

AMDiS tutorial

Simon Vey, Thomas Witkowski

February 22, 2010

Contents

1	Introduction	5
2	Installation	7
2.1	Installation of the AMDiS library	7
2.2	Compilation of an example application	8
3	Application makefile	11
4	Implementation of example problems	15
4.1	Stationary problem with Dirichlet boundary condition	15
4.1.1	Source code	16
4.1.2	Parameter file	18
4.1.3	Macro file	19
4.1.4	Output	21
4.2	Time dependent problem	21
4.2.1	Source code	22
4.2.2	Parameter file	26
4.2.3	Macro file	27
4.2.4	Output	28
4.3	Systems of PDEs	28
4.3.1	Source code	28
4.3.2	Parameter file	30
4.3.3	Macro file	30
4.3.4	Output	31
4.4	Coupled problems	31
4.4.1	Source code	32
4.4.2	Parameter file	36
4.4.3	Macro file	37
4.4.4	Output	37
4.5	Nonlinear problem	37
4.5.1	Source code	38
4.5.2	Parameter file	45
4.5.3	Macro file	45
4.5.4	Output	45
4.6	Neumann boundary conditions	46
4.6.1	Source code	46
4.6.2	Parameter file	47
4.6.3	Macro file	47
4.6.4	Output	47
4.7	Periodic boundary conditions	48
4.7.1	Source code	49
4.7.2	Parameter file	50

4.7.3	Macro file	50
4.7.4	Output	52
4.8	Projections	52
4.8.1	Source code	53
4.8.2	Parameter file	54
4.8.3	Macro file	56
4.8.4	Output	58
4.9	Parametric elements	58
4.9.1	Source code	59
4.9.2	Parameter file	63
4.9.3	Macro file	64
4.9.4	Output	64

Chapter 1

Introduction

The objective of this tutorial is to introduce the user into the main AMDiS features by giving some application examples.

Section 2 describes the installation of the AMDiS library and the building of user applications step by step. The corresponding application makefile is given in Section 3.

In Section 4, for every example the following aspects are described:

- **Abstract problem description:** In the header of each example section, the abstract problem definition is given. Sometimes, some solution strategies on a high abstraction level are mentioned, also.
- **Source code:** In the source code section, the listing of the example source code is explained.
- **Parameter file:** In this section, the parameter file is described. The parameter file contains parameters which are read by the application at runtime. The name of the parameter file is usually passed to the application as a command line argument.
- **Macro file:** In the macro file section, the definition of the coarse macro mesh is shown, which is the basis for adaptive refinements.
- **Output:** The AMDiS results are written to output files that contain the final mesh and the problem solution on this mesh. The output can be visualized by proper tools (*CrystalClear*, *ParaView*, *TecPlot*). In the output section, the visualized problem results are shown and discussed.

To avoid unnecessary repetitions, not every aspect of every example is described, but only those aspects that have not appeared in previous examples.

Chapter 2

Installation

2.1 Installation of the AMDiS library

To install the AMDiS library, the following steps must be performed:

1. Create the AMDiS source directory. This can be done in different ways. Here, we show three possibilities:

- Unpacking the archive file `AMDiS.tgz`:
– `> tar xvfz AMDiS.tgz`
- Checking out an SVN project:
– `> svn checkout --username developername
https://gforge.zih.tu-dresden.de/svn/amdis/trunk amdis`

2. Change into the AMDiS directory:

```
> cd amdis/AMDiS
```

3. Create the makefiles for your system using the `configure` script:

```
> ./configure <CONFIGURE-OPTIONS>
```

The `configure` script creates the needed makefiles. It can be called with the following options:

- `--prefix=<AMDIS-DIR>`: Installation path of the AMDiS library. The library file will be stored in `<AMDIS-DIR>/lib`. With `--prefix='pwd'` AMDiS will be installed in the current working directory, which mostly is a good choice.
- `--enable-debug`: If this option is used, AMDiS will be compiled with debug support. By default, AMDiS is compiled in an optimized mode without debug support.
- `--enable-umfpack`: If this option is used, AMDiS is compiled with support for the UMF-PACK solver library.
- `--with-mpi=<MPI-DIR>`: MPI installation path. Used for parallelization support.
- `--with-parmetis=<PARMETIS-DIR>`: ParMETIS installation path. ParMETIS is a parallel graph partitioning library, see <http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview>. Used for parallelization support.

If AMDiS should be compiled for parallel usage, the MPI and ParMETIS paths must be set.

4. Make the library:

```
> make install
```

If you have added a new source file or you want to change something on the automake-system, you have to rerun the following commands:

```
> libtoolize --force --copy
> aclocal
> autoconf
> automake --add-missing --copy
```

Then repeat steps 3 and 4. For the additional steps `libtool`, `automake` and `autotools` must be installed on your system.

Further information about the installation process can be found in the `README` file in the AMDiS source directory.

2.2 Compilation of an example application

For the compilation of the examples, described in this section, the following steps must be executed:

1. Get the sources:

- Unpack an archive file:

```
> gunzip demo.tar.gz
> tar xvf demo.tar
```

or
- CVS checkout:

```
> export CVSROOT=<CVSROOT>
> cvs checkout demo
```

or
- SVN checkout:

```
> svn checkout file://<SVN-REPOSITORY-PATH>/demo
```

2. Change into the demo directory:

```
> cd demo
```

3. Edit the Makefile:

- Set the AMDiS path and paths of other needed libraries.
- Set user flags.

The makefile is described in Section 3 in detail.

4. Make the application example:

```
> make <PROG-NAME>
```

`<PROG-NAME>` is the name of the application example.

To run the example, call:

- In the sequential case:

```
> ./<PROG-NAME> <PARAMETER-FILE>
```


- In the parallel case:
> mpirun <MPI-OPTIONS> ./<PROG-NAME> <PARAMETER-FILE>

The <MPI-OPTIONS> at least should contain the number of used processes, which is given by `-np <NUM-PROCS>`. For further MPI options see <http://www-unix.mcs.anl.gov/mpi/>.

Chapter 3

Application makefile

In this section, the organization of the application makefile is described which is used for the examples in this tutorial. The same organization can be used for other user applications, too.

In the first block, user flags and directories are specified.

```
# =====
# ===== flags and directories (to be modified by the user) =====
# =====

USE_PARALLEL_AMDIS = 0          # 0: sequential AMDiS, 1: parallel AMDiS
DEBUG               = 0          # 0: no debug, 1: debug mode

AMDIS_DIR           = <AMDIS-DIR> # fill the AMDiS installation path here
MPI_DIR             = <MPI-DIR>    # fill the MPI installation path here
PARMETIS_DIR        = <PARMETIS-DIR> # fill the ParMETIS installation path here
```

If USE_PARALLEL_AMDIS is set to 1, parallel applications will be supported. A necessary condition is that the AMDiS library is configured for parallelization, too (see Chapter 2). The DEBUG entry specifies, whether applications should be compiled in debug mode, or not. This entry is independent of the corresponding AMDiS settings, but if the AMDiS library was not compiled in debug mode, only application code can be debugged.

AMDIS_DIR stores the AMDiS installation path and must be set by the user. This is the path given to the AMDiS configure script by the --prefix option. The values of MPI_DIR and PARMETIS_DIR only are needed if parallelization should be supported. Here, the installation pathes of MPI and ParMETIS are stored.

In the next block, include pathes are defined.

```
# =====
# ===== includes pathes =====
# =====

AMDIS_INCLUDE       = -I$(AMDIS_DIR)/src
MPI_INCLUDE          = -I$(MPI_DIR)/include
PARMETIS_INCLUDE    = -I$(PARMETIS_DIR)

INCLUDES = -I. $(AMDIS_INCLUDE) $(MPI_INCLUDE) $(PARMETIS_INCLUDE)
```

Now, we introduce the needed libraries.

```
# =====
```

```
# ===== libraries =====
# =====

AMDIS_LIB    = -L$(AMDIS_DIR)/lib -lamdis
PARMETIS_LIB = -L$(PARMETIS_DIR) -lparmetis -lmetis

LIBS = $(AMDIS_LIB)
```

By default, LIBS contains only the AMDiS library. If USE_PARALLEL_AMDIS is 1, LIBS is extended by the ParMETIS library. In the same way, other libraries can be added. In the sequential case, we use the GNU C++ compiler `g++`, in the parallel case, the MPI C++ compiler `mpiCC`.

```
# =====
# ===== parallel or sequential ? =====
# =====

ifeq ($(USE_PARALLEL_AMDIS), 0)
    COMPILE = g++
else
    COMPILE = $(MPI_DIR)/bin/mpiCC
    LIBS += $(PARMETIS_LIB)
endif
```

The next block sets the compile flags. In debug mode, we use no optimization (`-O0`) and add symbolic debug information (`-g`). Otherwise, we compile with optimization level 2 (`-O2`).

```
# =====
# ===== compile flags =====
# =====

ifeq ($(DEBUG), 0)
    CPPFLAGS = -O2
else
    CPPFLAGS = -g -O0
endif
```

We use the `libtool` in the AMDiS installation path for linking.

```
# =====
# ===== libtool linking =====
# =====

LIBTOOL = $(AMDIS_DIR)/libtool
LINK = $(LIBTOOL) --mode=link $(COMPILE)
```

Now, we define rules to create and delete objects files.

```
# =====
# ===== rules =====
# =====

clean:
    -rm -rf *.o

.cc.o: *.cc
    $(COMPILE) $(INCLUDES) $(CPPFLAGS) -c -o $*.o $^
```

The second rule creates needed object files automatically using the corresponding C++ files.

Finally, we define rules for the linking of user applications. Here, we present only the rule for the `ellipt` application. Other applications can be created in an analog way.

```
# =====
# ===== user programs =====
# =====

VPATH = ../src:${AMDIS_DIR}/src

# ===== myprog =====

ELLIPT_OFILES = ellipt.o

ellipt: $(ELLIPT_OFILES)
    $(LINK) $(CPPFLAGS) -o ellipt $(ELLIPT_OFILES) $(LIBS)
```

The `VPATH` variable contains all pathes, where sources can be located. The `ellipt` rule first creates all needed object files defined in `ELLIPT_OFILES`. In this example, only `ellipt.o` is needed. Then all needed object files and libraries are linked together. The `-o` option specifies that the executable will be written to the file `ellipt`.

Chapter 4

Implementation of example problems

4.1 Stationary problem with Dirichlet boundary condition

As example for a stationary problem, we choose the Poisson equation

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \quad (4.1)$$

$$u = g \quad \text{on } \partial\Omega \quad (4.2)$$

with

$$f(x) = -(400x^2 - 20dow) e^{-10x^2} \quad (4.3)$$

$$g(x) = e^{-10x^2}. \quad (4.4)$$

dim is the problem dimension and thus the dimension of the mesh the problem will be discretized on. dow is the dimension of the world the problem lives in. So world coordinates are always real valued vectors of size dow . Note that the problem is defined in a dimension independent way. Furthermore, dow has not to be equal to dim as long as $1 \leq dim \leq dow \leq 3$ holds.

Although the implementation described in Section 4.1.1 is dimension independent, we focus on the case $dim = dow = 2$ for the rest of this section. The analytical solution on $\Omega = [0, 1] \times [0, 1]$ is shown in Figure 4.1.

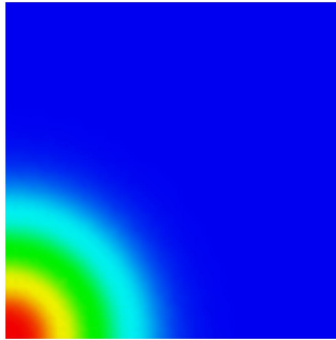


Figure 4.1: Solution of the Poisson equation on the unit square.

		1d	2d	3d
source code	src/	ellipt.cc		
parameter file	init/	ellipt.dat.1d	ellipt.dat.2d	ellipt.dat.3d
macro file	macro/	macro.stand.1d	macro.stand.2d	macro.stand.3d
output files	output/	ellipt.mesh, ellipt.dat		

Table 4.1: Files of the `ellipt` example.

4.1.1 Source code

For this first example, we give the complete source code. But to avoid loosing the overview, we sometimes interrupt the code to give some explaining comment. The first three lines of the application code are:

```
#include "AMDIS.h"
using namespace AMDIS;
using namespace std;
```

In the first line, the `AMDIS` header is included. In line 2 and 3, used namespaces are introduced. `std` is the C++ standard library namespace, used e.g. for the STL classes. `AMDIS` provides its own namespace `AMDIS` to avoid potential naming conflicts with other libraries.

Now, the functions f and g will be defined by the classes `F` and `G`:

```
class G : public AbstractFunction<double, WorldVector<double>>
{
public:
    double operator()(const WorldVector<double>& x) const
    {
        return exp(-10.0 * (x * x));
    }
};
```

`G` is a sub class of the templated class `AbstractFunction<R, T>` that represents a mapping from type `T` to type `R`. Here, we want to define a mapping from \mathbb{R}^{dow} , implemented by the class `WorldVector<double>`, to \mathbb{R} , represented by the data type `double`. The actual mapping is defined by overloading the `operator()`. $x*x$ stands for the scalar product of vector x with itself.

