axis $a$ and a short half axis $b$. Implement and use the commandline parameters `--long` and `--short` for the half-axises $a$ and $b$.

b. Introduce an additional parameter `--angle`, which passes an angle in degrees to the program, and rotate the ellipse area by this angle.

## 6.4 Artificial Test Problem (atp)

This example is implemented in subdirectory

KASKADE7/tutorial/artificial_1d_testProblem.

Files: **atp.cpp**, **atp.hh**, **function.gnu**, **Makefile**

The example features the use of the use of KASKADE7 for solving an ODE boundary value problem. The example is an artificially generated scalar ODE for the function u: $\mathbb{R} \to \mathbb{R}$

$$u(x) = \exp(-x^2). \tag{19}$$

The equation is

$$\frac{d^2u}{dx^2} - (0.9\exp(-x^2) + 0.1u)(4x^2 - 2) + s \cdot g(x) = 0 \tag{20}$$

where

$$\begin{aligned} &a) \quad g(x) = \exp(u) - \exp(\exp(-x^2)) \\ &b) \quad s \in \{-1, 0, 1\}. \end{aligned} \tag{21}$$

The equation is solved in the interval $[-3,3]$ with the simplified boundary conditions

$$u(-3) = u(3) = 0\,. \tag{22}$$

The problem is nonlinear for $s = 1$ or $s = -1$, and linear for $s = 0$.
The initial values are

$$u^0(x) = 0 \;\text{ for }\; x \in [-3,3]\;. \tag{23}$$

In the following, we take some looks on the main program in **atp.cpp**, focussing only on the parts which significantly differ from the code we considered in the previous example program.

**Defining the grid.** In this example, we have to use a one-dimensional grid on the interval $[3,3]$. A 1D-grid implementation is included within Dune, implemented as the class `Dune::OneDGrid`. We are starting with a course grid, consisting of the three points -3,0,3, and refine it `refinements` times. As there is in KASKADE7 no utility-routine available for the creation of the start-grid, we must define the start-grid step by step, using the Dune GridFactory. In the following, we create a GridFactory called factory, and tell the factory by calling the method insertVertex about the three grid-points -3,0,3. The desired start-grid we wish to define, consists of the two elements, i.e. subintervals, $[-3,0]$ and $[0,3]$, which we we tell the factory by calling the method insertElement. In the next step, we create from the

result factory a grid, which will be after creation immediately refined using the `globalRefine` method. Finally, the resulting grid is moved under control of the KASKADE7 gridmanager.

```
//   one-dimensional space: dim=1
int const dim=1;
using Grid = Dune::OneDGrid;

Dune::GridFactory<Grid> factory;
...
// point (in case of dimension>1: vertex) coordinates v[0]
Dune::FieldVector<double,dim> v;
v[0]=-3; factory.insertVertex(v);
v[0]=0; factory.insertVertex(v);
v[0]=3; factory.insertVertex(v);
std::vector<unsigned int> vid(2);
Dune::GeometryType gt(Dune::GeometryType::simplex,dim);
vid[0]=0; vid[1]=1; factory.insertElement(gt,vid);
vid[0]=1; vid[1]=2; factory.insertElement(gt,vid);
// interval defined by 2 point indices
std::unique_ptr<Grid> grid( factory.createGrid() ) ;
// the coarse grid will be refined refinements times
grid->globalRefine(refinements);
// some information on the refined mesh
std::cout << "Grid: " << grid->size(1) << " points " << std::
    endl;
// a gridmanager is constructed
// as connector between geometric and algebraic information
GridManager<Grid> gridManager(std::move(grid));
```

**The remaining.** The functional class, which defines the atp-ODE and the boundary conditions, is called ATPFunctional, and the constructor of this class accept the parameter $s$ from (21) as the only argument, named `fsign` in the code below. For solution of the arising linear systems in the Newton-iteration loop, we use the direct linear solver UMFPACK, for which the matrix must be supplied in triplet-format, e.g. as two integer-arrays `ridx` and `cidx`, which hold the row- and column-indices and the double-array `data`, which holds the corresponding values of the matrix-elements. The line

```
MatrixAsTriplet<double> triplet(nnz);
```

reserves the needed space for the matrix nonzero elements, and the line

```
  triplet = A.get<MatrixAsTriplet<double> >();
```

copies to the reserved MatrixAsTriplet class space the assembled galerkin-operator matrix.

The lines below compute the LU-factorization of the matrix hold in triplet and copies the right hand side of the linear system to the `std::vector<double> rhs`.

```
Factorization<double> *matrix = 0;
matrix = new UMFFactorization<double>(size,0,triplet.ridx,
    triplet.cidx,triplet.data);
assembler.toSequence(0,neq,rhs.begin());
```

The line

```
matrix->solve(rhs,sol);
```

finally solves the linear system, storing the solution to the `std::vector<double> sol`.
In the remaining code the vector sol is converted to the

```
VariableSet::VariableSet dx(variableSet);
```

container class, and the Newton-correction is computed by multiplying with $-1$, and then is added to the iterate `VariableSet::VariableSet x`. Finally the norms of the Newton-correction `dx` and the norm of the residuum `rhs` are computed for print-out and convergence-checking. The computed iterates can be viewed using gnuplot, when the bool variable `graphicalOutput` is set to true - what is done by default.

```
...
using Functional = ATPFunctional<double,VariableSet>;
...
Functional F(fsign);
Assembler assembler(gridManager,spaces);
VariableSet::VariableSet x(variableSet);
VariableSet::VariableSet dx(variableSet);

// set nnz to the number of structural nonzero elements of the
    matrix to be assembled below
size_t  nnz = assembler.nnz(0,neq,0,nvars,false);
size_t  size = variableSet.degreesOfFreedom(0,nvars);
AssembledGalerkinOperator<Assembler,0,neq,0,nvars> A(assembler);
MatrixAsTriplet<double> triplet(nnz);

std::vector<double> rhs(size), sol(size);

int k=0;
L2Norm l2Norm;
double norm_dx, norm_rhs;
x=0;
```

```
  std::cout << std::endl << "Newton iteration starts:" << std::
      endl <<
            "iter    ||correction||            ||F||  assemble time
                 linsolve time"
            << std::endl;

// begin of ordinary Newton iteration loop
  do
  {
    boost::timer::cpu_timer assembTimer;
    assembler.assemble(linearization(F,x));
    double assembleTime = (double)(assembTimer.elapsed().user)/1e9
        ;
    triplet = A.get<MatrixAsTriplet<double> >();
    //      for (k=0; k< nnz; k++)
    //        {
    //          printf("%3d %3d %e\n", triplet.ridx[k], triplet.
        cidx[k], triplet.data[k]);
    //        }
    boost::timer::cpu_timer directTimer;
    Factorization<double> *matrix = 0;
    matrix = new UMFFactorization<double>(size,0,triplet.ridx,
        triplet.cidx,triplet.data);
    assembler.toSequence(0,neq,rhs.begin());
    for (int l=0; l<rhs.size(); ++l) assert(std::isfinite(rhs[l]))
        ;
    matrix->solve(rhs,sol);
    double solveTime = (double)(directTimer.elapsed().user)/1e9;
    delete matrix;
    for (int l=0; l<sol.size(); ++l) assert(std::isfinite(sol[l]))
        ;
    dx.read(sol.begin());
    dx *= -1;
    x += dx;
    norm_dx=l2Norm(component<0>(dx));
    VariableSet::VariableSet tmp(dx);
    tmp.read(rhs.begin());
    norm_rhs=l2Norm(component<0>(tmp));
    std::cout << std::setw(4) << k+1 << "  "
              << std::setw(15) << std::setprecision(5) << std::
                  scientific << norm_dx << "  "
              << std::setw(15) << norm_rhs << "  " << std::
                  setprecision(3);
    std::cout.unsetf(std::ios::fixed | std::ios::scientific);
    std::cout << std::fixed << std::setw(6) << "        " << std::
        setprecision(3)
              << assembleTime << "s         " << std::setw(6) <<
                  solveTime << "s " << std::endl;
    // output of solution in VTK format for visualization,
```

```
    // the data are written as ascii stream into file temperature.
        vtu,
    // possible is also binary
    // char fname[6];
    // IoOptions options;
    // options.outputType = IoOptions::ascii;
    // sprintf(fname,"atp_%#02d",k);
    // writeVTK(x,fname,IoOptions().setOrder(order));
    if ( graphicalOutput )
    {
      IoOptions gnuplotOptions;
      std::string s = "names.gnuplotinfo." + getParameter(pt, "
          gnuplotinfo", empty);
      gnuplotOptions.info = static_cast<IoOptions::Info>(
          getParameter(pt,s,0));
      writeGnuplotFile(x,"function",gnuplotOptions);
    };
     k ++;
  }
  while ( norm_dx > 1.0e-5 );
```

## 6.5 Using embedded error estimation

This example is implemented in subdirectory

KASKADE7/tutorial/Embedded_errorEstimation.

Files: **peaksource.cpp**, **peaksource.hh**, **Makefile**
We consider a simple stationary heat transfer problem in the two-dimensional region $\Omega$, similar to that in Section 6.2. The corresponding linear partial differential equation is given by

$$\begin{aligned} -\nabla\cdot(\nabla T) &= f(x) \qquad \text{x} \in \Omega \\ T &= T_b(x) \qquad \text{on } \partial\Omega \end{aligned} \tag{24}$$

which can be solved in two and three space dimensions. We determine the right-hand side $f$ of the equation so that

$$T = x(x-1.0)y(y-1.0)exp(-100((x-0.5)^2+(y-0.5)^2))$$

is the soluton of (24) in 2d. On the whole boundary $\partial\Omega$ we prescribe the values of $T$ (Dirichlet condition). In 3D, the equation reads analogously.

Equation (24) is treated in the same way as before the stationary heat transfer problem (6.2). However, we now control the discretization error by an error estimation. The estimator provides information about the global and the local error.