The class `F` is defined in a similar way:

```
class F : public AbstractFunction<double, WorldVector<double>>
{
public:
    F(int degree) : AbstractFunction<double, WorldVector<double>>(degree) {}

    double operator()(const WorldVector<double>& x) const
    {
        int dow = Global::getGeo(WORLD);
        double r2 = (x * x);
        double ux = exp(-10.0 * r2);
        return -(400.0 * r2 - 20.0 * dow) * ux;
    }
};
```

`F` gets the world dimension from the class `Global` by a call of the static function `getGeo(WORLD)`. The degree handed to the constructor determines the polynomial degree with which the function should be considered in the numerical integration. A higher degree leads to a quadrature of higher order in the assembling process.

Now, we start with the main program:


```
// ===== main program =====
int main(int argc, char* argv[])
{
    FUNCNAME("main");

    // ===== check for init file =====
    TEST_EXIT(argc == 2)("usage: ellipt initfile\n");

    // ===== init parameters =====
    Parameters::init(true, argv[1]);
```

The macro `FUNCNAME` defines the current function name that is used for command line output, e.g. in error messages. The macro `TEST_EXIT` tests for the condition within the first pair of brackets. If the condition does not hold, an error message given in the second bracket pair is prompted and the program exits. Here the macro is used to check, whether the parameter file was specified by the user as command line argument. If this is the case, the parameters are initialized by `Parameters::init(true, argv[1])`. The first argument specifies, whether the initialized parameters should be printed after initialization for debug reasons. The second argument is the name of the parameter file.

Now, a scalar problem with name `ellipt` is created and initialized:

```
// ===== create and init the scalar problem =====
ProblemScal ellipt("ellipt");
ellipt.initialize(INIT_ALL);
```

The name argument of the problem is used to identify parameters in the parameter file that belong to this problem. In this case, all parameters with prefix `ellipt->` are associated to this problem. The initialization argument `INIT_ALL` means that all problem modules are created in a standard way. Those are: The finite element space including the corresponding mesh, needed system matrices and vectors, an iterative solver, an estimator, a marker, and a file writer for the computed solution. The initialization of these components can be controlled through the parameter file (see Section 4.1.2).

The next steps are the creation of the adaptation loop and the corresponding `AdaptInfo`:

```
// === create adapt info ===
AdaptInfo adaptInfo("ellipt->adapt");

// === create adapt ===
AdaptStationary adapt("ellipt->adapt", ellipt, adaptInfo);
```

The `AdaptInfo` object contains information about the current state of the adaptation loop as well as user given parameters concerning the adaptation loop, like desired tolerances or maximal iteration numbers. Using `adaptInfo`, the adaptation loop can be inspected and controlled at runtime. Now, a stationary adaptation loop is created, which implements the standard *assemble-solve-estimate-adapt* loop. Arguments are the name, again used as parameter prefix, the problem as implementation of an iteration interface, and the `AdaptInfo` object. The adaptation loop only knows when to perform which part of an iteration. The implementation and execution of the single steps is delegated to an iteration interface, here implemented by the scalar problem `ellipt`.

Now, we define boundary conditions:

```
// ===== add boundary conditions =====
ellipt.addDirichletBC(1, new G);
```

We have one Dirichlet boundary condition associated with identifier 1. All nodes belonging to this boundary are set to the value of function `G` at the corresponding coordinates. In the macro file (see Section 4.1.3) the Dirichlet boundary is marked with identifier 1, too. So the nodes can be uniquely determined.

The operators now are defined as follows:

```
// ===== create matrix operator =====
Operator matrixOperator(Operator::MATRIX_OPERATOR, ellipt.getFESpace());
matrixOperator.addSecondOrderTerm(new Laplace_SOT);
ellipt.addMatrixOperator(&matrixOperator);

// ===== create rhs operator =====
int degree = ellipt.getFESpace()->getBasisFcts()->getDegree();
Operator rhsOperator(Operator::VECTOR_OPERATOR, ellipt.getFESpace());
rhsOperator.addZeroOrderTerm(new CoordsAtQP_ZOT(new F(degree)));
ellipt.addVectorOperator(&rhsOperator);
```

First, we define a matrix operator (left hand side operator) on the finite element space of the problem. Now, we add the term $-\Delta u$ to it. Note that the minus sign isn't explicitly given, but implicitly contained in `Laplace_SOT`. With `addMatrixOperator` we add the operator to the problem. The definition of the right hand side is done in a similar way. We choose the degree of our function to be equal to the current basis function degree.

Finally we start the adaptation loop and afterwards write out the results:

```
// ===== start adaption loop =====
adapt.adapt();

// ===== write result =====
ellipt.writeFiles(adaptInfo, true);
}
```

The second argument of `writeFiles` forces the file writer to print out the results. In time dependent problems it can be useful to write the results only every i -th timestep. To allow this behavior the second argument has to be `false`.

4.1.2 Parameter file

The name of the parameter file must be given as command line argument. In the 2d case we call:

```
> ./ellipt init/ellipt.dat.2d
```

In the following, the content of file `init/ellipt.dat.2d` is described:

```
dimension of world:          2

elliptMesh->macro file name:  ./macro/macro.stand.2d
elliptMesh->global refinements: 0
```

The dimension of the world is 2, the macro mesh with name `elliptMesh` is defined in file `./macro/macro.stand.2d` (see Section 4.1.3). The mesh is not globally refined before the adaptation loop. A value of n for `elliptMesh->global refinements` means n bisections of every macro element. Global refinements before the adaptation loop can be useful to save computation time by starting adaptive computations with a finer mesh.

```
ellipt->mesh:                  elliptMesh
ellipt->dim:                    2
ellipt->polynomial degree:      3
```

Now, we construct the finite element space for the problem `ellipt` (see Section 4.1.1). We use the mesh `elliptMesh`, set the problem dimension to 2, and choose Lagrange basis functions of degree 3.

```

ellipt->solver:                cg      % no bicgstab cg gmres odir ores
ellipt->solver->max iteration:  1000
ellipt->solver->tolerance:      1.e-8
ellipt->solver->left precon:    diag  % no, diag

```

We use the *conjugate gradient method* as iterative solver. The solving process stops after maximal 1000 iterations or when a tolerance of 10^{-8} is reached. Furthermore, we apply diagonal pre-conditioning.

```

ellipt->estimator:              residual % residual , recovery
ellipt->estimator->error norm:  1
ellipt->estimator->C0:          0.1
ellipt->estimator->C1:          0.1

```

As error estimator we use the residual method. The used error norm is the H1-norm (instead of the L2-norm: 2). Element residuals (C0) and jump residuals (C1) both are weighted by factor 0.1.

```

ellipt->marker->strategy:       2      % 0: no 1: GR 2: MS 3: ES 4:GERS
ellipt->marker->MSGamma:        0.5

```

After error estimation, elements are marked for refinement and coarsening. Here, we use the maximum strategy with $\gamma = 0.5$.

```

ellipt->adapt->tolerance:       1e-4
ellipt->adapt->max iteration:   100
ellipt->adapt->refine bisections: 2

```

The adaptation loop stops, when an error tolerance of 10^{-4} is reached, or after maximal 100 iterations. An element that is marked for refinement, is bisected twice within one iteration. Analog elements that are marked for coarsening are coarsened twice per iteration.

```

ellipt->output->filename:       output/ellipt
ellipt->output->AMDiS format:   1
ellipt->output->AMDiS mesh ext: .mesh
ellipt->output->AMDiS data ext: .dat

```

The result is written in AMDiS-format to the files `output/ellipt.mesh` and `output/ellipt.dat`. The first contains the final mesh, the second contains the corresponding solution values.

4.1.3 Macro file

In Figure 4.2 one can see the macro mesh which is described by the file `macro/macro.stand.2d`. First, the dimension of the mesh and of the world are defined:

```

DIM: 2
DIM_OF_WORLD: 2

```

Then the total number of elements and vertices are given:

```

number of elements: 4
number of vertices: 5

```

The next block describes the two dimensional coordinates of the five vertices:

```

vertex coordinates:
0.0 0.0
1.0 0.0
1.0 1.0
0.0 1.0
0.5 0.5

```

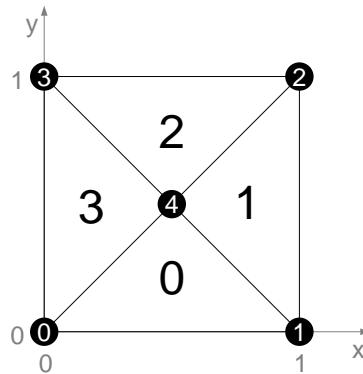


Figure 4.2: Two dimensional macro mesh

The first two numbers are interpreted as the coordinates of vertex 0, and so on.

Corresponding to these vertex indices now the four triangles are given:

```
element vertices :
0 1 4
1 2 4
2 3 4
3 0 4
```

Element 0 consists in the vertices 0, 1 and 4. The numbering is done anticlockwise starting with the vertices of the longest edge.

It follows the definition of boundary conditions:

```
element boundaries :
0 0 1
0 0 1
0 0 1
0 0 1
```

The first number line means that element 0 has no boundaries at edge 0 and 1, and a boundary with identifier 1 at edge 2. The edge with number i is the edge opposite to vertex number i . The boundary identifier 1 corresponds to the identifier 1 we defined within the source code for the Dirichlet boundary. Since all elements of the macro mesh have a Dirichlet boundary at edge 2, the line 0 0 1 is repeated three times.

The next block defines element neighborhoods. -1 means there is no neighbor at the corresponding edge. A non-negative number determines the index of the neighbor element.

```
element neighbours :
1 3 -1
2 0 -1
3 1 -1
0 2 -1
```

This block is optional. If it isn't given in the macro file, element neighborhoods are computed automatically.

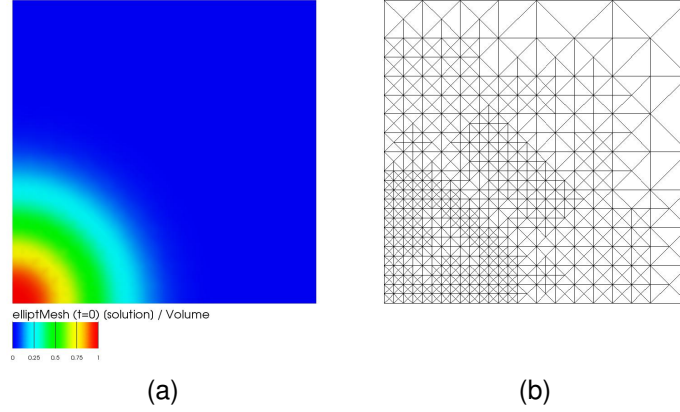


Figure 4.3: (a): Solution after 9 iterations, (b): corresponding mesh

4.1.4 Output

Now, the program is started by the call `./ellipt init/ellipt.dat.2d`. After 9 iterations the tolerance is reached and the files `output/ellipt.mesh` and `output/ellipt.dat` are written. In Figure 4.3(a) the solution is shown and in 4.3(b) the corresponding mesh. The visualizations was done by the VTK based tool **CrystalClear**.

4.2 Time dependent problem

This is an example for a time dependent scalar problem. The problem is described by the heat equation

$$\partial_t u - \Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \times (t^{begin}, t^{end}) \quad (4.5)$$

$$u = g \quad \text{on } \partial\Omega \times (t^{begin}, t^{end}) \quad (4.6)$$

$$u = u_0 \quad \text{on } \Omega \times (t^{begin}). \quad (4.7)$$

We solve the problem in the time interval (t^{begin}, t^{end}) with Dirichlet boundary conditions on $\partial\Omega$. The problem is constructed, such that the exact solution is $u(x, t) = \sin(\pi t)e^{-10x^2}$. So we set

$$f(x, t) = \pi \cos(\pi t)e^{-10x^2} - (400x^2 - 20dow) \sin(\pi t)e^{-10x^2} \quad (4.8)$$

$$g(x, t) = \sin(\pi t)e^{-10x^2} \quad (4.9)$$

$$u_0(x) = \sin(\pi t^{begin})e^{-10x^2}. \quad (4.10)$$

We use a variable time discretization scheme. Equation (4.5) is approximated by

$$\frac{u^{new} - u^{old}}{\tau} - (\theta \Delta u^{new} + (1 - \theta) \Delta u^{old}) = f(\cdot, t^{old} + \theta\tau). \quad (4.11)$$

$\tau = t^{new} - t^{old}$ is the timestep size between the old and the new problem time. u^{new} is the (searched) solution at $t = t^{new}$. u^{old} is the solution at $t = t^{old}$, which is already known from the last timestep. The parameter θ determines the implicit and explicit treatment of Δu . For $\theta = 0$ we have the forward explicit Euler scheme, for $\theta = 1$ the backward implicit Euler scheme. $\theta = 0.5$ results in the Crank-Nicholson scheme. If we bring all terms that depend on u^{old} to the right hand side, the equation reads

$$\frac{u^{new}}{\tau} - \theta \Delta u^{new} = \frac{u^{old}}{\tau} + (1 - \theta) \Delta u^{old} + f(\cdot, t^{old} + \theta\tau). \quad (4.12)$$

		1d	2d	3d
source code	src/	heat.cc		
parameter file	init/	heat.dat.1d	heat.dat.2d	heat.dat.3d
macro file	macro/	macro.stand.1d	macro.stand.2d	macro.stand.3d
output files	output/	heat.<t>.mesh, heat.<t>.dat		

Table 4.2: Files of the `heat` example. In the output file names, `<t>` is replaced by the time.