Local error estimates are used to refine the mesh where the estimated error exceeds a threshold. If the estimate of the global error indicates sufficient accuracy the sequence of *compute solution on a fixed mesh*, *estimate the discretization error*, and *refine* stops.

```
bool accurate = true;
do {
   refSteps++;

   boost::timer::cpu_timer assembTimer;
   VariableSet::Representation x(variableSet);
   assembler.assemble(linearization(F,x));
   CoefficientVectors solution(VariableSet::
       CoefficientVectorRepresentation<0,neq>::init(variableSet));
   solution = 0;
   CoefficientVectors rhs(assembler.rhs());
   AssembledGalerkinOperator<Assembler,0,1,0,1> A(assembler,
       onlyLowerTriangle);
   MatrixAsTriplet<double> tri = A.get<MatrixAsTriplet<double> >()
       ;

   if (direct) {
     boost::timer::cpu_timer directTimer;
     directInverseOperator(A,directType,property).applyscaleadd
         (-1.0,rhs,solution);
     x.data = solution.data;
     if ( verbosity>0) std::cout << "direct solve: " << (double)(
         directTimer.elapsed().user)/1e9 << "s\n";
   }
   else {
     int iteSteps = getParameter(pt, "solver.iteMax", 1000);
     double iteEps = getParameter(pt, "solver.iteEps", 1.0e-10);
     boost::timer::cpu_timer iteTimer;
     typedef VariableSet::CoefficientVectorRepresentation<0,neq>::
         type LinearSpace;
     Dune::InverseOperatorResult res;
     solution = 1.0;
     switch (precondType)
     {
       ...
       ...
       case JACOBI:
       default:
       {
         JacobiPreconditioner<AssembledGalerkinOperator<Assembler
             ,0,1,0,1> > jacobi(A,1.0);
         Dune::CGSolver<LinearSpace> cg(A,jacobi,iteEps,iteSteps
```

```
            ,0);
          cg.apply(solution,rhs,res);
        }
        break;
    }
    solution *= -1.0;
    x.data = solution.data;
    if ( verbosity>0) std::cout << "iterative solve eps= " <<
        iteEps << ": " << (res.converged?"converged":"failed") << "
         after " << res.iterations << " steps, rate=" << res.
        conv_rate << ", time=" << (double)(iteTimer.elapsed().user)
        /1e9 << "s\n";
    }

    // error estimate
    VariableSet::Representation e = x;
    projectHierarchically(variableSet,e);
    e -= x;

    // use of error estimate for adaptive mesh refinement
    accurate = embeddedErrorEstimator(variableSet,e,x,
        IdentityScaling(),tol,gridManager);

    // necessary updates for next loop cycle
    nnz = assembler.nnz(0,1,0,1,onlyLowerTriangle);;
    size = variableSet.degreesOfFreedom(0,1);

    // VariableSet::Representation xx may be used beyond the do...
        while loop
    xx.data = x.data;
}   while (!accurate);
```

In this partition of the main program we have the loop in which the solution of the linear system (resulting from finite element discretization) is followed by an embedded estimation of the discretization error (as described in Section 2.4) and by adaptive mesh refinement as long as the estimated error exceeds a threshold. If the estimate is smaller than the threshold the value of `accurate` is set to *true* otherwise to *false*.

The embedded error estimator needs an approximation of order $p$ which is considered in hierarchical bases. The difference between order $p-1$ projection (provided by function *projectHierarchically*) and order $p$ solution yields a measure for the error. For instance, if we provide a solution with quadratic finite elements ($order = 2$), we substract the embedded (in hierarchical bases point of view) part of $order = 1$, of cause in the same bases in which the solution is considered.

Parameters of the program (e.g., order, refinement, etc) can be specified by the dynamic parameter handling as described in example (6.2). As default values we

set $order = 2$, $refinement = 5$, $solver.type = direct$, etc., look into the source code for the other parameters.

In top of the file *peaksource.cpp* you find the definition

```
#define DIM 2
```

which is used to select the space dimension of the problem, possible values are 2 and 3. While the grid for the 2d problem is defined directly in the top of the main program we exported the definition for 3D into the file *cubus.hh*. The three-dimensional problem needs a lot of memory when started with $refinement = 5$, therefore we recommend a moderate refinement of the initial mesh, e.g., $refinement = 3$, and to use an iterative linear solver.

The most important parameters for controlling the adaptive process are the absolute tolerance $atol$ and the relative tolerances $rtol$ to be specified via the dynamic parameter handling. The adaptive refinement is stopped as soon as we have for the estimated error $e$:

$$||e||^2 < atol^2 + rtol^2||u||^2$$

.

After computing the solution it is written to file *temperature.vtu* for visualization objects.

## 6.6 Using hierarchical error estimation

This example is implemented in subdirectory

KASKADE7/tutorial/HB_errorEstimation

Files: **heat.cpp**, **heat3d.cpp**, **poisson.hh**, **Makefile**

Here we consider the same problem as in the Section 6.5. In 2D, we define the region $\Omega$ in the main program (*heat.cpp*). The 3D example is presented in *heat3d.cpp*. Here we have to use the dynamic parameter handling in order to specify a file in which a geometry is described in the *amiramesh* format. One such file is *cube.am* describing the unit cube. The user may specify the file name in the command for starting the executable or else *cube.am* is taken as default:

```
./heat3d --file cube.am
```

On the whole boundary we prescribe the values $T = 0$ (Dirichlet condition). In 3D, the equation reads analogously, using additional terms for the third dimension.

As in the example of Section 6.5 we control the discretization error by an error estimation. The estimator provides information about the global and the local error. Local error estimates are used to refine the mesh where the estimated error exceeds a threshold. If the estimate of the global error indicates sufficient accuracy the sequence of *compute solution on a fixed mesh*, *estimate the discretization error*, and *refine* stops. The sequence stops also if the maximum number of refinement steps *maxAdaptSteps* is reached.

```
do
{
  refSteps++;

  CoefficientVectors solution(VariableSet::
      CoefficientVectorRepresentation<0,neq>::init(variableSet));
  solution = 0;
  CoefficientVectors hilfe(VariableSet::
      CoefficientVectorRepresentation<0,neq>::init(variableSet));
  hilfe = 0;
  assembler.assemble(linearization(F,x));
  CoefficientVectors rhs(assembler.rhs());
  typedef AssembledGalerkinOperator<Assembler,0,1,0,1> AssOperator
      ;
  AssembledGalerkinOperator<Assembler,0,1,0,1> A(assembler,
      onlyLowerTriangle);

  if (direct)
  {
    directInverseOperator(A,directType,property).applyscaleadd
        (-1.0,rhs,solution);
  }
  else
  {
    switch (iterateType)
    {
      case CG:
      {
        if ( verbosity>0 ) std::cout << "preconditioned cg solver
            is used" << std::endl;
        JacobiPreconditioner<AssOperator> jacobi(A,1.0);
        Dune::CGSolver<LinearSpace> cg(A,jacobi,iteEps,iteSteps,
            verbosity-1);
        cg.apply(hilfe,rhs,res);
      }
      break;
      case PCG:
      {
        if ( verbosity>0) std::cout << "   preconditioned cascadic
```

```
            multigrid solver I is used" << std::endl;
        JacobiPreconditioner<AssOperator> jacobi(A,1.0);
        NMIIIPCGSolver<LinearSpace> pcg(A,jacobi,iteEps,iteSteps,
            verbosity-1);
        pcg.apply(hilfe,rhs,res);
      }
      break;
      case APCG:
      {
        if (verbosity>0) std::cout << "   preconditioned cascadic
            multigrid solver II is used" << std::endl;
        int addedIterations = getParameter(pt, "solver.APCG.
            addedIterations", 10);
        JacobiPreconditioner<AssOperator> jacobiPCG(A,1.0);
        NMIIIAPCGSolver<LinearSpace> apcg(A,jacobiPCG,iteEps,
            iteSteps,verbosity-1,addedIterations);
        apcg.apply(hilfe,rhs,res);
      }
      break;
      default:
        std::cout << "Solver not available" << std::endl;
        throw -111;
    }
    solution.axpy(-1,hilfe);
}

dx.data = solution.data;


// Do hierarchical error estimation. Remember to provide the
    very same underlying problem to the
// error estimator functional as has been used to compute dx (do
    not modify x!).
std::vector<double> errorDistribution(is.size(0),0.0);
typedef VariableSet::GridView::Codim<0>::Iterator CellIterator;
double maxErr = 0.0;
double errLevel = 0.0;

if (!tolX.empty())
{
  tmp *= 0;
  estGop.assemble(ErrorEstimator(LinearizationAt<Functional>(F,x
      ),dx));
  int const estNvars = ErrorEstimator::AnsatzVars::noOfVariables
      ;
  int const estNeq = ErrorEstimator::TestVars::noOfVariables;
  size_t estNnz = estGop.nnz(0,estNeq,0,estNvars,false);
  size_t estSize = exVariableSet.degreesOfFreedom(0,estNvars);
```

```
    std::vector<int> estRidx(estNnz), estCidx(estNnz);
    std::vector<double> estData(estNnz), estRhs(estSize),
        estSolVec(estSize);
    estGop.toSequence(0,estNeq,estRhs.begin());

    // iterative solution of error estimator

    typedef AssembledGalerkinOperator<EstAssembler> AssEstOperator
        ;
    AssEstOperator agro(estGop);
    Dune::InverseOperatorResult estRes;
    ExCoefficientVectors estRhside(estGop.rhs());
    ExCoefficientVectors estSol(ExVariableSet::
        CoefficientVectorRepresentation<0,1>::init(exVariableSet))
        ;
    estSol = 1.0 ;
    JacobiPreconditioner<AssEstOperator> jprec(agro, 1.0);
    jprec.apply(estSol,estRhside); //single Jacobi iteration
    estSol.write(estSolVec.begin());

    // Transfer error indicators to cells.
    for (CellIterator ci=variableSet.gridView.begin<0>(); ci!=
        variableSet.gridView.end<0>(); ++ci)
    {
      typedef H1ExSpace::Mapper::GlobalIndexRange GIR;
      double err = 0;
      GIR gix = spaceEx.mapper().globalIndices(*ci);
      for (GIR::iterator j=gix.begin(); j!=gix.end(); ++j)
         err += fabs(component<0>(estSol)[*j]);
      errorDistribution[is.index(*ci)] = err;
      if (fabs(err)>maxErr) maxErr = fabs(err);
    }

    errLevel = 0.5*maxErr;
    if (minRefine>0.0)
    {
      std::vector<double> eSort(errorDistribution);
      std::sort(eSort.begin(),eSort.end(),compareAbs);
      int minRefineIndex = minRefine*(eSort.size()-1);
      double minErrLevel = fabs(eSort[minRefineIndex])+1.0e-15;
      if (minErrLevel<errLevel) errLevel = minErrLevel;
    }

    errNorm = 0 ;
    for (k=0; k < estRhs.size() ; k++ )
       errNorm +=  fabs(estRhs[k]*estSolVec[k]) ;
    }

  // apply the Newton correction here
```

```
x += dx;

if ( verbosity>0)
  std::cout << "step = " << refSteps << ": " << size << " points
      ,   ||estim. err|| = " << errNorm <<  std::endl;

// graphical output of solution, mesh is not yet refined
std::ostringstream fn;
fn << "graph/peak-grid";
fn.width(3);
fn.fill('0');
fn.setf(std::ios_base::right,std::ios_base::adjustfield);
fn << refSteps;
fn.flush();
writeVTK(x,fn.str(),IoOptions().setOrder(order));
std::cout << "   output written to file " << fn.str() << std::
    endl;

if (refSteps>maxAdaptSteps)
{
  std::cout << "max. number of refinement steps is reached" <<
      std::endl;
  break;
}

// Evaluation of (global/local) error estimator information
if (!tolX.empty())
{
  if (errNorm<requested)
  {
    accurate = true ;
    std::cout << "||estim. error|| is smaller than requested" <<
        std::endl;
  }
  else
  {
    // Refine mesh.
    int noToRefine = 0;
    double alphaSave = 1.0 ;
    std::vector<bool> toRefine( is.size(0), false ) ; //for
        adaptivity in compression
    std::vector< std::vector<bool> > refinements;
    for (CellIterator ci=variableSet.gridView.begin<0>(); ci!=
        variableSet.gridView.end<0>(); ++ci)
      if (fabs(errorDistribution[is.index(*ci)]) >= alphaSave*
          errLevel)
      {
        noToRefine++;
        toRefine[is.index(*ci)] = true ;
```

```
          gridManager.mark(1,*ci);
        }

      refinements.push_back(toRefine);
      accurate = !gridManager.adaptAtOnce();
    }
  }

  size = variableSet.degreesOfFreedom(0,1);
  qk = size/dNk;
  dNk = size;
  yk += pow(dNk,alpha);
  zk = pow(dNk,alpha)*(pow(errNorm/requested,d*alpha)-pow(qk,alpha
      ))/(pow(qk,alpha)-1.0);
  iteEps = safety*beta*errNorm*yk/(yk+zk);

} while (!accurate);
```

## 6.7 SST pollution

This example is implemented in subdirectory

KASKADE7/tutorial/sst_pollution.

Files: **sst.cpp**, **sst_nleqErr.cpp**, **sst_giant.cpp**, **sst.hh**, **Makefile**

The following example is a straightforward extension of an instationary 1-D-model for the pollution of the stratosphere by supersonic transports (SSTs). The rather crude model describes the interaction of the chemical species O, $O_3$, NO and $NO_2$ along with a simple diffusion process.

The equations for the stationary 2D-model are:

$$\begin{aligned}
0 &= D\Delta u_1 + k_{1,1} - k_{1,2}u_1 + k_{1,3}u_2 + k_{1,4}u_4 - k_{1,5}u_1u_2 - k_{1,6}u_1u_4\,,\\
0 &= D\Delta u_2 + k_{2,1}u_1 - k_{2,2}u_2 + k_{2,3}u_1u_2 - k_{2,4}u_2u_3\,,\\
0 &= D\Delta u_3 - k_{3,1}u_3 + k_{3,2}u_4 + k_{3,3}u_1u_4 - k_{3,4}u_2u_3 + 800.0 + SST\,,\\
0 &= D\Delta u_4 - k_{4,1}u_4 + k_{4,2}u_2u_3 - k_{4,3}u_1u_4 + 800.0\,,
\end{aligned} \tag{25}$$

where

$$
\begin{aligned}
D &= 0.5 \cdot 10^{-9} \,, \\
k_{1,1}, \ldots, k_{1,6} &: 4 \cdot 10^5, 272.443800016, 10^{-4}, 0.007, 3.67 \cdot 10^{-16}, 4.13 \cdot 10^{-12} \,, \\
k_{2,1}, \ldots, k_{2,4} &: 272.4438, 1.00016 \cdot 10^{-4}, 3.67 \cdot 10^{-16}, 3.57 \cdot 10^{-15} \,, \\
k_{3,1}, \ldots, k_{3,4} &: 1.6 \cdot 10^{-8}, 0.007, 4.1283 \cdot 10^{-12}, 3.57 \cdot 10^{-15} \,, \\
k_{4,1}, \ldots, k_{4,3} &: 7.000016 \cdot 10^{-3}, 3.57 \cdot 10^{-15}, 4.1283 \cdot 10^{-12} \,, \\
SST &= \begin{cases} 3250 & \text{if } (x,y) \in [0.5, 0.6]^2 \\ 360 & \text{otherwise.} \end{cases}
\end{aligned}
$$

The boundary conditions are

$$
\left. \frac{\partial u_i}{\partial n} \right|_{\partial \Omega} = 0 \qquad \text{for } i = 1, \ldots, 4 \tag{26}
$$

and the domain $\Omega$ is just the unit square of $\mathbb{R}^2$. Two sets of startvalues are supplied - the first one produces a mildly nonlinear problem, which can be solved applying an ordinary Newton iteration. The startvalues for the ordinary Newton iteration are

$$
\begin{aligned}
u_1^0(x,y) &= 1.306028 \cdot 10^6 \,, \\
u_2^0(x,y) &= 1.076508 \cdot 10^{12} \,, \\
u_3^0(x,y) &= 6.457715 \cdot 10^{10} \,, \\
u_4^0(x,y) &= 3.542285 \cdot 10^{10}
\end{aligned}
\qquad (x,y) \in \Omega \,. \tag{27}
$$

The second set of startvalues produces a highly nonlinear problem for which it is necessary to apply a damped Newton method in order to solve it. The second set of startvalues is

$$
\begin{aligned}
u_1^0(x,y) &= 10^9 \,, \\
u_2^0(x,y) &= 10^9 \,, \\
u_3^0(x,y) &= 10^{13} \,, \\
u_4^0(x,y) &= 10^7
\end{aligned}
\qquad (x,y) \in \Omega \,. \tag{28}
$$

The solution of the mildly nonlinear variant of the SST problem is in detail featured in the main programm **sst.cpp**, where the Newton iteration loop is included, as well as computations of the norms of the Newton correction and residual vector in each iteration step. The linear systems arising in each Newton step are solved using the iterative DUNE solver BiCGSTAB together with the ILUK preconditioner.

For solving the highly nonlinear variant of the SST problem, we provide headerfiles for two alternative damped Newton methods. The use of the two methods are featured in the programs **sst_nleqErr.cpp** and **sst_giant.cpp**.

In the program **sst_nleqErr.cpp** the nonlinear solver class *NleqSolver*, which is defined in the headerfile **algorithm/nleq_err.hh** is used to solve the problem. The class *NleqSolver* is a simplified implementation for KASKADE7 of the algorithm due to Deuflhard [8], p.148f, without the switch to a Quasi-Newton iteration nearby the solution.

In the program **sst_giant.cpp** the nonlinear solver class *Giant*, which is defined in the headerfile **algorithm/giant_gbit.hh** is used to solve the problem. The class *Giant* is an implementation for KASKADE7 of the GIANT-GBIT algorithm due to Deuflhard [8], p.160f.

## 6.8 Stokes equation

This example is implemented in subdirectory

KASKADE7/tutorial/stokes.

Files: **stokes.cpp**, **stokes.hh**, **Makefile**
We consider the stationary 2D driven cavity problem for incompressible fluids on the unit square $\Omega$ governed by the linear Stokes equation:

$$\begin{aligned} -\Delta u + \nabla p &= 0 && \text{(x,y)} \in \Omega = [0,1]\times[0,1] \\ \nabla \cdot u &= 0 && \text{(x,y)} \in \Omega \\ u(x,y) &= -1 && \text{on } \partial\Omega \text{ if } y = 1 \\ u(x,y) &= 0 && \text{elsewhere on } \partial\Omega, \end{aligned} \tag{29}$$

with velocity $u = u(x,y)$ and pressure $p = p(x,y)$.

## 6.9 Elasticity

This example is implemented in subdirectory

KASKADE7/tutorial/elastomechanics.

Files: **elastomechanics.cpp**, **elastomechanics.hh**, **Makefile**
We consider the three dimentional linear elasticity problem defined by the Navier-Lame equations on a unit cube domain $\Omega$ and we compute the dispacement vector $u$ for the corresponding vector linear elliptic pde where $f$ is the force vector:

$$\begin{aligned} \nabla \cdot \sigma &= 0 \qquad\qquad (x,y,z) \in \Omega = [0,1]\times[0,1]\times[0,1] \\ \sigma &= \lambda tr(\varepsilon) I + 2\mu\varepsilon \\ \varepsilon &= \tfrac{1}{2}(\nabla u + (\nabla u)^T) \\ F(u) &= \tfrac{1}{2}(\sigma : \varepsilon) \\ G(u) &= \tfrac{1}{2}\alpha(u-u_b)^2 - fu \end{aligned} \tag{30}$$

Inserting the linearized Green Lagrange strain tensor $\varepsilon$ into the stress tensor $\sigma$ we end up with the Navier-Lame equations, where $\lambda$ and $\mu$ are the Lame constants.

$$(\lambda+\mu)\nabla(\nabla\cdot u) + \mu\nabla^2 u = 0 \tag{31}$$

Going through the main program *elastomechanics.cpp* the set up for defining the grid and computing the solution is done analogously to the 3D stationary heat transfer in (6.2), with the difference in giving the displacement variable 3 components:

```
auto varSetDesc = makeVariableSetDescription(makeSpaceList(&
    h1Space), make_vector(Variable<SpaceIndex<0>,Components<3>>(
    "u")));
```

For creating the functional object an *ElasticModulus* object is passed to the ElasticityFunctional defined in *elastomechanics.hh*, the *ElasticModulus* object can be initilized by either choosing a predefined material in KASKADE7 library where in this example steel is used (a list of the known materials is printed to the command line when running the example) or by giving the Lame constants $\lambda$ and $\mu$.

```
using Functional = ElasticityFunctional<VarSetDesc>;
using Assembler = VariationalFunctionalAssembler<LinearizationAt
    <Functional> >;
using CoefficientVectors = VarSetDesc::
    CoefficientVectorRepresentation<0,1>::type;

// Create the variational functional.
Functional F(ElasticModulus::material(material));
//Functional F(ElasticModulus(double lambda, double mu));
```

Moving to *elastomechanics.hh* to define the necessary *DomainCache* we will make use of the *LameNavier* class and its member functions *d0(), d1(), d2() and setLinearizationPoint()*, where we creat an object *energy* of it which is initilized with the *ElasticModulus*

*setLinearizationPoint()* takes the tensor of displacement derivative at the current evaluation point as it's input. Where it defines the displacement derivatives around which to linearize as will as the strain tensor.

```
// Deriving from FunctionalBase introduces default D1 and D2
    structures.
template <class VarSet>
class ElasticityFunctional: public Kaskade::FunctionalBase<
    VariationalFunctional>
{
public:
...

using ElasticEnergy = Kaskade::Elastomechanics::LameNavier<dim,
    Scalar>;

...


  class DomainCache
  {
  public:
    DomainCache(ElasticityFunctional const& functional, typename
        AnsatzVars::VariableSet const& vars_, int flags=7)
    : vars(vars_), energy(functional.moduli)
    {}

    template <class Entity>
    void moveTo(Entity const& entity) {}

    template <class Position, class Evaluators>
    void evaluateAt(Position const& x, Evaluators const&
        evaluators)
    {
      energy.setLinearizationPoint( boost::fusion::at_c<u_Idx>(
          vars.data).derivative(boost::fusion::at_c<u_Space_Idx>(
          evaluators)) );
    }
```

The member function *d0()* returns the scalar value of the elastic stored energy density which defines our functional *F(u)*:

$$\begin{aligned} F(u) &= \tfrac{1}{2}(\sigma : \varepsilon) \\ &= \tfrac{\lambda}{2} tr(\varepsilon)^2 + \mu(\varepsilon : \varepsilon) \end{aligned} \tag{32}$$

```
    Scalar d0() const
```

```
  {
    return energy.d0();
  }
```

The member function *d1()* returns the vector defining the first directional derivative of the elastic stored energy density which defines our $F'(u)$:

$$F'(u,\varphi_i) = \lambda tr(\varepsilon)\nabla\varphi_i + 2\mu(\varepsilon)\nabla\varphi_i \tag{33}$$

```
  template<int row>
  Vector d1 (VariationalArg<Scalar,dim> const& arg) const

  {
    return energy.d1(arg);
  }
```