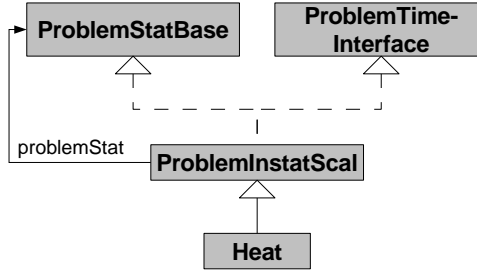


Figure 4.4: UML diagram for class `Heat`.

4.2.1 Source code

Now, we describe the crucial parts of the source code. First, the functions f and g are defined. In contrast to the elliptic example, the functions now are time dependent. This is implemented by deriving the function classes also from class `TimedObject`. This class provides a pointer to the current time, as well as corresponding setting and getting methods. The usage of a pointer to a real value allows to manage the current time in one location. All objects that deal with the same time, point to the same value. In our example, f is evaluated at $t = t^{old} + \theta\tau$, while g (the Dirichlet boundary function for u^{new}) is evaluated at $t = t^{new}$. Function g is implemented as follows:

```

class G : public AbstractFunction<double, WorldVector<double> >,
          public TimedObject
{
public:
    double operator()(const WorldVector<double>& x) const
    {
        return sin(M_PI * (*timePtr)) * exp(-10.0 * (x * x));
    }
};

```

The variable `timePtr` is a base class member of `TimedObject`. This pointer has to be set once before g is evaluated the first time. Implementation of function f is done in the same way.

Now, we begin with the implementation of class `Heat`, that represents the instationary problem. In Figure 4.4, its class diagram is shown. `Heat` is derived from class `ProblemInstatScal` which leads to following properties:

`Heat` implements the `ProblemTimeInterface`, so the adaptation loop can set the current time and schedule timesteps.

`Heat` implements `ProblemStatBase` in the role as initial (stationary) problem. The adaptation loop can compute the initial solution through this interface. The single iteration steps can be overloaded by sub classes of `ProblemInstatScal`. Actually, the initial solution is computed

through the method `solveInitialProblem` of `ProblemTimeInterface`. But this method is implemented by `ProblemInstatScal` interpreting itself as initial stationary problem.

Heat knows another implementation of `ProblemStatBase`: This other implementation represents a stationary problem which is solved within each timestep.

The first lines of class `Heat` are:

```
class Heat : public ProblemInstatScal
{
public:
    Heat(ProblemScal &heatSpace)
        : ProblemInstatScal("heat", heatSpace)
    {
        theta = -1.0;
        GET_PARAMETER(0, name + "->theta", "%f", &theta);
        TEST_EXIT(theta >= 0)("theta not set!\n");
        theta1 = theta - 1;
    }
}
```

The argument `heatSpace` is a pointer to the stationary problem which is solved each timestep. It is directly handed to the base class constructor of `ProblemInstatScal`. In the body of the constructor, θ is read from the parameter file and stored in a member variable. The member variable `theta1` stores the value of $\theta - 1$. A pointer to this value is used later as factor in the θ -scheme.

The next lines show the implementation of the time interface.

```
void setTime(AdaptInfo *adaptInfo)
{
    rhsTime = adaptInfo->getTime() - (1 - theta) * adaptInfo->getTimestep();
    boundaryTime = adaptInfo->getTime();
    tau1 = 1.0 / adaptInfo->getTimestep();
}

void closeTimestep(AdaptInfo *adaptInfo)
{
    ProblemInstatScal::closeTimestep(adaptInfo);
    WAIT;
}
```

The method `setTime` is called by the adaptation loop to inform the problem about the current time. The right hand side function f will be evaluated at $t^{old} + \theta\tau = t^{new} - (1 - \theta)\tau$, the Dirichlet boundary function g at t^{new} . t^{new} is the current time, τ is the current timestep, both set by the adaptation loop and stored in `adaptInfo`. `tau1` stores the value of $\frac{1}{\tau}$, which is used later as factor for the zero order time discretization terms.

The method `closeTimestep` is called at the end of each timestep by the adaptation loop. In the default implementation of `ProblemInstatScal::closeTimestep`, the solution is written to output files, if specified in the parameter file. Note that the base class implementation of a method must be explicitly called, if the method is overwritten in a sub class. The macro `WAIT` waits until the return key is pressed by the user, if the corresponding entry in the parameter file is set to 1. The macro `WAIT REALLY` would wait, independent of parameter settings. If `closeTimestep` wouldn't be overloaded here, the default implementation without the `WAIT` statement would be called after each timestep.

Now, the implementation of the `ProblemStatBase` interface begins. As mentioned above, the instationary problem plays the role of the initial problem by implementing this interface.

```

void solve(AdaptInfo *adaptInfo)
{
    problemStat->getSolution()->interpol(exactSolution);
}

void estimate(AdaptInfo *adaptInfo)
{
    double errMax, errSum;
    errSum = Error<double>::L2Err(*exactSolution,
                                *(problemStat->getSolution()),
                                0, &errMax, false);
    adaptInfo->setEstSum(errSum, 0);
    adaptInfo->setEstMax(errMax, 0);
}

```

Here, only the solve and the estimate step are overloaded. For the other steps, there are empty default implementations in `ProblemInstatScal`. Since the mesh is not adapted in the initial problem, the initial adaptation loop will stop after one iteration. In the solve step, the exact solution is interpolated on the macro mesh and stored in the solution vector of the stationary problem. In the estimate step, the L2 error is computed. The maximal element error and the sum over all element errors are stored in `adaptInfo`. To make the exact solution known to the problem, we need a setting function:

```

void setExactSolution(AbstractFunction<double, WorldVector<double>> *fct)
{
    exactSolution = fct;
}

```

Now, we define some getting functions and the private member variables:

```

double *getThetaPtr() { return &theta; };
double *getTheta1Ptr() { return &theta1; };
double *getTau1Ptr() { return &tau1; };
double *getRHSTimePtr() { return &rhsTime; };
double *getBoundaryTimePtr() { return &boundaryTime; };

private:
    double theta;
    double theta1;
    double tau1;
    double rhsTime;
    double boundaryTime;
    AbstractFunction<double, WorldVector<double>> *exactSolution;
};

```

The definition of class `Heat` is now finished. In the following, the main program is described.

```

int main(int argc, char** argv)
{
    // ===== check for init file =====
    TEST_EXIT(argc == 2)("usage: heat initfile\n");

    // ===== init parameters =====
    Parameters::init(false, argv[1]);

    // ===== create and init stationary problem =====

```



```

ProblemScal heatSpace("heat->space");
heatSpace.initialize(INIT_ALL);

// ===== create instationary problem =====
Heat heat(heatSpace);
heat.initialize(INIT_ALL);

```

So far, the stationary space problem `heatSpace` and the instationary problem `heat` were created and initialized. `heatSpace` is an instance of `ProblemScal`. `heat` is an instance of the class `Heat` we defined above. `heatSpace` is given to `heat` as its stationary problem.

The next step is the creation of the needed `AdaptInfo` objects and of the instationary adaptation loop:

```

// create adapt info for heat
AdaptInfo adaptInfo("heat->adapt");

// create initial adapt info
AdaptInfo adaptInfoInitial("heat->initial->adapt");

// create instationary adapt
AdaptInstationary adaptInstat("heat->adapt",
                               heatSpace,
                               adaptInfo,
                               heat,
                               adaptInfoInitial);

```

The object `heatSpace` is handed as `ProblemIterationInterface` (implemented by class `ProblemScal`) to the adaptation loop. `heat` is interpreted as `ProblemTimeInterface` (implemented by class `ProblemInstatScal`).

The definitions of functions f and g are:

```

// ===== create boundary functions =====
G *boundaryFct = new G;
boundaryFct->setTimePtr(heat.getBoundaryTimePtr());
heat.setExactSolution(boundaryFct);

heatSpace.addDirichletBC(1, boundaryFct);

// ===== create rhs functions =====
int degree = heatSpace.getFESpace()->getBasisFcts()->getDegree();
F *rhsFct = new F(degree);
rhsFct->setTimePtr(heat.getRHSTimePtr());

```

The functions interpreted as `TimedObjects` are linked with the corresponding time pointers by `setTimePtr`. The boundary function is handed to `heat` as exact solution and as Dirichlet boundary function with identifier 1 to `heatSpace`.

Now, we define the operators:

```

// ===== create operators =====
double one = 1.0;
double zero = 0.0;

// create laplace
Operator A(Operator::MATRIX_OPERATOR | Operator::VECTOR_OPERATOR,
           heatSpace.getFESpace());
A.addSecondOrderTerm(new Laplace_SOT);

```

```

A.setUhOld(heat.getOldSolution());
if (*(heat.getThetaPtr()) != 0.0)
    heatSpace.addMatrixOperator(A, heat.getThetaPtr(), &one);
if (*(heat.getTheta1Ptr()) != 0.0)
    heatSpace.addVectorOperator(A, heat.getTheta1Ptr(), &zero);

```

Operator A represents $-\Delta u$. It is used as matrix operator on the left hand side with factor θ and as vector operator on the right hand side with factor $-(1-\theta) = \theta - 1$. These assemble factors are the second arguments of `addMatrixOperator` and `addVectorOperator`. The third argument is the factor used for estimation. In this example, the estimator will consider the operator only on the left hand side with factor 1. On the right hand side the operator is applied to the solution of the last timestep. So the old solution is handed to the operator by `setUhOld`.

```

// create zero order operator
Operator C(Operator::MATRIX_OPERATOR | Operator::VECTOR_OPERATOR,
    heatSpace.getFESpace());
C.addZeroOrderTerm(new Simple_ZOT);
C.setUhOld(heat.getOldSolution());
heatSpace.addMatrixOperator(C, heat.getTau1Ptr(), heat.getTau1Ptr());
heatSpace.addVectorOperator(C, heat.getTau1Ptr(), heat.getTau1Ptr());

```

The `Simple_ZOT` of operator C represents the zero order terms for the time discretization. On both sides of the equation u is added with $\frac{1}{\tau}$ as assemble factor and as estimate factor.

Finally, the operator for the right hand side function f is added and the adaptation loop is started:

```

// create RHS operator
Operator F(Operator::VECTOR_OPERATOR, heatSpace.getFESpace());
F.addZeroOrderTerm(new CoordsAtQP_ZOT(rhsFct));
heatSpace.addVectorOperator(F);

// ===== start adaption loop =====
int errorCode = adaptInstat.adapt();
}

```

`CoordsAtQP_ZOT` is a zero order term that evaluates a given function f_{ct} at all needed quadrature points. At the left hand side, it would represent the term $f_{ct}(x, t) \cdot u$, on the right hand side, just $f_{ct}(x, t)$. Note that the old solution isn't given to the operator here. Otherwise the term would represent $f_{ct}(x, t) \cdot u^{old}$ on the right hand side.

4.2.2 Parameter file

In this section, we show only the relevant parts of the parameter file `heat.dat.2d`.

First the parameter θ for the time discretization is defined:

```
heat->theta: 1.0
```

Then we define the initial timestep and the time interval:

```

heat->adapt->timestep: 0.1
heat->adapt->start time: 0.0
heat->adapt->end time: 1.0

```

Now, tolerances are determined:

```

heat->adapt->tolerance: 0.01
heat->adapt->rel space error: 0.5
heat->adapt->rel time error: 0.5

```

```
heat->adapt->time theta 1:      1.0
heat->adapt->time theta 2:      0.3
```

The total tolerance is divided in a space tolerance and a time tolerance. The space tolerance is the maximal allowed space error, given by the product of `tolerance` and `rel space error`. It is reached by adaptive mesh refinements. The time tolerance is the maximal allowed error, due to the timestep size. It is given by the product of `tolerance` and `rel time error` and `time theta 1`. It is relevant, only if an implicit time strategy with adaptive timestep size is used. The parameter `time theta 2` is used to enlarge the timestep, if the estimated time error falls beneath a given threshold.

```
heat->adapt->strategy :          1
heat->adapt->time delta 1:       0.7071
heat->adapt->time delta 2:       1.4142
```

If `strategy` is 0, an explicit time strategy with fixed timestep size is used. A value of 1 stands for the implicit strategy. The time tolerance is reached by successively multiplying the timestep with `time delta 1`. If the estimated timestep error is smaller than the product of `tolerance` and `rel time error` and `time theta 2` at the end of a timestep, the timestep size is multiplied by `time delta 2`.