The member function *d2()* returns the matrix defining the second directional derivative of the elastic stored energy density which defines our $F''(\varphi_i,\varphi_j)$:

$$F''(\varphi_i,\varphi_j) = \varepsilon(\nabla\varphi_i)^T C\varepsilon(\nabla\varphi_j) \tag{34}$$

Where $C$ is the fourth-order linear elastic stiffness tensor and $\varepsilon(\nabla\varphi)$ represents the strain displacement matrix.

$$\varepsilon(\nabla\varphi)^T = \begin{bmatrix} \frac{\partial\varphi}{\partial x} & 0 & 0 \\ 0 & \frac{\partial\varphi}{\partial y} & 0 \\ 0 & 0 & \frac{\partial\varphi}{\partial z} \\ 0 & \frac{\partial\varphi}{\partial z} & \frac{\partial\varphi}{\partial y} \\ \frac{\partial\varphi}{\partial z} & 0 & \frac{\partial\varphi}{\partial x} \\ \frac{\partial\varphi}{\partial y} & \frac{\partial\varphi}{\partial x} & 0 \end{bmatrix}, C = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix}$$

```
  template<int row, int col>
  Matrix d2 (VariationalArg<Scalar,dim> const& argTest,
      VariationalArg<Scalar,dim> const& argAnsatz) const
  {
    return energy.d2(argTest,argAnsatz);
  }

private:
  typename AnsatzVars::VariableSet const& vars;
  ElasticEnergy energy;
```

```
};
```

Moving to the boundary cache we will define an inhomogeneous Neumann boundary condition of 1N on the cube's bottom face which has an outward normal of [0 , -1 , 0] and a homogeneous Neumann boundary condition on the cube's top face having an outward normal of [0 , 1 , 0], the remaining 4 faces we will assign a homogeneous Dirichlet boundary condition. To do so we will make use of the member function *moveTo* which takes a *FaceIterator* iterating over the cells intersecting with the domain boundary to assign the penalty value *alpha* and the force value *beta* to each cell on the boundary. The *FaceIterator* object *const & face* points to a *Dune::intersection* type which has the member function *centerUnitOuterNormal()* returning the outward unit normal *n* of the current cell.

```
class BoundaryCache : public CacheBase<ElasticityFunctional,
    BoundaryCache>
{
public:
  using FaceIterator = typename AnsatzVars::Grid::
      LeafIntersectionIterator;

  BoundaryCache(ElasticityFunctional const& f_, typename
      AnsatzVars::VariableSet const& vars_, int flags=7)
  : vars(vars_)
  {}


  void moveTo(FaceIterator const& face)
  {
    Vector up(0); up[1] = 1;                          // unit
        upwards pointing vector
    auto n = face->centerUnitOuterNormal();           // unit
        outer normal

    if (n*up > 0.5)           // top face: natural boundary
        conditions (homogeneous Neumann, zero normal stress)
    {
      alpha = 0;              // retardation force factor
      beta  = 0;              // absolute force vector
    }
    else if (n*up < -0.5)     // bottom face: force boundary
        condition (inhomogeneous Neumann, given normal stress)
    {
      alpha = 0;
      beta  = up;
    }
    else                      // side face: essential boundary
```

```
      conditions (homogeneous Dirichlet)
  {
    alpha = 1e14;          // requires large penalty for hard
       materials such as steel with a Young's modulus around
       7e9
    beta  = 0;
  }
}
```

In this example we will define the necessary member functions $d0^{\Gamma}()$, $d1^{\Gamma}\_impl()$, and $d2^{\Gamma}\_impl()$ as follows:

$$\begin{aligned} d0^{\Gamma}() &= \tfrac{1}{2}\alpha(u-u_0)^2 - \beta u \\ d1^{\Gamma}\_impl(\varphi_i) &= \alpha u\varphi_i - \beta\varphi_i \\ d2^{\Gamma}\_impl(\varphi_i,\varphi_j) &= \alpha(\varphi_i\varphi_j) \end{aligned} \tag{35}$$

```
template <class Evaluators>
void evaluateAt(Dune::FieldVector<typename AnsatzVars::Grid::
    ctype,dim-1> const& x, Evaluators const& evaluators)
{
  using namespace boost::fusion;

  u = at_c<u_Idx>(vars.data).value(at_c<u_Space_Idx>(
      evaluators));
}

Scalar
d0() const
{
  return alpha*(u*u)/2 - beta*u;
}

template<int row>
Scalar d1_impl (VariationalArg<Scalar,dim,dim> const& arg)
    const
{
  return alpha*(u*arg.value) - beta*arg.value;
}

template<int row, int col>
Scalar d2_impl (VariationalArg<Scalar,dim,dim> const &arg1,
    VariationalArg<Scalar,dim,dim> const &arg2) const
{
  return alpha*(arg1.value*arg2.value);
}
```

```
private:
  typename AnsatzVars::VariableSet const& vars;
  Vector u, beta;
  Scalar alpha;
};
```

It is also valid to replace the member functions $d1^{\Gamma}\_impl()$, and $d2^{\Gamma}\_impl()$ with $d1^{\Gamma}()$, and $d2^{\Gamma}()$

```
    template<int row>
    Dune::FieldVector<Scalar, dim> d1 (VariationalArg<Scalar,dim
        > const& arg) const
    {
      return alpha * u*arg.value[0] - beta*arg.value[0];
    }

    template<int row, int col>
    Dune::FieldMatrix<Scalar, dim, dim>
    d2 (VariationalArg<Scalar,dim> const &arg1, VariationalArg<
        Scalar,dim> const &arg2) const
    {
      return alpha * arg1.value[0] * arg2.value[0] * unitMatrix<
          Scalar,dim>();
    }
```

Going back to *elasomechanics.cpp* we will discuss how to postprocess the normal stressess $\sigma_{xx}$, $\sigma_{yy}$ and $\sigma_{zz}$ and output them with the displacement field to the file *elasto.vtu*. To do so we will make use of the *Kaskade::interpolateGlobally* function which interpolates from a *FunctionSpaceElement* to another. We interpolate values from the FE space *h1Space* where the solution was carried out to be stored on a *l2Space*. First we creat a *l2Space* with 3 components( *normalStress*)

```
  //Postprocessing

  L2Space<Grid>::Element_t<DIM> normalStress(l2Space);
```

The *interpolateGlobally* function will iterate over all evaluation points in the solution space, within it we redefine the functional and pass the solution $x$ to *setLinearizationPoint* after which use the functional member function *cauchyStress* to retrieve stress components.

```
  interpolateGlobally<PlainAverage>(normalStress,makeFunctionView
      (h1Space, [&] (auto const& evaluator)
   {
```

```
      auto modulus = ElasticModulus::material(material);
      HyperelasticVariationalFunctional<Elastomechanics::
          MaterialLaws::StVenantKirchhoff<DIM>,
          LinearizedGreenLagrangeTensor<double,DIM>>
      energy(modulus);

      energy.setLinearizationPoint(component<0>(x).derivative(
          evaluator));
      auto stress = Dune::asVector(energy.cauchyStress());

      return Dune::FieldVector<double,3>{stress[0],stress[4],
          stress[8]};
    }));

     auto vsd = makeVariableSetDescription(makeSpaceList(&l2Space)
         ,
     boost::fusion::make_vector(Variable<SpaceIndex<0>,Components
         <3>>("NormalStress"),
     Variable<SpaceIndex<0>,Components<3>>("Dispacement")));

    auto data = vsd.variableSet();
    component<0>(data) = normalStress;
    component<1>(data) = component<0>(x);

  // output of solution in VTK format for visualization,
  // the data are written as ascii stream into file elasto.vtu,
  // possible is also binary
  if (vtk)
  {
    ScopedTimingSection ts("computing time for file i/o",timer);
    writeVTKFile(data,"elasto",IoOptions().setOrder(order).
        setPrecision(7));
  };
 .....

  return 0;
}
```

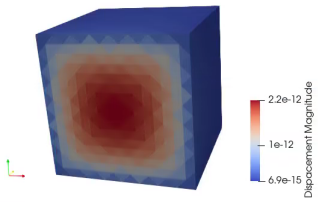


Figure 2: Paraview visualization of the displacement field magnitude

## 6.10 Instationary heat tranfer

This example is implemented in subdirectory

KASKADE7/tutorial/instationary_heattransfer.

Files: **movingsource.cpp**, **movingsource.hh**, **integrate.hh**, **Makefile**
We consider a simple instationary heat transfer problem in the two-dimensional region $\Omega$. The corresponding linear parabolic equation is given by

$$\begin{aligned} \frac{\partial T}{\partial t} - \nabla \cdot (\kappa(x) \nabla T) &= f(x) \qquad \text{x} \in \Omega, \quad 0 < t <= T \\ T &= T_b(x,t) \qquad \text{on } \Gamma \\ T &= T_0 \qquad \text{on } \Omega \end{aligned} \tag{36}$$

The term $\Gamma$ denotes the boundary $\partial\Omega$ where we have to define boundary conditions. We prescribe the values of the solution $T$ (Dirichlet condition). The initial value of the solution at $t = 0$ is given by the function $T_0$ on the region $\Omega$.

In KASKADE7 , the solution of this problem is based on a Rothe method, i.e. we first discretize in time, then the resulting elliptic problems by a finite element method (FEM). Therefore it is necessary to consider the system (36) in its weak formulation, see appendix. Analogous to the stationary problems considered before we define the mesh, the spaces for the spatial discretization, and the functional due to weak formulation:

```
int main(int argc, char *argv[])
{
  int const dim = 2;
  int refinements = 6, order = 2, extrapolOrder = 2, maxSteps =
      40;
  double dt = 0.1, maxDT = 1.0, T = 10.0, rTolT = 1.0e-2, aTolT =
      1.0e-2, rTolX = 1.0e-5, aTolX = 1.0e-5, writeInterval = 1.0;

  boost::timer::cpu_timer totalTimer;
  std::cout << "Start moving source tutorial program" << std::endl
      ;

  typedef Dune::UGGrid<dim> Grid;
  std::unique_ptr<Grid> grid( RefineGrid<Grid>(refinements) );

  std::cout << "Grid: " << grid->size(0) << " " << grid->size(1)
     << " " << grid->size(2) << std::endl;

  GridManager<Grid> gridManager(std::move(grid));
```

```
  // construct involved spaces.
  typedef FEFunctionSpace<ContinuousLagrangeMapper<double,Grid::
      LeafGridView> > H1Space;

  H1Space temperatureSpace(gridManager,gridManager.grid().leafView
      (),order);

  typedef boost::fusion::vector<H1Space const*> Spaces;
  Spaces spaces(&temperatureSpace);

  typedef boost::fusion::vector<VariableDescription<0,1,0> >
      VariableDescriptions;
  std::string varNames[1] = { "u" };

  typedef VariableSetDescription<Spaces,VariableDescriptions>
      VariableSet;
  VariableSet variableSet(spaces,varNames);

  typedef MovingSourceEquation<double,VariableSet> Equation;
  Equation Eq;

  std::vector<VariableSet::Representation> solutions;

  Eq.time(0);
  VariableSet::Representation x(variableSet);
  Eq.scaleInitialValue<0>(InitialValue(0),x);

  x = integrate(gridManager, Eq, variableSet, spaces, gridManager.
      grid(), dt, maxDT, T, maxSteps, rTolT, aTolT, rTolX, aTolX,
      extrapolOrder, std::back_inserter(solutions), writeInterval,
       x, DirectType::MUMPS);

  std::cout << "End moving source tutorial program" << std::endl;
  std::cout << "used cpu time: " << (double)(totalTimer.elapsed().
      user)/1e9 << "s\n";
}
```

Kernel of this implementation is the function *integrate* which performs the integration in time based on the *LIMEX* method:

```
template <class Grid, class Equation, class VariableSet, class
    Spaces, class OutIter>
typename VariableSet::Representation
         integrate(GridManager<Grid>& gridManager,
                   Equation& eq, VariableSet const& variableSet,
                       Spaces const& spaces,
                   Grid const& grid, double dt, double dtMax,
                   double T, int maxSteps, double rTolT, double
```

```
                        aTolT,
                    double rTolX, double aTolX, int extrapolOrder,
                    OutIter out, double outInterval,
                    typename VariableSet::Representation x,
                    DirectType directType,
                    int verbosity = 0)
{
  // write initial position
  *out = x;
  ++out;
  double outTime = eq.time()+outInterval;

  std::vector<std::pair<double,double> > tolX(variableSet.
      noOfVariables);
  std::vector<std::pair<double,double> > tolXC(variableSet.
      noOfVariables);

  for (int i=0; i<tolX.size(); ++i)
  {
    tolX[i] = std::make_pair(aTolX,rTolX);
    tolXC[i] = std::make_pair(aTolX/100,rTolX/100);
  }

  std::vector<std::pair<double,double> > tolT(variableSet.
      noOfVariables);
  for (int i=0; i<tolT.size(); ++i)
    tolT[i] = std::make_pair(aTolT,rTolT);

  Limex<Equation> limex(gridManager,eq,variableSet,directType,ILUK
      ,verbosity);

  Dune::FieldVector<double,2> samplePos(0);

  int steps;
  bool done = false;
  for (steps=0; !done && steps<maxSteps; ++steps) {
    if (eq.time()>T-1.1*dt)
    {
      dt = T-eq.time();
      done = true;
    }

    typename VariableSet::Representation dx(x);

    int redMax = 5;
    int red = 0;
    double factor = 1.0;
    double err;
```

```
do
{
  dt *= factor;
  dx = limex.step(x,dt,extrapolOrder,tolX);

  std::vector<std::pair<double,double> >
       errors = limex.estimateError(x,extrapolOrder,
           extrapolOrder-1);
  err = 0;
  for (int i=0; i<errors.size(); ++i)
    err = std::max(err, errors[i].first/(tolT[i].first+tolT[i
        ].second*errors[i].second));

  factor = std::pow(0.5/err,1./(extrapolOrder+2));
  factor = std::max(0.5,std::min(factor,1.33));
  std::cout << eq.time() << ' ' << dt << ' ' << err << ' ' <<
      red << ' '
            << variableSet.degreesOfFreedom()
            << ' ' << component<0>(x).value(samplePos)[0] << '
                \n';
  ++red;
} while(err>1 && red<=redMax);

if ( verbosity>0 )
{
  std::cout.flush();
  std::cout << "t= " << eq.time() << " dt = " << dt << "
      factor = " << factor << " red=" << red << '\n';
};

// write linearly interpolated equidistant output
assert(eq.time()<=outTime);
while (outTime<=eq.time()+dt)
{
  typename VariableSet::Representation z(x);
  z.axpy((outTime-eq.time())/dt,dx);
  outTime += outInterval;
}

// step ahead
x += dx;
eq.time(eq.time()+dt);
dt = std::min(dt*factor,dtMax);

// perform mesh coarsening
coarsening(variableSet,x,eq.scaling(),tolX,gridManager);

if (1)
{
```

```
      // debugging output
      typedef typename Grid::LeafGridView LeafGridView;
      LeafGridView leafView = gridManager.grid().leafView();

      IoOptions options;
      options.outputType = IoOptions::ascii;
      writeVTK(x,"graph/outScaledGrid"+paddedString(2*steps+1),
          options.setOrder(1));

      if ( verbosity>0 )
        std::cout << variableSet.degreesOfFreedom() << " values
            written to " << "graph/outScaledGrid"+paddedString(2*
            steps+1) << std::endl;
    }

  }
  std::cout << '\n';

  limex.reportTime(std::cout);

  if (!done)
    std::cout << "*** maxSteps reached ***\n";

  return x;
}
```

## 6.11 Navier-Stokes equations

This example is implemented in subdirectory

KASKADE7/tutorial/instationary_NavierStokes.

Files: **navierStokes.cpp**, **navierStokes.hh**, **integrate_navierStokes.hh**, **Makefile**
We consider the instationary 2D driven cavity problem for incompressible fluids on the unit square $\Omega$ governed by the Navier-Stokes equations:

$$\begin{aligned} u_t - \nu\Delta u + (u\cdot\nabla)u + \nabla p &= 0 && \text{(x,y)} \in \Omega = [0,1]\times[0,1] \\ \nabla\cdot u &= 0 && \text{(x,y)} \in \Omega \\ u(x,y) &= -1 && \text{on } \partial\Omega \text{ if } y = 1 \\ u(x,y) &= 0 && \text{elsewhere on } \partial\Omega, \end{aligned} \tag{37}$$

with velocity $u = u(x,y)$ and pressure $p = p(x,y)$.

# 7 Parameter administration

## 7.1 Introduction

KASKADE7 is used to compute the solution of a partial differential equation. Often there are a lot of parameters describing details of the equation or details of the solving algorithm. The programmer has two ways to handle these parameters. One way is to define them statically in top of the main program and compile the code whenever one of the values has to be changed. The second way is to use a dynamical support via the command line. To achieve this, different approaches are possible. One possible way is to use the `boost::program_options` methods to define parameters, related default values and descriptions. Another approach which we decided to implement, is based on the `boost::property_tree` utilities. While the program_options approach gives you the advantage of producing a actually helpful output when calling your program with the help option, this approach does not support quite good the simple implementation of self-speaking parameter-names. While with the *program_options* approach, you would have to program the association of names to internal integer values in each application program newly, using our implementation based on using the definitions from the `default.json` file, and in the application the consistent names defined in `utilities/enums.hh`, allows you to use both on the command line as also in your application programs source code self-speaking names.

## 7.2 Implementation in KASKADE7

We use the **boost** packages **property_tree** and **program_options**. In the main program we have to include "utilities/kaskopt.hh" and to add

```
int verbosity = 1;
bool dump = true;
boost::property_tree::ptree *pt = getKaskadeOptions(argc, argv,
    verbosity, dump);
```

The call of `getKaskadeOptions(argc, argv, verbosity, dump)` generates a pointer to a **property_tree** storing a set of hierarchically ordered parameter names each described by a string. On the leaf level a parameter is connected to a default value of one of the types (string, int, double,...). The initial specifications in the property_tree is provided by the entries in the file *default.json*.

Let's develop a structure of the tree in this file:

```
{
 "solver":
```

```
   {
    },
 "names":
   {
   }
}
```

The tree has two main branches, `"solver"` and `"names"`, each of them generates further branches. For example the branch `"solver"` offers the following new branches:

```
 "solver":
   {
     "type": "direct",
     "direct": "UMFPACK",
     "iterate": "CG",
     "property": "GENERAL",
     "preconditioner": "NONE",
     "iteEps": 1.0e-12,
     "iteMax": 1000,
     "ILUT":
       {
       },
     "ICC":
       {
       },
     "BOOMERAMG":
       {
       },
   },
```

The branches `"type"`, `"direct"`, `"iterate"`, `"property"`, ... end on this level (the leaf level) getting a default value. Other branches like `"ILUT"` generate new branches, e.g.,

```
     "ILUT":
       {
         "dropTol": 0.01,
         "lfil": 140
       },
```

We have defined a function call `getParameter` extracting the values of parameters from the `argc -, argv[] -` list. This kind of parameter handling is considered in the following.

Starting the KASKADE7 application such a set of hierarchically ordered parameters is read from the file `default.json` stored in the KASKADE7 root directory. Then the default values of all predefined parameters may be modified by command line. The code has not to be recompiled.

We give some examples illustrating this mechanism, compare also the application in the stationary heat problem from Section 6.2:

**Static parameters.** The user defines all the parameters describing the problem, the descretization, or the solver directly in the code, preferably in top of the main module, e.g.;

- Particular parameters of the problem are set, e.g., the diffusion coefficient $\kappa$ in a heat transfer equation

  **kappa** = 1.0;

- Parameter for discretization

  **refinement** = 5;
  **order** = 3;

  By this the initial grid is refined 5 times, and cubic (order 3) ansatz functions are selected.

- For selecting a direct solver we can set the parameters

  **direct** = true;
  **directType** = DirectType::MUMPS;

  The variable **direct** may have the value TRUE or FALSE and can be used to switch between direct or iterative solution of linear systems. If the option *direct = true* is chosen the user may select also a particular direct solver from an enumeration class defined in the file /utilities/enums.hh, i.e.

```
enum class DirectType { UMFPACK, PARDISO, MUMPS, SUPERLU,
    UMFPACK3264, UMFPACK64 };
```

- Selection and control of an iterative linear solver

  **direct** = false;
  **iterateType** = IterateType::CG;
  **iteEps** = 1.0e-8;
  **iteSteps** = 500;
  **preconditionerType** = PrecondType::ILUK;

  If the option *direct = false* is chosen the linear systems will be solved by an iterative solver. The particular solver may be one of the enumeration class in the file /utilities/enums.hh, i.e.

```
enum class IterateType { CG, BICGSTAB, GMRES, PCG, APCG, SGS
    };
```

A suitable preconditioner is selected from enumeration class in the file /utilities/enums.hh, i.e.

```
enum class PrecondType { NONE, JACOBI, PARTIAL, ILUT, ILUK,
    ARMS, INVERSE, ADDITIVESCHWARZ, BOOMERAMG, EUCLID,
    TRILINOSML, SSOR, ICC0, ICC, ILUKS };
```

The disadvantage of static parameter handling is that recompiling is necessary whenever a parameter is to be changed.

**Dynamic parameters.** Instead of defining parameters statically in the code we offer a possibility to change them via command line.

- `./heat --solver.direct UMFPACK`

  Starting the executable `heat` results in messages of the following kind
  solver.direct=DirectType::MUMPS changed to DirectType::UMFPACK
  Start program (r.1103)
  End program

- Evaluation of a parameter in the program.

```
std::cout << "Start program (r." << getParameter(pt,"version"
    ,empty) << ")";
```

  Some parameter (e.g., the version number) are generated automatically.

```
std::string s("names."), empty;
s += getParameter(pt, "solver.direct", empty);
DirectType directType = static_cast<DirectType>(getParameter(
    pt, s, 2));
directInverseOperator(A,directType,properties).applyscaleadd
    (-1.0,rhs,solution);
```

  The parameter of the routine `getParameter` are the data bank `pt`, the name of the parameter and a default value.