The following lines determine, whether coarsening is allowed in regions with sufficient small errors, and how many refinements or coarsenings are performed for marked elements.

```
heat->adapt->coarsen allowed:     1
heat->adapt->refine bisections:   2
heat->adapt->coarsen bisections:  2
```

Now, the output behavior is determined:

```
heat->space->output->filename :   output/heat

heat->space->output->AMDiS format: 1
heat->space->output->AMDiS mesh ext: .mesh
heat->space->output->AMDiS data ext: .dat

heat->space->output->write every i-th timestep: 10

heat->space->output->append index: 1
heat->space->output->index length: 6
heat->space->output->index decimals: 3
```

In this example, all output filenames start with prefix `output/heat` and end with the extensions `.mesh` and `.dat`. Output is written after every 10th timestep. The time of the single solution is added after the filename prefix with 6 letters, three of them are decimals. The solution for $t = 0$ e.g. would be written in the files `output/heat00.000.mesh` and `output/heat00.000.dat`.

Finally, we set parameter `WAIT` to 1. So each call of the macro `WAIT` in the application will lead to an interruption of the program, until the `return` key is pressed.

```
WAIT:                               1
```

4.2.3 Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section 4.1.3.

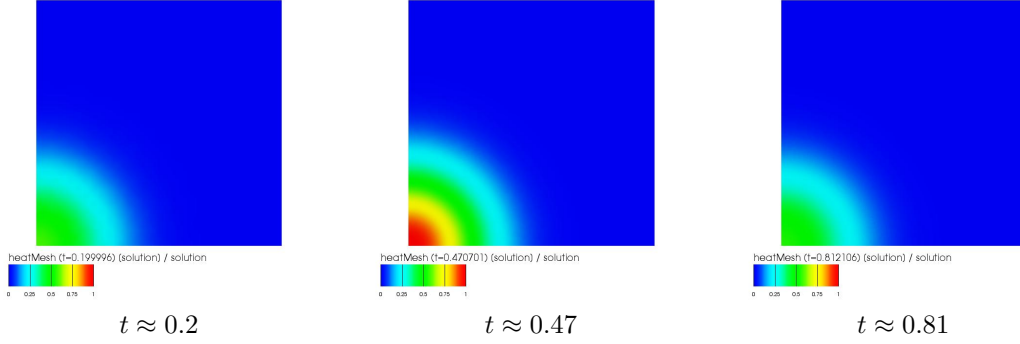


Figure 4.5: The solution at three different timesteps.

		1d	2d	3d
source code	src/	vecellipt.cc		
parameter file	init/	vecellipt.dat.1d	vecellipt.dat.2d	vecellipt.dat.3d
macro file	macro/	macro.stand.1d	macro.stand.2d	macro.stand.3d
output files	output/	vecellipt_comp<c>.mesh, vecellipt_comp<c>.dat		

Table 4.3: Files of the `vecellipt` example. In the output file names, `<c>` is replaced by the component number.

4.2.4 Output

As mentioned above, the output files look like `output/heat00.000.mesh` and `output/heat00.000.dat`. Depending on the corresponding value in the parameter file only the solution after every i -th timestep is written. In Figure 4.5, the solution at three timesteps is visualized.

4.3 Systems of PDEs

In this example, we show how to implement a system of coupled PDEs. We define

$$-\Delta u = f \quad (4.13)$$

$$u - v = 0. \quad (4.14)$$

For the first equation, we use the boundary condition and definition of function f from Section 4.1. The second equation defines a second solution component v , which is coupled to u , such that $v = u$. For the second equation, no boundary conditions have to be defined. The system can be written in matrix-vector form as

$$\begin{pmatrix} -\Delta & 0 \\ I & -I \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} f \\ 0 \end{pmatrix}, \quad (4.15)$$

where I stands for the identity and 0 for a *zero operator* (or for the absence of any operator). This is a very simple example without practical relevance. But it is appropriate to demonstrate the main principles of implementing vector valued problems.

4.3.1 Source code

Instead of a scalar problem, we now create and initialize the vector valued problem `vecellipt`:

```
ProblemVec vecellipt("vecellipt");
vecellipt.initialize(INIT_ALL);
```

The `AdaptInfo` constructor is called with the number of problem components, which is defined in the parameter file.

```
// === create adapt info ===
AdaptInfo adaptInfo("vecellipt->adapt", vecellipt.getNumComponents());
```

```
// === create adapt ===
AdaptStationary adapt("vecellipt->adapt", vecellipt, adaptInfo);
```

The adaptation loop doesn't care about the component number. It treats `vecellipt` only as implementation of the iteration interface.

The Dirichlet boundary condition for the first equation is defined by

```
// ===== add boundary conditions =====
vecellipt.addDirichletBC(1, 0, 0, new G);
```

The first argument is the condition identifier, as in the scalar case. The second and third argument define the the component, the boundary condition belongs to.

The operator definitions for the first equation are:

```
// ===== create operators =====
Operator matrixOperator00 (Operator::MATRIX_OPERATOR,
                          vecellipt.getFESpace(0),
                          vecellipt.getFESpace(0));
matrixOperator00.addSecondOrderTerm(new Laplace_SOT);
vecellipt.addMatrixOperator(&matrixOperator00, 0, 0);

int degree = vecellipt.getFESpace(0)->getBasisFcts()->getDegree();
Operator rhsOperator0 (Operator::VECTOR_OPERATOR,
                      vecellipt.getFESpace(0));
rhsOperator0.addZeroOrderTerm(new CoordsAtQP_ZOT(new F(degree)));
vecellipt.addVectorOperator(&rhsOperator0, 0);
```

Operator `matrixOperator00` represents the $-\Delta$ operator. Each operator belongs to two finite element spaces, the *row space* and the *column space*. If an operator has the position (i, j) in the operator matrix, the row space is the finite element space of component i and the column space is the finite element space of component j . The finite element spaces can differ in the used basis function degree. The underlying meshes must be the same. After `matrixOperator00` is created, it is handed to the problems operator matrix at position $(0, 0)$. The right hand side operator `rhsOperator0` only needs a row space, which is the finite element space of component 0 (u). It is handed to the operator vector at position 0 .

Now, the operators for the second equation are defined:

```
Operator matrixOperator10 (Operator::MATRIX_OPERATOR,
                          vecellipt.getFESpace(1), vecellipt.getFESpace(0));
matrixOperator10.addZeroOrderTerm(new Simple_ZOT);
vecellipt.addMatrixOperator(matrixOperator10, 1, 0);

Operator matrixOperator11 (Operator::MATRIX_OPERATOR,
                          vecellipt.getFESpace(1), vecellipt.getFESpace(1));
matrixOperator11.addZeroOrderTerm(new Simple_ZOT(-1.0));
vecellipt.addMatrixOperator(matrixOperator11, 1, 1);
```

Note that the operator `matrixOperator10` can have different finite element spaces, if the spaces of the two components differ. The operators I and $-I$ are implemented by `Simple_ZOT`, once with a fixed factor of 1 and once with a factor of -1 .

4.3.2 Parameter file

First, the number of components and the basis function degrees are given. We use Lagrange polynomials of degree 1 for the first component and of degree 2 for the second component.

```
vecellipt->components:          2

vecellipt->polynomial degree[0]: 1
vecellipt->polynomial degree[1]: 2
```

In general, the linear system of equations for systems of PDEs is not symmetric. So with the GMRes solver, we use a solver that doesn't assume symmetric matrices.

```
vecellipt->solver:               gmres
```

Note that we have only one solver, because the equations of our system are assembled in one linear system of equations.

Each equation can have its own estimator. In this case, adaptivity should be managed only by the first component. So the second equation has no estimator.

```
vecellipt->estimator[0]:        residual
vecellipt->estimator[1]:        no
```

Also the marking strategy can differ between the components. Refinement is done, if at least one component has marked an element for refinement. Coarsening only is done, if all components have marked the element for coarsening. In our example, only component 0 will mark elements.

```
vecellipt->marker[0]->strategy: 2
vecellipt->marker[1]->strategy: 0
```

We have only one adaptation loop, which does maximal 6 iterations. The tolerance can be determined for each component. The total tolerance criterion is fulfilled, if all criteria of all components are fulfilled.

```
vecellipt->adapt->max iteration: 6

vecellipt->adapt[0]->tolerance:  1e-4
vecellipt->adapt[1]->tolerance:  1e-4
```

Also the output can be controlled for each component individually:

```
vecellipt->output[0]->filename:  output/vecellipt_comp0

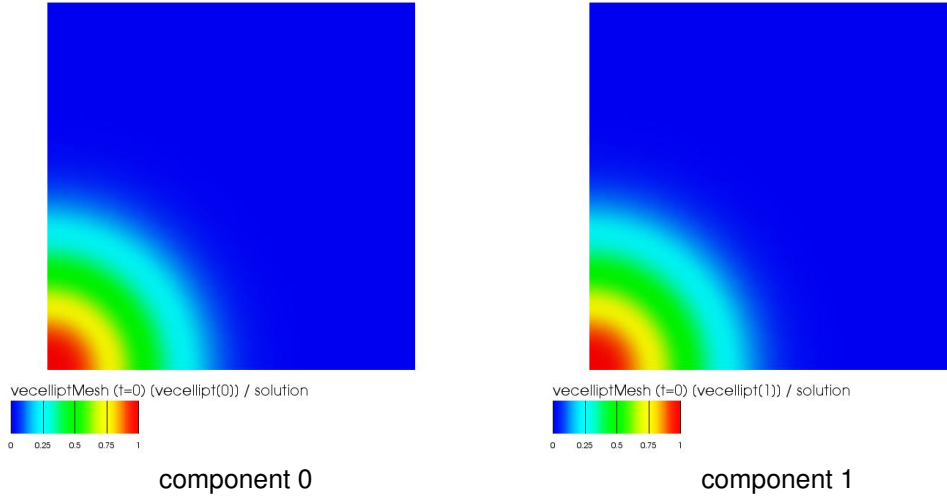
vecellipt->output[0]->AMDIS format:  1
vecellipt->output[0]->AMDIS mesh ext: .mesh
vecellipt->output[0]->AMDIS data ext: .dat

vecellipt->output[1]->filename:  output/vecellipt_comp1

vecellipt->output[1]->AMDIS format:  1
vecellipt->output[1]->AMDIS mesh ext: .mesh
vecellipt->output[1]->AMDIS data ext: .dat
```

4.3.3 Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section 4.1.3.

Figure 4.6: The two solution components for u and v .

		1d	2d	3d
source code	src/	couple.cc		
parameter file	init/	couple.dat.1d	couple.dat.2d	couple.dat.3d
macro file	macro/	macro.stand.1d	macro.stand.2d	macro.stand.3d
output files	output/	couple.mesh, couple.dat		

Table 4.4: Files of the couple example.

4.3.4 Output

Component 0 of the solution (approximation of u) is written to the files `output/vecellipt0.mesh` and `output/vecellipt0.dat`. Component 1 of the solution (approximation of v) is written to the files `output/vecellipt1.mesh` and `output/vecellipt1.dat`. The two components are visualized in Figure 4.6.

4.4 Coupled problems

In this example, we solve the same problem as in Section 4.3, but here we treat the two equations as two coupled problems. The main difference is that the equations now aren't assembled into the same large system of equations, but into two separated systems of equations, that have to be solved separately. We define the two problems

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \quad (4.16)$$

$$u = g \quad \text{on } \partial\Omega \quad (4.17)$$

and

$$v = u. \quad (4.18)$$

We first solve the first problem and then use its solution to solve the second problem. This happens in every iteration of the adaptation loop. Both problems should use the same mesh. Mesh adaptation is done by the first problem. So one iteration now looks like illustrated in Figure 4.7.

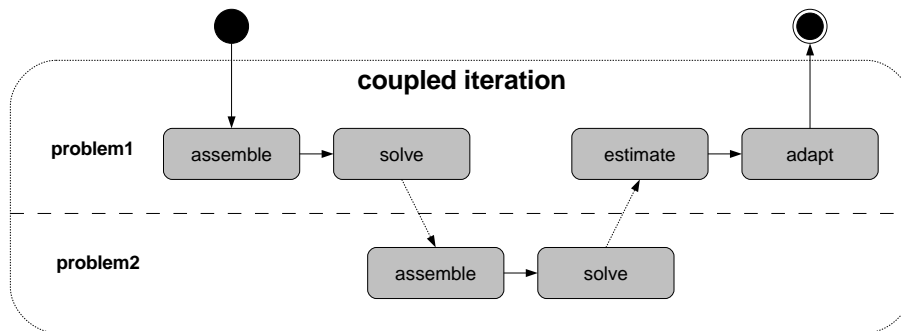


Figure 4.7: State diagram of the coupled iteration.

4.4.1 Source code

In the previous examples, the iteration was implemented always by the corresponding problem class. In this example, we want to couple two problems within one iteration, so the default implementation can't be used. For that reason, we define our own coupled iteration class `MyCoupledIteration` which implements the `ProblemIterationInterface`:

```
class MyCoupledIteration : public ProblemIterationInterface
{
public:
    MyCoupledIteration(ProblemStatBase *prob1,
                       ProblemStatBase *prob2)
        : problem1(prob1),
          problem2(prob2)
    {}
}
```

In the constructor pointers to the two problems are assigned to the private members `problem1` and `problem2`. Note that the pointers point to the interface `ProblemStatBase` and not to `ProblemScal`. This leads to a more general implementation. If e.g. two vector valued problems should be coupled in the future, we could use our iteration class without modifications.