- After reading the set of parameters is documented in a file which includes in its name the calendar date, clock and process id, e.g.,

```
run-2010-02-24-12-37-85744.json
run-2010-02-24-12-55-86099.json
```

- Rerun of the program with the same set of parameter values as in a run before

```
./heat --default run-2010-02-24-12-37-85744.json
```

- Application of a user defined set of parameter values

```
./heat --addons use_amg.json
```

The parameter values of the file `use_amg.json` substitute the default values. The contents of this file maybe look like

```
{
 "solver":
   {
     "type": "iterate",
     "iterate": "CG",
     "property": "GENERAL",
     "preconditioner": "BOOMERAMG",
     "BOOMERAMG":
       {
         "steps": 5,
         "coarsentype": 10,
       }
   }
}
```

Completely new sets of parameters can be provided by this method.

**Structure of a parameter file**. Allowed are files of json, xml, or info format. Here we present an excerpt of an info file:

```
solver
{
    type direct
    name MUMPS
}
names
{
    type
    {
        iterate 0
        direct 1
    }
    direct
    {
        UMFPACK 0
        PARDISO 1
        MUMPS 2
    }
}
run
```

```
{
    version 1103:1104M
    kaskade7 /Users/roitzsch/Kaskade7
    starttime 2010-02-25-09-47
    pid 92109
}
```

or formulated in xml - format:

```
<?xml version="1.0" encoding="utf-8"?>
<solver>
  <type>direct</type>
  <name>MUMPS</name>
</solver>
<names>
  <type>
    <iterate>0</iterate>
    <direct>1</direct>
  </type>
  <direct>
    <UMFPACK>0</UMFPACK>
    <PARDISO>1</PARDISO>
    <MUMPS>2</MUMPS>
  </direct>
</names>
<run>
  <version>1103:1104M</version>
  <kaskade7>/Users/roitzsch/Kaskade7</kaskade7>
  <starttime>2010-02-25-09-26</starttime>
  <pid>-92016</pid>
</run>
```

# 8 Grid types

There are several grid types we can use in KASKADE7 . They work on different element types in 1D, 2D and 3D, e.g., simplices like triangles and tetrahedra, prisms, pyramids, and cubes and are realized as C++ classes. In general they don't include functions for mesh generation but an initial mesh must be provided by the user. However, the grid classes have features to refine or to coarsen (globally or locally) the user defined initial grid. The main libraries available via DUNE interface are

- UGGrid
- ALUGrid
- AlbertaGrid

The class GridManager connects geometrical information about the grid with algebraic information due to the finite element discretization.

# 9 Linear solvers

## 9.1 Direct solvers

KASKADE7 includes some of the well-known direct solvers, such as MUMPS, UMFPACK and PARDISO (see section 4 and appendix C).

The factorization base class `Factorization` has `Scalar` and `SparseIndexInt` as template parameters. The constructor should do the factorization of the sparse matrix and solve method will compute the solution. The matrix is defined by vectors `ridx`, `cidx` and `data` containing the row and column index and the value of the matrix entry.

```
MUMPSFactorization<double,int> matrix(n,nnz,ridx,cidx,data,
    SYMMETRIC);
matrix.solve(rhs,sol);
```

There is predefined factory of solvers which is used by the `directInverseOperator` subroutine.

```
DirectType solver = DirectType::UMFPACK;
MatrixProperties prop = MatrixProperties::GENERAL;
int onlyLowerTriangle = true;

AssembledGalerkinOperator<Assembler,0,1,0,1> A(assembler,
    onlyLowerTriangle);

directInverseOperator(A,solver,prop).applyscaleadd(-1.0,rhs,
    solution);
```

In this case the corresponding initial object has to be included at the lining stage (see the Makefile in `tutorial/stationary_heattransfer` or explicitly in the code.

## 9.2 Iterative solvers, preconditioners

In KASKADE7 we use iterative linear solvers offered by the DUNE ISTL library. The main important are the *cg*-method and the *bicgstab - mthod*. This library also provides a set of preconditioners including *Jacobi*-, *SOR*-, *SSOR* , and *ILU*. In addition there are some other preconditioners from other libraries, see Table 1.
The interfaces between these solvers and KASKADE7 can be studied in the tutorial example `tutorial/stationary_heattransfer`.

| | | |
|---|---|---|
| Incomplete LU | ILUK | `www-users.cs.umn.edu/~saad/software/ITSOL/` |
| Incomplete LU | ILUP | `www-users.cs.umn.edu/~saad/software/ITSOL/` |
| Algebraic MG | ARMS | `www-users.cs.umn.edu/~saad/software/ITSOL/` |
| Algebraic MG | BOOMERAMG | `acts.nersc.gov/hypre/` |

Table 1: Some available preconditioners

# 10 Miscellaneous

## 10.1 Coding Style

The following coding guideline should be followed when programming KASKADE7 modules and headers:

- Indentation depth is 2 spaces, don't use tabs
- Variables must begin with lowcase letters, separate words with upcase letters (e.g. onlyLowerTriangle)
- Names of classes must begin with upcase letters (e.g. HeatFunctional)
- Macronames consist of upcase letters only, words are separated by underlines (e.g. SPACEDIM, ENABLE_ALUGRID)
- Constructor arguments preferably with trailing underline (e.g. in the HeatFunctional constructor: Scalar q_)
- get/set methods names must be named getXYZ() and setXYZ()
- Braces must go on a new line and not indented
- Namespaces must be indented
- Use sensible/comprehensive variable/class/function names (e.g. int refinements, class DomainCache, void evaluateAt(...))
- Comment your program for yourself to understand it, when you look at it at a later time (one year after programming) and for other users to easily comprehend and use it. Adhere to the following documentation suggestions:
  - Write documentation for a more or less experienced user. Assume a good understanding of common C++ features and idioms, but do not omit documentation when using more arcane language features. Assume a good general understanding of numerical mathematics and finite elements, but do not assume in-depth knowledge of every details

of, say, error estimators or flux limiters. Apply **common sense** when deciding what and how to document.

- Use doxygen to document the *interface* of classes, methods, and free functions. Prefer a semantic documentation: tell what a class/method-/function does on a mathematical or abstract level, if appropriate with formulas.
- For classes, document on a top level view what the class (including its methods) is supposed to do, what's it good for, in which situations it should be used. State requirements of template parameters (use the `\tparam` tag). If appropriate, state class invariants and memory consumption. In complex settings, consider providing code examples for class usage.
- For methods and functions, document on a more detailed level what they are supposed to do, and which assumptions the parameters have to satisfy (preconditions). Describe what guarantees for the return values or output parameters hold (postconditions). If appropriate, state runtime complexity. Relate this to the class documentation.
- Use plain comments to document the *implementation*. Prefer a semantic documentation here, too. State design and implementation decisions.

## 10.2 Measurement of cpu time

We use the **boost** package **timer** to perform measurements of user time, wall-clock time, and system time. For that, we have to include

```
#include <boost/timer/timer.hpp>
```

We start such a time measurement by adding a line like the following:

```
boost::timer::cpu_timer timer;
```

where `timer` stands for an arbitrary name chosen by the user. The time elapsed since defining the timer is given by

```
(double)(timer.elapsed().user)/1e9;
(double)(timer.elapsed().wall)/1e9;
```

with scaling factor $1e9$.

For example, we measure the user time for the assembly in the Stokes problem, see 6.8:

```
boost::timer::cpu_timer timer;
assembler.assemble(linearization(F,x));
std::cout << (double)(timer.elapsed().user)/1e9 << "s\n";
```

The timer function `format()` prints wall clock time, user time, and system time and some additional information, e.g. in the example above

```
boost::timer::cpu_timer timer;
assembler.assemble(linearization(F,x));
std::cout << "% -- " << totalTimer.format() << "s\n";
```

yields the following output:

```
% --  0.615291s wall, 0.460000s user + 0.150000s system = 0.610000s CPU (99.1%)
```

## 10.3 Namespaces

## 10.4 Multithreading

## 10.5 Special aspects of the DUNE grid interface

As already mentioned KASKADE7 is based on the DUNE core modules. The DUNE grid interface enables the separation of the finite element code from the grid implementation. Furthermore, the DUNE grid module provides several grid implementations. You can consult the class documentation and tutorials on the DUNE webpage [2] for questions concerning the usage of the grid interface. Nevertheless, some approaches for specific issues are not that easy to find out. Therefore, we will explain them here.

The following example shows how the grid interface can be employed for finding the vertices (their global indices) of an (boundary) face. In this application we have boundary segments with different properties and we want to find all segments with a certain property and mark the corresponding vertices. First we present the code snippet which achieves this an then explain the details.

```
1 const int dim = 3;
2 using Grid = Dune::UGGrid<dim>;
3 std::vector<int> propertiesOfBoundarySegments;
4 std::vector<int> propertiesOfElements; // not further used
5 std::unique_ptr<Grid> gridPtr(Dune::GmshReader<Grid>::read("grid/
      coarseCube.msh",propertiesOfBoundarySegments,
      propertiesOfElements));
6 GridManager<Grid> gridManager(std::move(gridPtr));
7
8 Grid::LeafGridView leafGridView = gridManager.grid().leafGridView
      ();
```

```
9  Dune::MultipleCodimMultipleGeomTypeMapper<Grid::LeafGridView,Dune
       ::MCMGVertexLayout> vertexMapper(leafGridView);
10 Dune::BitSetVector<1> dirichletNodes(leafGridView.size(dim),false)
       ;
11 for (const auto& element : Dune::elements(leafGridView)) {
12   const Dune::ReferenceElement<double,dim>& refElement = Dune::
         ReferenceElements<double, dim>::general(element.type());
13   if(element.hasBoundaryIntersections()) {
14     for(const auto& intersection : Dune::intersections(
           leafGridView,element)) {
15       if(intersection.boundary()) {
16         if(propertiesOfBoundarySegments[intersection.
             boundarySegmentIndex()]==1) { // has desired property
             value?
17           int localFaceIndex = intersection.indexInInside();
18           int nvFace = refElement.size(localFaceIndex, 1, dim);
19           for (int i=0; i<nvFace; i++) {
20             int localVertexIndex = refElement.subEntity(
                 localFaceIndex, 1, i, dim);
21             dirichletNodes[vertexMapper.subIndex(element,
                 localVertexIndex,dim)] = true;
22           }
23         }
24       }
25     }
26   }
27 }
```

In lines 1 to 10 needed variables are defined. In particular, in line 5 the grid (and boundary segments together with their property) is read from a gmsh-file in which the positions of the vertices, the elements, the boundary segments and their properties are recorded. Since the grid interface guarantees that the boundary segments are ordered in the same way as they are inserted, we can later query the index of a boundary segment and then get its property by the vector propertiesOfBoundarySegments . The purpose of the for-loops in lines 11 and 14 is to iterate over all boundary segments (which is an intersection located at the boundary). The member function boundarySegmentIndex() of the class Intersection provides the index of the boundary segment and we can check whether it has a certain property (line 16). By the local index of the boundary segment (as it is a face of an element) and with the aid of the reference element (which was defined in line 12) we obtain the local indices of the vertices of the segment (line 20). By a suitable mapper we then get the global indices and can mark the vertices (line 21).

# 11 Details in C++ implementation

## 11.1 Smart pointers (`std::unique_ptr`)

The class `std::unique_ptr` defines a sort of smart pointer. Unique pointers make sure that the object they refence to is deleted when the life of the pointer ends. Additionally only one unique pointer can hold a reference, which allows to explicitly express ownership of objects on the heap.

In our first example (see Section 6.2) the ownership of the `new Grid` is transfered form `grid` to `gridManager` on line 4. The value of `grid` is zero after this takeover.

```
int const dim = 2;
typedef Dune::SGrid<dim,dim> Grid
std::unique_ptr<Grid> grid(new Grid);
GridManager<Grid> gridManager(std::move(grid));
```

## 11.2 Fusion vectors (`boost::fusion::vector`)

The `boost::fusion::vector` template allows the definiton of container classes with elements of different types. In our example (Stokes) the finite element $H1$-spaces for the pressure and velocity have different order and are collected in the `Spaces` vector.

```
H1Space pressureSpace(gridManager,gridManager.grid().leafIndexSet
    (),ord-1);
H1Space velocitySpace(gridManager,gridManager.grid().leafIndexSet
    (),ord);

typedef vector<H1Space const*,H1Space const*> Spaces;
Spaces spaces(&pressureSpace,&velocitySpace);
```

The elements are accessed by the `at_c`-template or an iterative template like `foreach`.

```
class PrintSpace
{
    template<typename T>
    void operator()(T& t) const
```

```
    {
        std::cout << t << std::endl;
    }
};
std::cout << at_c<0>(spaces) << std::endl;
std::cout << at_c<1>(spaces) << std::endl;
// or equivalent
foreach(spaces, PrintSpace);
```

# 12 The dos and don'ts – best practice

- Place using-directives as locally as possible. Especially in header files you should never place them at global scope or in namespaces, as then the scope of the using-directive will extend to all files included after this header file. Therefore this clutters the global namespace and foils the idea behind namespaces and sooner or later will lead to errors that are difficult to find.

- Be careful with committing changes to the SVN repository. Especially, when you have modified header files, ensure that the commands "make install" and "make tutorial" still complete without errors before committing your changes.

- When iterating e.g. over cells, create the end-iterator *outside* the `for` statement:

  ```
  auto itEnd = gridView.end<0>();
  for(auto it = gridView.begin<0>(); it != itEnd; ++it)
      ...
  ```

  Creating the end iterator incurs a small but often non-negligible cost. Current compilers (as of 2015-03-11) appear not to be smart enough to optimize the repeated creation of the end iterator in the test of the for loop away. Hence obtain it once beforehand.

- Avoid copying entities or entity pointers whenever possible. Entites, and – depending on the grid implementation – even entity pointers, may be large objects, and copying/creating them is expensive.

- Objects like `Geometry` are returned by reference, and – again depending on the grid used – may be created on demand, which is expensive. Store references that are used more than one time:

```
auto const& geometry = cell.geometry();
for( int i = 0; i < dim+1; i++ )
   std::cout << geometry.corner(i) << std::endl;
```

# 13 Modules

This section briefly describes a couple of modules that provide additional functionality or convenience routines, but are not considered to be part of the core KASKADE7 libraries, e.g., because their applicability is limited to a quite narrow problem domain.

## 13.1 Differential operator library

Defining problems as in 6.9 can be a tedious and error-prone task. Consider an optimization problem subject to linear elasticity constraints: the Lamé-Navier operator appears twice (once in the constraint and once in the adjoint equation) and has to be coded anew, even though linear elasticity probably has been implemented already. For this reason, a couple of standard differential operators are predefined in `fem/diffops/`. We will present the (unspectacular) definition of the Poisson equation $-\Delta u = f$ in terms of the predefined Laplace differential operator class just to show the principle. There is, of course, no real improvement, but this is different for more complex differential operators, such as linear elastomechanics or the Stokes problem.

```
#include "fem/diffops/laplace.hh"

template <class Scalar, class VarSet>
class HeatFunctional
{
public:
  typedef VarSet OriginVars;
  typedef VarSet AnsatzVars;
  typedef VarSet TestVars;

  class DomainCache
  {
  public:
    template <class Entity>
    void moveTo(Entity const &)
    {
      laplace.setDiffusionTensor(1.0);
    }
```

```
  template <class Position, class Evaluators>
  void evaluateAt(Position const& x, Evaluators const&
      evaluators)
  {
    using namespace boost::fusion;
    int const uIdx = result_of::value_at_c<typename OriginVars::
        Variables,0>::type::spaceIndex;

    u  = component<0>(data).value(at_c<uIdx>(evaluators));
    du = component<0>(data).derivative(at_c<uIdx>(evaluators))
        [0];
    laplace.setLinearizationPoint(du);
    f = 1.0;
  }

  Scalar
  d0() const
  {
    return laplace.d0() - f*u;
  }

  template<int row, int dim>
  Dune::FieldVector<Scalar, TestVars::template Components<row>::
      m>
  d1 (VariationalArg<Scalar,dim> const& arg) const
  {
    return laplace.d1(arg)-f*arg.value;
  }

  template<int row, int col, int dim>
  Dune::FieldMatrix<Scalar, TestVars::template Components<row>::
      m, AnsatzVars::template Components<col>::m>
  d2 (VariationalArg<Scalar,dim> const &argTest, VariationalArg<
      Scalar,dim> const& argAnsatz) const
  {
    return laplace.d2(argTest,argAnsatz);
  }

private:
  Scalar u, f;
  Dune::FieldVector<Scalar,AnsatzVars::Grid::dimension> du;
  Kaskade::Laplace<Scalar,AnsatzVars::Grid::dimension> laplace;
};
```

## 13.2 Deforming grid manager

### 13.2.1 Motivation

In finite element computations the discretization of the spatial domain $\Omega$ influences the accuracy of the solution and the efficiency of numerical solvers in many ways. E.. the discretization $\Omega_h$ of $\Omega$ contains artificial corners. The possibly occuring corner singularities at these artificial corners lead to unnecessary adaptive refinement, thus reducing the efficiency of numerical solvers. Moreover the boundary $\partial\Omega_h$ does in general not coincide with the boundary $\partial\Omega$. The same holds for inner boundaries in the case of multi-phase problems. Thus the position of the imposed boundary conditions is resolved inexactly as well as the orientation of the surface's normal vectors.

Both effects may be reduced if more information than the coarsest discretization $\Omega_{h_c}$ is available. In many cases there exists a finer discretization $\Omega_{h_f}$, which has been coarsened in order to efficiently solve the problem. If this is not the case it is also possible to use knowledge on tangent plane continuity, possibly of subsets, of $\partial\Omega$. The implemented `DeformingGridManager` is able to

- consider inner boundaries if phase information is provided
- adjust/smoothen (parts of) the inner and/or outer boundaries
- detect and preserve "sharp" features
- control angle conditions.

### 13.2.2 Getting started

The class `DeformingGridManager` is implemented using a policy based class design (implementations mentioned below can be found in `namespace Policy`). The following three policies have to be specified:

- `OuterBoundaryPolicy`: Determines the behaviour on domain boundary. Choose one of:
  - `ConsiderOuterBoundary`: Consider complete domain boundary (default).
  - `IgnoreOuterBoundary`: Ignore complete domain boundary.
  - `SpecifyOuterBoundary`: Specify particular parts of the outer boundary that shall be considered/ignored. The parts that shall be considered/ignored are identified via their `boundary segment index`. In general the specified ids will be considered and the remaining ids

will be ignored.
If `skipSpecifiedIds==true` it is the other way round.

If the given policies do not satisfy your needs you may implement your own policy. Then your policy must implement the method:

```
template<class Face> bool ignoreOuterBoundary(Face const&
    face) const;
```

- `InnerBoundaryPolicy`: Determines the behaviour on inner boundaries. Note that `InnerBoundaryPolicy!=IgnoreInnerBoundary` does only make sense if a phase element is passed to the constructor of `DeformingGridManager`, i.e. inner boundaries only exist if phase information has been provided. Choose one of:
  - `ConsiderInnerBoundary`: Consider all inner boundaries.
  - `IgnoreInnerBoundary`: Ignore all inner boundaries (default).
  - `SpecifyInnerBoundary`: Specify particular parts of the inner boundaries that shall be considered/ignored. The parts that shall be considered/ignored are identified via a pair of phase ids associated with the neighbouring phases. In general the specified pairs will be considered and the remaining pairs will be ignored.
    If `skipSpecifiedIds==true` it is the other way round.

  If the given policies do not satisfy your needs you may implement your own policy. Then your policy must implement the method:

```
bool ignoreInnerBoundary(int phaseId, int neighbourId) const;
```

- `ThresholdPolicy<Scalar>`: Feature detection. Choose one of:
  - `NoGradientThreshold`: No feature detection (default).
  - `ApplyGradientThreshold`: Simple feature detection. Considering the provided or computed surface normals in local 2-dimensional coordinate systems leads to a scalar quantity, the slope in the local coordinate system. This policy allows to specify a threshold for this slope. If the threshold is exceeded it is assumed that we detected a sharp corner that should not be smoothened. Note that $slope = 1$ is equivalent to an angle of $2 \cdot 45^\circ = 90^\circ$.

If the given policies do not satisfy your needs you may implement your own policy. Then your policy must implement the method:

```
void applyGradientThreshold(Scalar &gradient) const;
```

**Usage:** In order to use the deforming grid manager include the file "fem/deforminggridmanager.hh" instead of "fem/gridmanager.hh". Then you may start using this grid manager implementation using the following `typedef`:

```
typedef DeformingGridManager < double ,UGGrid<3>,Policy ::
    IgnoreOuterBoundary , Policy::ConsiderInnerBoundary , Policy::
    ApplyGradientThreshold > GridManager;
```

Supposing that there exists a grid stored in a `std::unique_ptr` and a phase element `phaseElement` (a `FunctionSpaceElement` holding the phase ids) we can create an object of `GridManager`:

```
using namespace Policy;
GridManager gridManager(std::move(grid), phaseElement,
    IgnoreOuterBoundary(), ConsiderInnerBoundary(),
    ApplyGradientThreshold(0.71));
```

**Remark:** As it seems not to be possible to determine adequate a priori estimates that guarantee that angle conditions are not violated during mesh refinement, aa posteriori verification step has been implemented. In this step the deformation that is associated with the specified policies is damped as long as degradation of mesh regularity is considered not acceptable. It may be better to move this damping step to another policy, but currently you can not change it.

### 13.2.3 Advanced usage

# 14 Gallery of projects

# 15 KASKADE7 publications

KASKADE7 provided numerical simulation results for many articles and two books. Here we present the most important of them:

- Peter Deuflhard, Martin Weiser:
  Numerische Mathematik 3. Adaptive Lösung partieller Differentialgleichungen,
  de Gruyter, 2011.