Now, we implement the needed interface methods:

```
void beginIteration(AdaptInfo *adaptInfo)
{
    FUNCNAME(" StandardProblemIteration::beginIteration()");
    MSG("\n");
    MSG("begin of iteration %d\n", adaptInfo->getSpaceIteration()+1);
    MSG("=====\n");
}

void endIteration(AdaptInfo *adaptInfo) {
    FUNCNAME(" StandardProblemIteration::endIteration()");
    MSG("\n");
    MSG("end of iteration %d\n", adaptInfo->getSpaceIteration()+1);
    MSG("=====\n");
}
```

These two functions are called at the beginning and at the end of each iteration. Here, we only prompt some output.

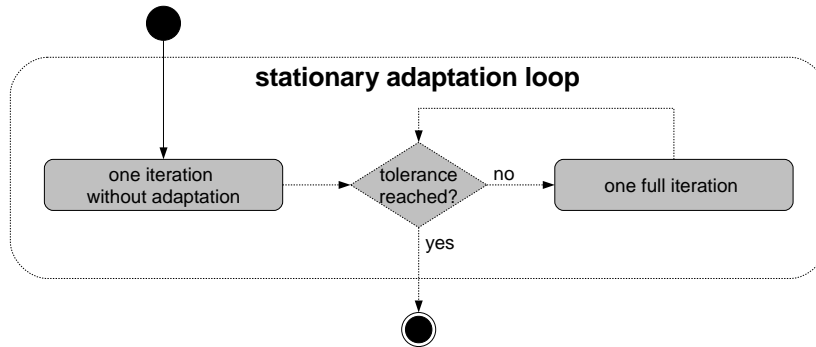


Figure 4.8: Stationary adaptation loop.

The method `oneIteration` is the crucial part of our implementation:

```

Flag oneIteration(AdaptInfo *adaptInfo, Flag toDo = FULL_ITERATION)
{
    Flag flag, markFlag;
    if (toDo.isSet(MARK)) markFlag = problem1->markElements(adaptInfo);
    if (toDo.isSet(ADAPT) && markFlag.isSet(MESH_REFINED))
        flag = problem1->refineMesh(adaptInfo);

    if (toDo.isSet(BUILD)) problem1->buildAfterCoarsen(adaptInfo, markFlag);
    if (toDo.isSet(SOLVE)) problem1->solve(adaptInfo);

    if (toDo.isSet(BUILD)) problem2->buildAfterCoarsen(adaptInfo, markFlag);
    if (toDo.isSet(SOLVE)) problem2->solve(adaptInfo);

    if (toDo.isSet(ESTIMATE)) problem1->estimate(adaptInfo);
    return flag;
}

```

The `toDo` flag is used by the adaptation loop to determine which parts of the iteration should be performed. The first iteration is always an iteration without mesh adaptation (see Figure 4.8). So we start our iteration by marking and adapting the mesh. The mesh and its adaptation is managed by the first problem. So we call `markElements` and `refineMesh` of `problem1`. Note that no mesh coarsenings have to be performed in our example. Afterwards, `problem1` assembles its system of equations by `buildAfterCoarsen`. Assembly and mesh adaptation are nested operations in AMDiS (`buildBeforeRefine`, `refineMesh`, `buildBeforeCoarsen`, `coarsenMesh`, `buildAfterCoarsen`). Here, we implement a simplified version.

After `problem1` has solved its system of equations, `problem2` can assemble and solve its equations system using the solution of the first problem as right hand side. In the method `oneIteration`, only the order of method calls is determined. The dependency to the solution of the first problem is created later when the operator for the right hand side of `problem2` is created.

After also the second problem computed its solution, `problem1` does the error estimation (remember: mesh adaptation is managed by `problem1`).

Now, the access to the coupled problems is implemented and the member variables are defined:

```
int getNumProblems()
```

```

    {
        return 2;
    }

    ProblemStatBase *getProblem(int number = 0)
    {
        FUNCNAME("CoupledIteration::getProblem()");
        if (number == 0) return problem1;
        if (number == 1) return problem2;
        ERROR_EXIT("invalid problem number\n");
        return NULL;
    }

private:
    ProblemStatBase *problem1;
    ProblemStatBase *problem2;
};

```

The class `MyCoupledIteration` is finished now.

The next class, `Identity`, implements the identity $I(x) = x$ for double variables. An arbitrary degree can be given to it. The class is used later to determine the quadrature degree used for the right hand side of `problem2`.

```

class Identity : public AbstractFunction<double, double>
{
public:
    Identity(int degree) : AbstractFunction<double, double>(degree) {}

    double operator()(const double& x) const
    {
        return x;
    }
};

```

Now, we start with the main program:

```

int main(int argc, char* argv[])
{
    FUNCNAME("main");
    TEST_EXIT(argc == 2)("usage: couple initfile\n");
    Parameters::init(true, argv[1]);

    // ===== create and init the first problem =====
    ProblemScal problem1("problem1");
    problem1.initialize(INIT_ALL);

    // ===== add boundary conditions for problem1 =====
    problem1.addDirichletBC(1, new G);

```

So far, we created and initialized `problem1` and its boundary conditions.

Now, we create `problem2`. It should have its own finite element space, system, solver and file writer, but the mesh should be adopted from `problem1`.

```

    // ===== create and init the second problem =====
    Flag initFlag =
        INIT_FE_SPACE |

```

```

INIT_SYSTEM |
INIT_SOLVER |
INIT_FILEWRITER;

Flag adoptFlag =
  CREATE_MESH |
  INIT_MESH;

ProblemScal problem2("problem2");
problem2.initialize(initFlag,
                  &problem1,
                  adoptFlag);

```

The operators for the first problem are defined like in Section 4.1.1. Here, we only show the operators of `problem2`.

```

// ===== create operators for problem2 =====
Operator matrixOperator2(Operator::MATRIX_OPERATOR,
                        problem2.getFESpace());
matrixOperator2.addZeroOrderTerm(new Simple_ZOT);
problem2.addMatrixOperator(&matrixOperator2);

Operator rhsOperator2(Operator::VECTOR_OPERATOR, problem2.getFESpace());
rhsOperator2.addZeroOrderTerm(new VecAtQP_ZOT(problem1.getSolution(),
                                              new Identity(degree)));
problem2.addVectorOperator(&rhsOperator2);

```

At the left hand side, we have just an ordinary `Simple_ZOT`. At the right hand side, we have a zero order term of the form $f(u)$ with $f = I$ the identity. u is given by the solution DOF vector of `problem1`. I maps the values of the DOF vector evaluated at quadrature points to itself. The function degree is used to determine the needed quadrature degree in the assembler.

Now, the adaptation loop is created:

```

// ===== create adaptation loop and iteration interface =====
AdaptInfo *adaptInfo = new AdaptInfo("couple->adapt", 1);

MyCoupledIteration coupledIteration(&problem1, &problem2);

AdaptStationary *adapt = new AdaptStationary("couple->adapt",
                                              &coupledIteration,
                                              adaptInfo);

```

Note that not a pointer to one of the problems is passed to the adaptation loop, but a pointer to the `coupledIteration` object, which in turn knows both problems.

The adaptation loop is now started. After it is finished, the solutions of both problems are written.

```

// ===== start adaptation loop =====
adapt->adapt();

// ===== write solution =====
problem1.writeFiles(adaptInfo, true);
problem2.writeFiles(adaptInfo, true);
}

```

4.4.2 Parameter file

We have one adaptation loop called `couple→adapt`:

```
couple→adapt→tolerance :      1e-8
couple→adapt→max iteration :    10
couple→adapt→refine bisections : 2
```

The coupled problem consists of two sub problems. The first problem creates the mesh, solves its linear system of equations, estimates the error, adapts the mesh, and finally writes its output:

```
coupleMesh→macro file name :    ./macro/macro.stand.2d
coupleMesh→global refinements : 0

problem1→mesh :                  coupleMesh
problem1→dim :                   2
problem1→polynomial degree :     1

problem1→solver :                cg % no, bicgstab , cg, gmres, odir , ores
problem1→solver→max iteration : 1000
problem1→solver→tolerance :      1.e-8
problem1→solver→left precon :    diag

problem1→estimator :             residual
problem1→estimator→C0 :          0.1 % constant of element residual
problem1→estimator→C1 :          0.1 % constant of jump residual

problem1→marker→strategy :       2 % 0: no 1: GR 2: MS 3: ES 4:GERS
problem1→marker→MSGamma :        0.5

problem1→output→filename :       output/problem1
problem1→output→AMDiS format :   1
problem1→output→AMDiS mesh ext : .mesh
problem1→output→AMDiS data ext : .dat
```

The second problem uses the mesh of `problem1`. So it creates no mesh, no estimator, and no marker. But a solver is needed to solve `problem2`'s linear system of equations, and a file writer to write the solution:

```
problem2→dim :                   2
problem2→polynomial degree :     1

problem2→solver :                cg % no, bicgstab , cg, gmres, odir , ores
problem2→solver→max iteration : 1000
problem2→solver→tolerance :      1.e-8
problem2→solver→left precon :    diag

problem2→estimator :             no

problem2→marker→strategy :       0

problem2→output→filename :       output/problem2
problem2→output→AMDiS format :   1
problem2→output→AMDiS mesh ext : .mesh
problem2→output→AMDiS data ext : .dat
```

4.4.3 Macro file

We again use the macro file `macro/macro.stand.2d`, which was described in Section 4.1.3.

4.4.4 Output

The solution of the first problem is written to the files `output/problem1.mesh` and `output/problem1.dat`. The solution of the second problem is written to the files `output/problem2.mesh` and `output/problem2.dat`. We don't visualize the results here, because they conform with the results showed in Section 4.3.4.

4.5 Nonlinear problem

We define the nonlinear problem

$$-\Delta u + u^4 = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \quad (4.19)$$

$$u = g \quad \text{on } \partial\Omega. \quad (4.20)$$

We choose the functions f and g so that the exact solution again is $u(x) = e^{-10x^2}$. This leads to

$$f(x) = -(400 - 20dow) e^{-10x^2} + (e^{-10x^2})^4 \quad (4.21)$$

$$g(x) = e^{-10x^2}, \quad (4.22)$$

with dow the world dimension.

We linearize the problem using the Newton method. First, we define an initial guess u_0 of the solution which is 0 for the first adaptation loop iteration. In later iterations we can use the solution of the last iteration interpolated to the current mesh as initial guess. In each Newton step, a correction d for the solution of the last step is computed by solving

$$DF(u_n)(d) = F(u_n) \quad (4.23)$$

for d , where $F(u) := -\Delta u + u^4 - f$ and

$$DF(u_n)(d) = \lim_{h \rightarrow 0} \frac{F(u_n + hd) - F(u_n)}{h} \quad (4.24)$$

$$= \lim_{h \rightarrow 0} \frac{-\Delta u_n - h\Delta d + \Delta u_n}{h} + \lim_{h \rightarrow 0} \frac{(u_n + hd)^4 - u_n^4}{h} \quad (4.25)$$

$$= -\Delta d + 4u_n^3 d \quad (4.26)$$

the directional derivative of F at u_n along d .

Then the solution is updated:

$$u_{n+1} := u_n - d. \quad (4.27)$$

We repeat this procedure until $\|d\|_{L^2} < tol$ with tol a given tolerance for the Newton method.

In our example, equation (4.23) reads:

$$-\Delta d + 4u_n^3 d = -\Delta u_n + u_n^4 - f. \quad (4.28)$$

In Figure 4.9, the Newton method is illustrated.

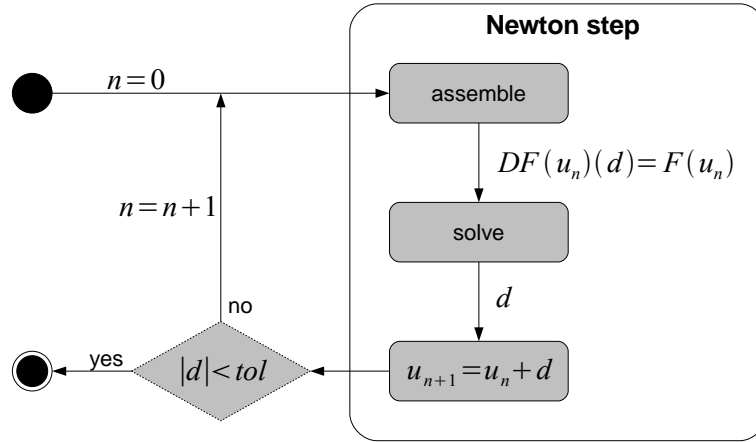


Figure 4.9: Solve step of the nonlinear problem.