- Peter Deuflhard, Martin Weiser:
  Adaptive numerical solution of PDEs,
  de Gruyter, to appear 2012.

- Olaf Schenk, Andreas Wächter, Martin Weiser:
  Inertia revealing preconditioning for large-scale nonconvex constrained optimization,
  SIAM J. Sci. Comp. 31(2): 939-960(2008).

- Martin Weiser:
  Optimization and Identification in Regional Hyperthermia,
  Int. J. Appl. Electromagn. and Mech. 30: 265-275(2009).

- Martin Weiser:
  Pointwise Nonlinear Scaling for Reaction-Diffusion Equations,
  Appl. Num. Math. 59 (8): 1858-1869(2009).

- Anton Schiela, Andreas Günther:
  An interior point algorithm with inexact step computation in function space for state constrained optimal control,
  Num. Math. 119(2): 373-407(2011), see also ZIB Report 09-01 (2009).

- Martin Weiser:
  On goal–oriented adaptivity for elliptic optimal control problems,
  to appear in Opt. Meth. Softw., see also ZIB-Report 09-08 (2009).

- Anton Schiela, Martin Weiser:
  Barrier methods for a control problem from hyperthermia treatment planning,
  in M. Diehl, F. Glineur, E. Jarlebring, W. Michiels (eds.): Recent Advances in Optimization and its Applications in Engineering (Proceedings of 14th Belgian-French-German Conference on Optimization 2009), 419-428(2010), Springer, see also ZIB-Report 09-36 (2009).

- Martin Weiser, Sebastian Götschel:
  State trajectory compression for optimal control with parabolic PDEs,
  SIAM J. Sci. Comp., 34(1):A161-A184(2012), see also ZIB Report 10-05 (2010).

- Sebastian Götschel, Martin Weiser, Anton Schiela:
  Solving optimal control problems with the Kaskade7 Finite Element Toolbox,

in Dedner, Flemisch, Klöfkorn (Eds.) Advances in DUNE, 101-112(2012), Springer, see also ZIB Report 10-25 (2010).

- Lars Lubkoll:
  Optimal control in implant shape design,
  thesis, TU Berlin 2011.

- Peter Deuflhard, Anton Schiela, Martin Weiser:
  Mathematical cancer therapy planning in deep regional hyperthermia,
  Acta Numerica 21: 307-378(2012), see also ZIB-Report 11-39 (2011).

- Lars Lubkoll, Anton Schiela, Martin Weiser:
  An optimal control problem in polyconvex hyperelasticity,
  ZIB-Report 12-08 (2012).

# A Weak formulations

## A.1 Stationary heat equation

$$\int_\Omega \kappa \nabla u \cdot \nabla v \, \mathrm{d}x + \int_\Omega quv \, \mathrm{d}x = \int_\Omega fv \, \mathrm{d}x \tag{38}$$

# B Online resources

## B.1 Git

- `https://git-scm.com/docs` common used commands in Git
- `https://git-scm.com/docs/git#_git_commands` complete list of Git commands
- `http://git.or.cz/course/svn.html` crash course for switching from SVN to Git
- `http://rogerdudler.github.io/git-guide/` short starting guide for Git
- `https://git-scm.com/book/en/v2` the entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress

# C Libraries

## ALBERTA

ALBERTA is an **A**daptive multi**L**evel finite element toolbox using **B**isectioning refinement and **E**rror control by **R**esidual **T**echniques for scientific **A**pplications. DUNE offers an abstraction to the ALBERTA grid manager through the AlbertaGrid class.

- Homepage: `http://www.alberta-fem.de/`
- Currently developed by University of Stuttgart, Institute of Applied Analysis and Numerical Simulation: `https://www.ians.uni-stuttgart.de`

## AMIRA

Amira is a software platform for 3D and 4D data visualization, processing, and analysis. It is being actively developed by Thermo Fisher Scientific in collaboration with the Zuse Institute Berlin (ZIB), and commercially distributed by Thermo Fisher Scientific.

Amira remains the development platform for many research projects at the Department of Visual Data Analysis at ZIB. For this purpose, ZIB maintains a version of Amira for Research Partners. Research partners are individuals who actively participate in a joint research project with ZIB aiming at joint publications, joint software development, and/or joint funding. ZIB can provide Amira for Research Partners for the duration of the project on an individual basis free of charge.

- Research partners obtain the software and a license as follows: `https://amira.zib.de/download.html`
- Amira at Thermo Fisher Scientific: `https://www.thermofisher.com/de/de/home/industrial/electron-microscopy/electron-microscopy-instruments-workflow-solutions/3d-visualization-analysis-software/amira-life-sciences-biomedical.html`
- Amira at Wikipedia: `https://en.wikipedia.org/wiki/Amira_(software)`

## BOOST

Boost is a set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing.

- Homepage: `https://www.boost.org/`
- Boost at Wikipedia: `https://en.wikipedia.org/wiki/Boost_(C%2B%2B_libraries)`

## DUNE

DUNE, the **D**istributed and **U**nified **N**umerics **E**nvironment is a modular toolbox for solving partial differential equations (PDEs) with grid-based methods. It supports the easy implementation of methods like Finite Elements (FE), Finite Volumes (FV), and also Finite Differences (FD).

DUNE is a community project. It is free software licensed under the GPL (version 2) with a so called "runtime exception" (see license). This licence is similar to the one under which the libstdc++ libraries are distributed. Thus it is possible to use DUNE even in proprietary software.

The underlying idea of DUNE is to create slim interfaces allowing an efficient use of legacy and/or new libraries. Modern C++ programming techniques enable very different implementations of the same concept using a common interface at a very low overhead. Thus DUNE ensures efficiency in scientific computations and supports high-performance computing applications.

You will find a class documentation for DUNE core modules of several versions at their homepage.

- Homepage: `https://dune-project.org/`

## HYPRE

Livermore's HYPRE library of linear solvers makes possible larger, more detailed simulations by solving problems faster than traditional methods at large scales. It offers a comprehensive suite of scalable solvers for large-scale scientific simulation, featuring parallel multigrid methods for both structured and unstructured grid problems. The HYPRE library is highly portable and supports a number of languages.

HYPRE is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License (as published by the Free Software Foundation) version 2.1 dated February 1999.

- Homepage: `https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods`

### ITSOL

ITSOL is a library of iterative solvers for general sparse linear systems of equations. ITSOL can be viewed as an extension of the itsol module in SPARSKIT. It is written in C and offers a selection of recently developed preconditioners.

- Homepage: `https://www-users.cs.umn.edu/~saad/software/ITSOL/`

### MUMPS

MUMPS (**MU**ltifrontal **M**assively **P**arallel **S**olver) is a package for solving systems of linear equations of the form $Ax = b$, where $A$ is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric, on distributed memory computers. MUMPS implements a direct method based on a multifrontal approach which performs a Gaussian factorization $A = LU$ where $L$ is a lower triangular matrix and $U$ an upper triangular matrix. If the matrix is symmetric then the factorization $A = LDL^T$ where $D$ is block diagonal matrix with blocks of order 1 or 2 on the diagonal is performed.

- Homepage: `http://mumps.enseeiht.fr/`

### PARDISO

The package PARDISO is a thread-safe, high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared-memory and distributed-memory multiprocessors.

- Homepage: `https://pardiso-project.org/`

### TAUCS

TAUCS is a C library of sparse linear solvers. The TAUCS incomplete Cholesky preconditioner is a more memory-efficient version of the incomplete LU preconditioner for symmetric positive definite matrices.

- Homepage: `https://www.tau.ac.il/~stoledo/taucs/`

### UG for DUNE

UG is a software tool for the numerical solution of partial differential equations on unstructured meshes in two and three space dimensions using multigrid methods. UG runs both sequentially and in parallel. DUNE offers an abstraction to the UG grid manager through the UGGrid class. UG is free software, available under the LGPLv2+.

- Information about UG at DUNE homepage: `https://dune-project.org/doc/install-ug/`

### UMFPACK

UMFPACK is a set of routines for solving unsymmetric sparse linear systems, $Ax = b$, using the **U**nsymmetric **M**ulti**F**rontal method. It uses dynamic memory allocation, and has a symbolic preordering and analysis phase that also reports the upper bounds on the nonzeros in $L$ and $U$, flop count, and memory usage in the numeric phase. It can be used for real and complex matrices, rectangular and square, and both non-singular and singular. It is included in SuiteSparse, a suite of sparse matrix algorithms.

- Homepage of SuiteSparse: `https://people.engr.tamu.edu/davis/suitesparse.html`

## References


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

[2] *DUNE, the Distributed and Unified Numerics Environment*. `http://www.dune-project.org/`.

[3] *UG, software package for solving partial differential equations*. `http://atlas.gcsc.uni-frankfurt.de/~ug/index.html`.

[4] Ivo Babuška. The finite element method with penalty. *Mathematics of computation*, 27(122):221–228, 1973.

[5] P. Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, R. Klöfkorn, R. Kornhuber, Mario Ohlberger, and Oliver Sander. A generic grid interface for parallel and adaptive scientific computing. part ii: Implementation and tests in dune. *Computing*, 82:121–138, 01 2008.

[6] Peter Bastian, Markus Blatt, Andreas Dedner, Christian Engwer, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. A generic grid interface for parallel and adaptive scientific computing. part i: Abstract framework. *Computing*, 82:103–119, 07 2008.

[7] Markus Blatt and Peter Bastian. The iterative solver template library. 06 2006.

[8] Peter Deuflhard. *Newton methods for nonlinear problems: affine invariance and adaptive algorithms*, volume 35. Springer Science & Business Media, 2011.

[9] Peter Deuflhard, Peter Leinen, and Harry Yserentant. Concepts of an adaptive hierarchical finite element code. *IMPACT of Computing in Science and Engineering*, 1(1):3–35, 1989.

[10] Peter Deuflhard and Ulrich Nowak. Extrapolation integrators for quasilinear implicit odes. In *Large Scale Scientific Computing*, pages 37–50. Springer, 1987.

[11] Michael Hinze, René Pinnau, Michael Ulbrich, and Stefan Ulbrich. *Optimization with PDE constraints*, volume 23. Springer Science & Business Media, 2008.

[12] Anders Logg. Automating the finite element method. *Archives of Computational Methods in Engineering*, 14(2):93–138, 2007.

[13] Bertrand Maury. Numerical analysis of a finite element/volume penalty method. *SIAM Journal on Numerical Analysis*, 47(2):1126–1148, 2009.

[14] Todd Veldhuizen. Using c++ template metaprograms. In *C++ gems*, pages 459–473. SIGS Publications, Inc., 1996.

[15] Martin Weiser, Tobias Gänzler, and Anton Schiela. A control reduced primal interior point method for a class of control constrained optimal control problems. *Computational Optimization and Applications*, 41(1):127–145, 2008.

[16] Gerhard W Zumbusch. Symmetric hierarchical polynomials and the adaptive hp-version. 1995.

# Index