		1d	2d	3d
source code	src/	nonlin.cc		
parameter file	init/	nonlin.dat.1d	nonlin.dat.2d	nonlin.dat.3d
macro file	macro/	macro.big.stand.1d	macro.big.stand.2d	macro.big.stand.3d
output files	output/	nonlin.mesh, nonlin.dat		

Table 4.5: Files of the `nonlin` example.

4.5.1 Source code

The main idea is to realize the Newton method as implementation of the *ProblemIterationInterface*, which replaces the standard iteration in the adaptation loop. The Newton method has to know the problem which has to be solved, as well as the implementation of one Newton step. Estimation and adaptation is done by the problem object. The assemble step and solve step are delegated to a Newton-step object.

Now, we describe the code step by step. The function g is defined like in the previous examples. In the following, we define a zero-function which is later used to implement the Dirichlet boundary condition for the Newton-step implementation (at domain boundaries no correction has to be done). The function F implements the right hand side function f .

```

class Zero : public AbstractFunction<double, WorldVector<double>>
{
public:
    double operator()(const WorldVector<double>& x) const
    {
        return 0.0;
    }
};

class F : public AbstractFunction<double, WorldVector<double>>
{
public:
    /// Constructor
    F(int degree)
        : AbstractFunction<double, WorldVector<double>>(degree)

```

```

    {}

    /// Implementation of AbstractFunction::operator().
    double operator()(const WorldVector<double>& x) const
    {
        int dow = x.getSize();
        double r2 = x * x;
        double ux = exp(-10.0 * r2);
        double ux4 = ux * ux * ux * ux;
        return ux4 - (400.0 * r2 - 20.0 * dow) * ux;
    }
};

```

The class X3 implements the function $u^3(x)$ used within the Newton step.

```

class X3 : public AbstractFunction<double, double>
{
public:
    X3() : AbstractFunction<double, double>(3) {}

    /// Implementation of AbstractFunction::operator().
    double operator()(const double& x) const
    {
        return x * x * x;
    }
};

```

In the following, we define an interface which has to be implemented by the Newton-step object.

```

class NewtonStepInterface
{
public:
    virtual void initNewtonStep(AdaptInfo *adaptInfo) = 0;
    virtual void exitNewtonStep(AdaptInfo *adaptInfo) = 0;
    virtual void assembleNewtonStep(AdaptInfo *adaptInfo, Flag flag) = 0;
    virtual void solveNewtonStep(AdaptInfo *adaptInfo) = 0;
    virtual DOFVector<double> *getCorrection() = 0;
};

```

The `initNewtonStep` method is called before each Newton step, the method `exitNewtonStep` after each Newton step. `assembleNewtonStep` assembles the linear system of equations needed for the next step, `solveNewtonStep` solves this system of equations. The solution is the correction d . The method `getCorrection` returns a pointer to the vector storing the correction.

Now, the Newton method will be implemented. Actually, the class `NewtonMethod` replaces the whole iteration in the adaptation loop, including mesh adaptation and error estimation. The Newton method, which is a loop over Newton steps, is one part of this iteration.

```

class NewtonMethod : public ProblemIterationInterface
{
public:
    NewtonMethod(const char *name,
                 ProblemScal *problem,
                 NewtonStepInterface *step)
        : problemNonlin(problem),
          newtonStep(step),

```

```

        newtonTolerance(1e-8),
        newtonMaxIter(100)
    {
        GET_PARAMETER(0, std::string(name) + "->tolerance", "%f",
                      &newtonTolerance);
        GET_PARAMETER(0, std::string(name) + "->max iteration", "%d",
                      &newtonMaxIter);
        solution = problemNonlin->getSolution();
        correction = newtonStep->getCorrection();
    }

```

In the constructor, pointers to the nonlinear problem and to the Newton-step object are stored to the class members `problemNonlin` and `newtonStep`. Furthermore, the parameters `newtonTolerance` and `newtonMaxIter` are initialized, and pointers to the nonlinear solution and to the correction vector are stored.

The following methods define one iteration in the adaptation loop.

```

void beginIteration(AdaptInfo *adaptInfo)
{
    FUNCNAME("NewtonMethod::beginIteration()");
    MSG("\n");
    MSG("begin of iteration %d\n", adaptInfo->getSpacelIteration()+1);
    MSG("=====\n");
}

Flag onelIteration(AdaptInfo *adaptInfo, Flag toDo = FULL_ITERATION)
{
    Flag flag = 0, markFlag = 0;

    if (toDo.isSet(MARK)) markFlag = problemNonlin->markElements(adaptInfo);
    if (toDo.isSet(ADAPT) && markFlag.isSet(MESH_REFINED))
        flag = problemNonlin->refineMesh(adaptInfo);
    if (toDo.isSet(ADAPT) && markFlag.isSet(MESH_COARSENEDED))
        flag |= problemNonlin->coarsenMesh(adaptInfo);

    if (toDo.isSet(SOLVE)) {
        newtonStep->initNewtonStep(adaptInfo);
        int newtonIteration = 0;
        double res = 0.0;
        do {
            newtonIteration++;
            newtonStep->assembleNewtonStep(adaptInfo, flag);
            newtonStep->solveNewtonStep(adaptInfo);
            res = correction->L2Norm();
            *solution -= *correction;
            MSG("newton iteration %d: residual %f (tol: %f)\n",
                newtonIteration, res, newtonTolerance);
        } while ((res > newtonTolerance) && (newtonIteration < newtonMaxIter));

        newtonStep->exitNewtonStep(adaptInfo);
    }

    if (toDo.isSet(ESTIMATE)) problemNonlin->estimate(adaptInfo);
    return flag;
}

```



```

}

void endIteration(AdaptInfo *adaptInfo)
{
    FUNCNAME("NewtonMethod::endIteration()");
    MSG("\n");
    MSG("end of iteration number: %d\n", adaptInfo->getSpaceIteration()+1);
    MSG("=====\n");
}

```

The methods `beginIteration` and `endIteration` only print some information to the standard output. In `oneIteration`, the iteration, including the loop over the Newton steps, is defined.

Finally, the methods `getNumProblems` and `getProblem` are implemented to complete the `ProblemIterationInterface`, and the private class members are defined.

```

int getNumProblems() { return 1; };

ProblemStatBase *getProblem(int number = 0)
{
    FUNCNAME("NewtonMethod::getProblem()");
    if (number == 0) return problemNonlin;
    ERROR_EXIT("invalid problem number\n");
    return NULL;
}

private:
    ProblemScal *problemNonlin;
    NewtonStepInterface *newtonStep;
    double newtonTolerance;
    int newtonMaxIter;
    DOFVector<double> *solution;
    DOFVector<double> *correction;
};

```

The class `Nonlin` implements both, the nonlinear problem and the Newton-step. Since the Newton step is accessed over an own interface, it is always clear, in which role a `Nonlin` instance is called by the Newton method.

```

class Nonlin : public ProblemScal,
               public NewtonStepInterface
{
public:
    Nonlin(const char *name)
        : ProblemScal(name)
    {}
}

```

In the constructor, the base class constructor of `ProblemScal` is called and the name is given to it.

In the initialization, the base class initialization is called, the correction vector is created and initialized, and Dirichlet boundary conditions are created.

```

void initialize(Flag initFlag,
               ProblemScal *adoptProblem = NULL,
               Flag adoptFlag = INIT_NOTHING)
{
    ProblemScal::initialize(initFlag, adoptProblem, adoptFlag);
}

```

```

correction = new DOFVector<double>(this->getFESpace(), "old solution");
correction->set(0.0);

dirichletZero = new DirichletBC(1, &zero, feSpace_);
dirichletG = new DirichletBC(1, &g, feSpace_);

solution->getBoundaryManager()->addBoundaryCondition(dirichletG);
systemMatrix->getBoundaryManager()->
    addBoundaryCondition(dirichletZero);
rhs->getBoundaryManager()->addBoundaryCondition(dirichletZero);
correction->getBoundaryManager()->addBoundaryCondition(dirichletZero);
};

```

To the solution of the nonlinear problem the function g is applied as Dirichlet boundary function. The system matrix, the correction vector and the right hand side vector build the system for the Newton step. Since no correction has to be done at the domain boundaries, zero Dirichlet conditions are applied to them.

In the destructor, the allocated memory is freed.

```

~Nonlin()
{
    delete correction;
    delete dirichletZero;
    delete dirichletG;
}

```

Now, we implement the Newton step functionality. First, in `initNewtonStep`, we fill the solution vector with boundary values. This will not be done automatically because we let the `solution_` pointer point to `correction`. The address of `solution_` is stored in `tmp`. After the Newton method is finished, the `solution_` pointer is reset to its original value in `exitNewtonStep`.

```

void initNewtonStep(AdaptInfo *adaptInfo)
{
    solution->getBoundaryManager()->initVector(solution);
    TraverseStack stack;
    EllInfo *ellInfo = stack.traverseFirst(mesh_, -1,
                                           Mesh::CALL_LEAF_EL |
                                           Mesh::FILL_COORDS |
                                           Mesh::FILL_BOUND);

    while (ellInfo) {
        solution->getBoundaryManager()->fillBoundaryConditions(ellInfo, solution);
        ellInfo = stack.traverseNext(ellInfo);
    }
    solution->getBoundaryManager()->exitVector(solution);

    tmp = solution;
    solution = correction;
}

void exitNewtonStep(AdaptInfo *adaptInfo)
{
    solution = tmp;
}

```

The implementation of `assembleNewtonStep` and `solveNewtonStep` just delegates the calls to the base class implementations in `ProblemSca1`.

```
void assembleNewtonStep(AdaptInfo *adaptInfo, Flag flag)
{
    ProblemScal::buildAfterCoarsen(adaptInfo, flag);
}
```

```
void solveNewtonStep(AdaptInfo *adaptInfo)
{
    ProblemScal::solve(adaptInfo);
}
```

Finally, the `getCorrection` method is implemented and private class members are defined.

```
DOFVector<double> *getCorrection()
{
    return correction;
}
```

```
private:
    DOFVector<double> *correction;
    DOFVector<double> *tmp;
    Zero zero;
    G g;
    DirichletBC *dirichletZero;
    DirichletBC *dirichletG;
};
```

Now, we start with the main program.

```
int main(int argc, char* argv[])
{
    FUNCNAME("main");

    TEST_EXIT(argc == 2)("usage: nonlin initfile\n");

    Parameters::init(false, argv[1]);

    Nonlin nonlin("nonlin");
    nonlin.initialize(INIT_ALL);

    AdaptInfo *adaptInfo = new AdaptInfo("nonlin->adapt", 1);

    NewtonMethod newtonMethod("nonlin->newton", &nonlin, &nonlin);

    AdaptStationary *adapt = new AdaptStationary("nonlin->adapt",
                                                    &newtonMethod,
                                                    adaptInfo);
```

An instance of class `NewtonMethod` was created with a pointer to a `Nonlin` object as nonlinear problem and as Newton-step implementation. Instead of the nonlinear problem, now, the object `newtonMethod` is handed to the adaptation loop as implementation of `ProblemIterationInterface`.

We have to add operators representing the Newton step equation $-\Delta d + 4u_n^3 d = -\Delta u_n + u_n^4 - f$ as well as operators representing the nonlinear problem $-\Delta u_n + u_n^4 = f$. When the operators are given to the problem, one can determine an assemble factor (second argument of `addMatrixOperator` and `addVectorOperator`) as well as an estimation factor (third argument) for each operator. So, we can manage both equations in one problem instance. Note that in the Newton step we solve for d . u_n there is known from the last Newton step. The term u_n^4 was

	$-\Delta v$		$u_n^3 v$		f	
	mat	vec	mat	vec	mat	vec
assemble($v=d$)	1	1	4	1	0	-1
estimate($v=u_n$)	1	0	1	0	0	1

Figure 4.10: Operator factors for the assemble step and for the estimate step.

implemented as $u_n^3 v$, where v for the Newton step equation is equal to d and in the nonlinear problem equation equal to u_n . So, the corresponding operator can be used in both equations just with different factors. In Figure 4.10, the operator factors for the assemble and the estimate step are shown.

```
// ===== create operators =====
double four = 4.0;
double one = 1.0;
double zero = 0.0;
double minusOne = -1.0;

Operator *nonlinOperator0 = new Operator(Operator::MATRIX_OPERATOR |
                                           Operator::VECTOR_OPERATOR,
                                           nonlin.getFESpace());

nonlinOperator0->setUhOld(nonlin.getSolution());
nonlinOperator0->addZeroOrderTerm(new VecAtQP_ZOT(nonlin.getSolution(),
                                                    new X3));

nonlin.addMatrixOperator(nonlinOperator0, &four, &one);
nonlin.addVectorOperator(nonlinOperator0, &one, &zero);

Operator *nonlinOperator2 = new Operator(Operator::MATRIX_OPERATOR |
                                           Operator::VECTOR_OPERATOR,
                                           nonlin.getFESpace());

nonlinOperator2->setUhOld(nonlin.getSolution());
nonlinOperator2->addSecondOrderTerm(new Laplace_SOT);

nonlin.addMatrixOperator(nonlinOperator2, &one, &one);
nonlin.addVectorOperator(nonlinOperator2, &one, &zero);

int degree = nonlin.getFESpace()->getBasisFcts()->getDegree();

Operator* rhsFunctionOperator = new Operator(Operator::VECTOR_OPERATOR,
                                              nonlin.getFESpace());
rhsFunctionOperator->addZeroOrderTerm(new CoordsAtQP_ZOT(new F(degree)));

nonlin.addVectorOperator(rhsFunctionOperator, &minusOne, &one);

Finally, the adaptation loop is started and after it is finished the result is written.

adapt->adapt();
nonlin.writeFiles(adaptInfo, true);
```

```
}
```

4.5.2 Parameter file

The used parameter file `nonlin.dat.2d` looks like:

```
dimension of world: 2
```

```
nonlinMesh->macro file name:      ./macro/macro_big_stand.2d
nonlinMesh->global refinements:    0
```

```
nonlin->mesh: nonlinMesh
```

```
nonlin->dim: 2
nonlin->polynomial degree: 1
```

```
nonlin->newton->tolerance: 1e-8
nonlin->newton->max iteration: 100
```

```
nonlin->solver: cg
nonlin->solver->max iteration: 1000
nonlin->solver->tolerance: 1.e-8
nonlin->solver->left precon: diag
```

```
nonlin->estimator: residual
nonlin->estimator->C0: 0.1
nonlin->estimator->C1: 0.1
```

```
nonlin->marker->strategy: 2
nonlin->marker->MSCGamma: 0.5
```

```
nonlin->adapt->tolerance: 1e-1
nonlin->adapt->max iteration: 100
```

```
nonlin->output->filename: output/nonlin
nonlin->output->AMDiS format: 1
nonlin->output->AMDiS mesh ext: .mesh
nonlin->output->AMDiS data ext: .dat
```

Here, as macro file `macro_big_stand.2d` is used, which is described in Section 4.5.3. The parameters `nonlin->newton->tolerance` and `nonlin->newton->max iteration` determine the tolerance of the Newton solver and the maximal number of Newton steps within each iteration of the adaptation loop.

4.5.3 Macro file

The used macro file `macro_big_stand.2d` is very similar to the macro file of the first example described in Section 4.1.3. Only the domain was changed from $\Omega = [0, 1]^2$ to $\Omega = [-1, 1]^2$ by adapting the vertex coordinates correspondingly.

4.5.4 Output

In Figure 4.11, the solution and the final mesh written after the adaptation loop are visualized. The solution is shown as height field where the values are interpreted as z-coordinates.

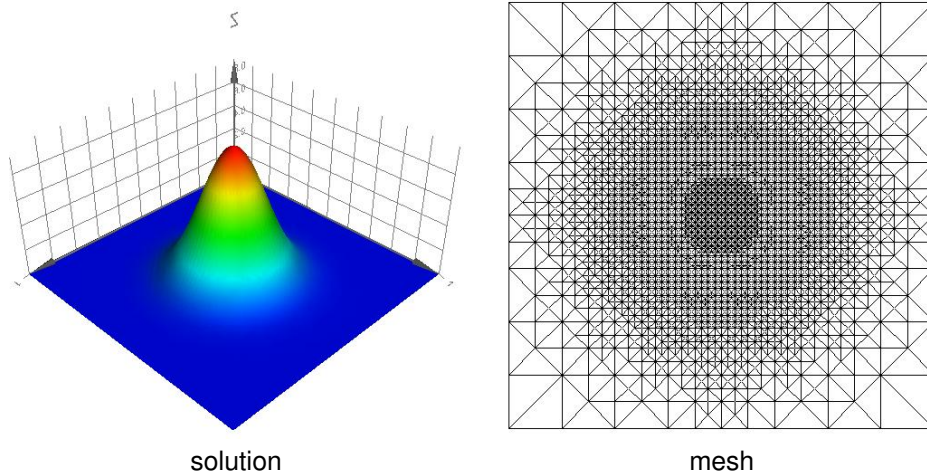


Figure 4.11: Solution and final mesh of the nonlinear problem

		1d	2d	3d
source code	src/	neumann.cc		
parameter file	init/	neumann.dat.1d	neumann.dat.2d	neumann.dat.3d
macro file	macro/	neumann.macro.1d	neumann.macro.2d	neumann.macro.3d
output files	output/	neumann.mesh, neumann.dat		

Table 4.6: Files of the *neumann* example.

4.6 Neumann boundary conditions

In this example, we solve the problem defined in Section 4.1. But now, we set the domain Ω to $[-0.5; 0.5]^2$, so the source f is located in the middle of Ω . Furthermore, we use Neumann boundary conditions on the left and on the right side of Ω . We set $A\nabla u \cdot \nu = 1$ at the Neumann boundary. So, the derivative in direction of the surface normal is set to 1 at these points. The rest of the boundary keeps unchanged (Dirichlet boundary, set to the true solution).

4.6.1 Source code

Only a few changes in the source code are necessary to apply Neumann boundary conditions. First, we define the function $N = 1$.

```
class N : public AbstractFunction<double, WorldVector<double>>
{
public:
    double operator()(const WorldVector<double>& x) const
    {
        return 1.0;
    }
};
```

In the main program we add the boundary conditions to our problem *neumann*.

```
int main(int argc, char* argv[])
{
    ...
```

```

    neumann.addNeumannBC(1, new N);
    neumann.addDirichletBC(2, new G);
    ...
}

```

Since the Dirichlet condition has a higher ID, it has a higher priority against the Neumann boundary condition. This is important, where different conditions meet each other in some points. In this example, these are the corner points of Ω . If Dirichlet boundary conditions are used together with boundary conditions of other type, the Dirichlet conditions should always have the higher priority.

4.6.2 Parameter file

In the parameter file, we use the file `./macro/neumann.macro.2d` as macro mesh file, described in the next section.

4.6.3 Macro file

The file `neumann.macro.2d` is listed below:

```

DIM: 2
DIM_OF_WORLD: 2

number of vertices: 5
number of elements: 4

vertex coordinates:
-0.5 -0.5
 0.5 -0.5
 0.5  0.5
-0.5  0.5
 0.0  0.0

element vertices:
0 1 4
1 2 4
2 3 4
3 0 4

element boundaries:
0 0 2
0 0 1
0 0 2
0 0 1

```

In contrast to the standard file `macro.stand.2d`, here the vertex coordinates are shifted to describe the domain $[-0.5; 0.5]^2$. Furthermore, the boundary block changed. The elements 0 and 2 have the Dirichlet boundary with ID 2 at edge 2. Elements 1 and 3 have the Neumann boundary condition with ID 1 applied to their local edge 2.

4.6.4 Output

In Figure 4.12, the solution is shown. At the Neumann boundaries, one can see the positive slope. At Dirichlet boundaries, the solution is set to $g(x)$.

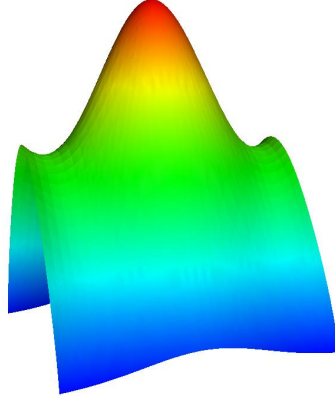


Figure 4.12: Solution of the problem with Neumann boundary conditions at two sides.

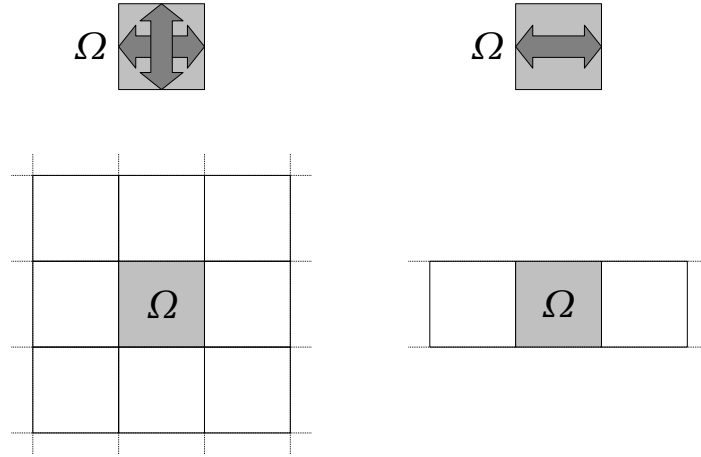


Figure 4.13: Two dimensional domain with periodic boundary conditions in both dimensions (left) and in only one dimension (right). In the first case, the solution at Ω can be propagated to the whole plane of \mathbb{R}^2 . In the second case, the solution only describes a band within \mathbb{R}^2 .

4.7 Periodic boundary conditions

Periodic boundary conditions allow to simulate an effectively infinite tiled domain, where the finite domain Ω is interpreted as one tile of the infinite problem domain. The solution outside of Ω can be constructed by periodically continuing the solution within Ω . In Figure 4.13 two examples for periodic boundary conditions on a two dimensional domain are illustrated. On the left hand side example, the upper and the lower part of the boundary as well as the left and the right part of the boundary are assigned to each other as periodic boundary. This results in a solution, which tiles the infinite plane. On the right hand side example, only the left and the right part of the domain boundary are assigned to each other, which results in an infinite band.

In AMDiS, there are two ways to implement periodic boundary conditions:

1. Changing the mesh topology (*mode 0*): Before the computation is started, the topology of the macro mesh is changed. Two vertices that are assigned to each other by a periodic boundary condition, are replaced by one single vertex, which is now treated as an inner vertex of the mesh (if it is not part of any other boundary). Since geometric data like coordinates are stored at elements and not at vertices, this modification does not change the

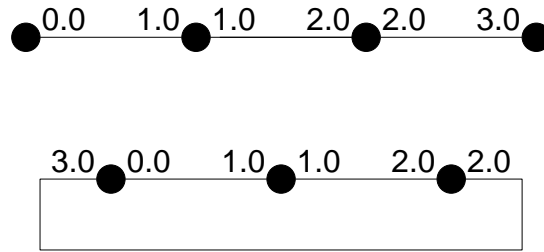


Figure 4.14: A one dimensional mesh with vertex coordinates stored at the elements and a corresponding periodic mesh with changed mesh topology. Note that the geometric data are not changed. The coordinates of the first vertex depend on the element it belongs to.

		1d	2d	3d
source code	src/	periodic.cc		
parameter file	init/	periodic.dat.1d	periodic.dat.2d	periodic.dat.3d
periodic file	init/	periodic.per.1d	periodic.per.2d	periodic.per..3d
macro file	macro/	periodic.macro.1d	periodic.macro.2d	periodic.macro.3d
output files	output/	periodic.mesh, periodic.dat		

Table 4.7: Files of the periodic example.

geometry of the problem. Topological information, like element neighborhood, does change. The method is illustrated in Figure 4.14.

2. Modify the linear system of equations in each iteration (*mode 1*): Sometimes it is necessary to store geometric information at vertices. E.g., if moving meshes are implemented with parametric elements, a DOF vector may store the coordinates. In this case, the mesh topology keeps unchanged, and the periodic boundary conditions are applied, like any other boundary condition, after the assemblage of the linear system of equations. In the application source code a boundary condition object has to be created, and in the macro file the periodic boundary must be specified.

In this section, we show how to use both variants of periodic boundary conditions. Again, we use the problem defined in Section 4.1. We choose $\Omega = [-0.2; 0.8] \times [-0.5; 0.5]$ (we do not use $\Omega = [-0.5; 0.5]^2$, because in this example periodic boundary conditions would then be equal to the trivial zero flux conditions).

We apply a periodic boundary condition which connects the left and the right edge of Ω . Since we do not know the exact solution of this periodic problem, we apply zero Dirichlet conditions at the lower and upper edge of the domain.

4.7.1 Source code

If we use *mode 0*, no modifications in the source code have to be made. For *mode 1*, we have to add a periodic boundary condition object to the problem.

```
periodic.addPeriodicBC(-1);
```

Note that periodic boundary conditions must be described by negative numbers.

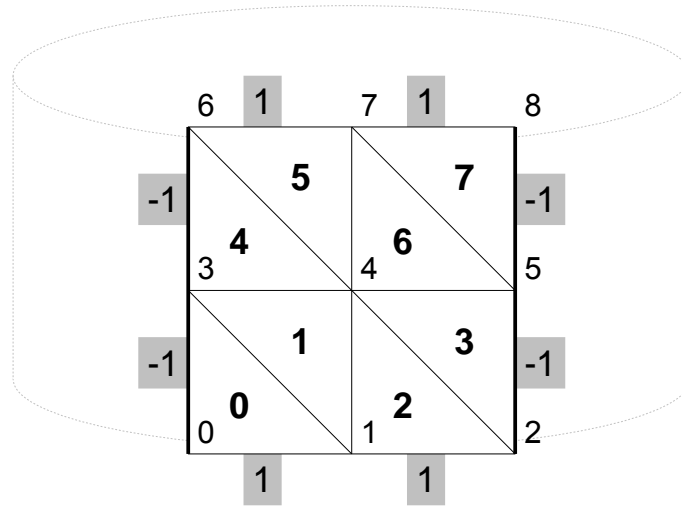


Figure 4.15: Macro mesh for the two dimensionaI periodic problem.

4.7.2 Parameter file

In the parameter file, we add an link to the *periodic file*.

```
periodicMesh->periodic file: ./init/periodic.per.2d
```

The periodic file `periodic.per.2d` contains the needed periodic information for the mesh.

associations: 2

mode	bc	el1	local	vertices	<->	el2	local	vertices
1	-1	4	1	2		7	2	1
1	-1	0	1	2		3	2	1

First, the number of edge associations (point associations in 2d, face associations in 3d) is given. Then each association is described in one line. The first entry is the mode which should be used for this periodic association. If the mode is 1, the next entry specifies the identifier of the used boundary condition. This identifier also must be used in the source code and in the macro file. If the mode is 0, the identifier is ignored. The rest of the line describes, which (local) vertices of which elements are associated with each other. The first association in this example is interpreted as follows: The local vertices 1 and 2 of element 4 are associated with the vertices 2 and 1 of element 7. Or more precisely, vertex 1 of element 4 is associated with vertex 2 of element 7, and vertex 2 of element 4 with vertex 1 of element 7.

4.7.3 Macro file

To avoid degenerated elements, one macro element must not contain two vertices which are associated with each other. Therefore, we choose a macro mesh with a few more elements, showed in Figure 4.15.

The corresponding file `periodic.macro.2d` looks like:

DIM: 2

DIM_OF_WORLD: 2

```
number of elements: 8
```

number of vertices : 9

element vertices :

```
1 3 0
3 1 4
2 4 1
4 2 5
4 6 3
6 4 7
5 7 4
7 5 8
```

element boundaries :

```
-1 1 0
0 0 0
0 1 0
-1 0 0
-1 0 0
0 1 0
0 0 0
-1 1 0
```

vertex coordinates :

```
-0.2 -0.5
0.3 -0.5
0.8 -0.5
-0.2 0.0
0.3 0.0
0.8 0.0
-0.2 0.5
0.3 0.5
0.8 0.5
```

element neighbours :

```
3 -1 1
2 4 0
1 -1 3
0 6 2
7 1 5
6 -1 4
5 3 7
4 -1 6
```

Compared to the macro file of Section 4.1.3, the vertex coordinates are shifted by -0.2 in x-direction.

In the boundary block `-1` specifies the periodic boundary. If we use *mode 0*, this boundaries are ignored (Here, the minus sign becomes important! Only boundary conditions with negative IDs are recognized as periodic boundaries and can be ignored if they are not used).

In the neighbors block, neighborships between elements that are connected by a periodic edge (point/face) are added. Note that this must also be done for *mode 1* periodic boundaries.

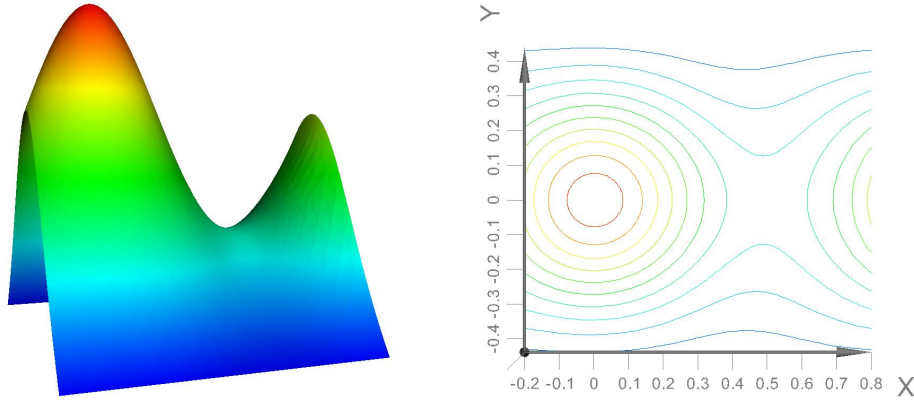


Figure 4.16: Solution of the problem with periodic boundary conditions at two sides (left) and iso lines of the solution for the values $0.1, 0.2, \dots, 1.1$ (right).

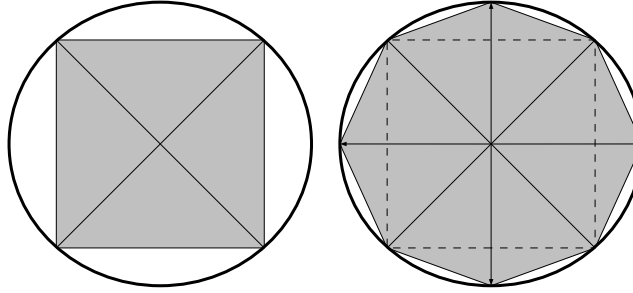


Figure 4.17: Boundary projection for a two dimensional mesh. The boundary vertices of the mesh are projected on the circle.

4.7.4 Output

In Figure 4.16, the solution of our periodic problem is shown as height field at the left hand side. At the right hand side, one can see iso lines for the values $0.1, 0.2, \dots, 1.1$.

4.8 Projections

In AMDiS, projections can be applied to the vertex coordinates of a mesh. There are two types of projections:

1. *Boundary projections*: Only vertices at the domain boundary are projected.
2. *Volume projections*: All vertices of the mesh are projected.

Projections are applied to all (boundary) vertices of the macro mesh and to each new (boundary) vertex, created during adaptive refinements. In Figure 4.17, this is illustrated for a two dimensional mesh which boundary vertices are successively projected to a circle. In Figure 4.18 the vertices of a one dimensional mesh are successively projected on the circle.

In this section, we give an example for both projection types. As projection we choose the projection to the unit sphere in 3d. In the first example, we start with the three dimensional cube $[-1, 1]^3$ and solve the three dimensional version of problem 4.1 in it. Furthermore, we apply a boundary projection to the unit sphere. In the second example, we set the right hand side f of

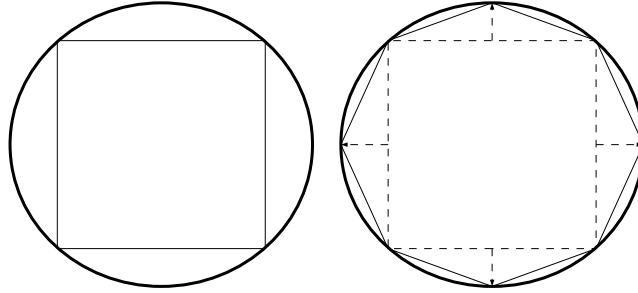


Figure 4.18: Volume projection for the one dimensional mesh. All vertices of this mesh are projected on the circle.

		1d	2d	3d
source code	src/			sphere.cc
parameter file	init/	-	-	sphere.dat.3d
macro file	macro/	-	-	sphere.macro.3d
output files	output/	sphere.mesh, sphere.dat		

Table 4.8: Files of the sphere example.

equation (4.1) to $2x_0$ (x_0 is the first component of x), and solve on the two dimensional surface of the sphere. Here, we start with a macro mesh that defines the surface of a cube and apply a volume projection to it.

4.8.1 Source code

First we define the projection by implementing a sub class `BallProject` of the base class `Projection`.

```
class BallProject : public Projection
{
public:
    BallProject(int id,
                ProjectionType type,
                WorldVector<double> &center,
                double radius)
        : Projection(id, type),
          center_(center),
          radius_(radius)
    {}
};
```

		1d	2d	3d
source code	src/			ball.cc
parameter file	init/	-	-	ball.dat.3d
macro file	macro/	-	-	ball.macro.3d
output files	output/	ball.mesh, ball.dat		

Table 4.9: Files of the ball example.

```

void project(WorldVector<double> &x)
{
    x -= center_;
    double norm = sqrt(x*x);
    TEST_EXIT(norm != 0.0)("can't project vector x\n");
    x *= radius_/norm;
    x += center_;
}

protected:
    WorldVector<double> center_;
    double radius_;
};

```

First, in the constructor, the base class constructor is called with a projection identifier and the projection type which can be `BOUNDARY_PROJECTION` or `VOLUME_PROJECTION`. The projection identifier is used to associated a projection instance to projections defined in the macro file. The method `project` implements the concrete projection of a point x in world coordinates.

If we compute on the surface, we redefine the function f .

```

class F : public AbstractFunction<double, WorldVector<double> >
{
public:
    F(int degree) : AbstractFunction<double, WorldVector<double> >(degree) {}

    double operator()(const WorldVector<double>& x) const
    {
        return -2.0 * x[0];
    }
};

```

In the main program, we create an instance of `BallProject` with ID 1, center 0 and radius 1. If we solve in the three dimensional volume of the sphere, the projection type is `BOUNDARY_PROJECTION`, because we project only boundary vertices to the sphere.

```

// ===== create projection =====
WorldVector<double> ballCenter;
ballCenter.set(0.0);
new BallProject(1,
                BOUNDARY_PROJECTION,
                ballCenter,
                1.0);

```

If we solve on the two dimensional surface of the sphere, the projection type is `VOLUME_PROJECTION`, because all vertices of the mesh are projected.

```

// ===== create projection =====
WorldVector<double> ballCenter;
ballCenter.set(0.0);
new BallProject(1,
                VOLUME_PROJECTION,
                ballCenter,
                1.0);

```

4.8.2 Parameter file

First, we present the parameter file for the volume projection case (two dimensional mesh).

```

dimension of world:      3

sphereMesh->macro file name:    ./macro/sphere_macro.3d
sphereMesh->global refinements:  10

sphere->mesh:              sphereMesh
sphere->dim:                2
sphere->polynomial degree:    1

sphere->solver:             cg
sphere->solver->max iteration: 100
sphere->solver->tolerance:    1.e-8
sphere->solver->left precon:  diag

sphere->estimator:         no
sphere->marker->strategy:    0

sphere->output->filename:     output/sphere
sphere->output->AMDiS format:  1
sphere->output->AMDiS mesh ext: .mesh
sphere->output->AMDiS data ext: .dat

```

The world dimension is 3, whereas the mesh dimension is set to 2. We use a macro mesh which defines the surface of a cube, defined in `./macro/sphere_macro.3d`, and apply 10 global refinements to it. In this example we do not use adaptivity. Thus, no estimator and no marker is used.

Now, we show the parameter file for the boudary projection case (three dimensional mesh).

```

dimension of world:      3

ballMesh->macro file name:    ./macro/macro.ball.3d
ballMesh->global refinements: 15

ball->mesh:               ballMesh
ball->dim:                 3
ball->polynomial degree:    1

ball->solver:             cg
ball->solver->max iteration: 1000
ball->solver->tolerance:    1.e-8
ball->solver->left precon:  diag

ball->estimator:         no
ball->marker->strategy:    0

ball->output->filename:     output/ball
ball->output->AMDiS format:  1
ball->output->AMDiS mesh ext: .mesh
ball->output->AMDiS data ext: .dat

```

The macro mesh is a three dimensional cube, defined in `./macro/macro.ball.3d`, and 15 times globally refined.

4.8.3 Macro file

First, the macro file for the two dimensional mesh.

```
DIM:          2
DIM_OF_WORLD: 3
```

```
number of vertices: 8
number of elements: 12
```

```
vertex coordinates:
-1.0    1.0   -1.0
 1.0    1.0   -1.0
 1.0    1.0    1.0
-1.0    1.0    1.0
-1.0   -1.0   -1.0
 1.0   -1.0   -1.0
 1.0   -1.0    1.0
-1.0   -1.0    1.0
```

```
element vertices:
3 1 0
1 3 2
2 5 1
5 2 6
6 4 5
4 6 7
7 0 4
0 7 3
2 7 6
7 2 3
1 4 0
4 1 5
```

```
element boundaries:
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
```

```
projections:
1 0 0
1 0 0
1 0 0
1 0 0
1 0 0
```



```

1 0 0
1 0 0
1 0 0
1 0 0
1 0 0
1 0 0
1 0 0
1 0 0

```

In the projections block, the projection IDs for each element are listed. There is one entry for each side of each element. Since we use volume projection, here, only the first entry of a line is used.

Now, we list the macro file for the three dimensional volume mesh.

```

DIM:          3
DIM_OF_WORLD: 3

```

```

number of vertices: 8
number of elements: 6

```

vertex coordinates :

```

-1.0 -1.0  0.0
 0.0 -1.0 -1.0
 0.0 -1.0  1.0
 1.0 -1.0  0.0
 0.0  1.0 -1.0
 1.0  1.0  0.0
-1.0  1.0  0.0
 0.0  1.0  1.0

```

element vertices :

```

0  5  4  1
0  5  3  1
0  5  3  2
0  5  4  6
0  5  7  6
0  5  7  2

```

element boundaries :

```

1  1  0  0
1  1  0  0
1  1  0  0
1  1  0  0
1  1  0  0
1  1  0  0

```

element neighbours :

```

-1  -1  1  3
-1  -1  0  2
-1  -1  5  1
-1  -1  4  0
-1  -1  3  5
-1  -1  2  4

```

projections :

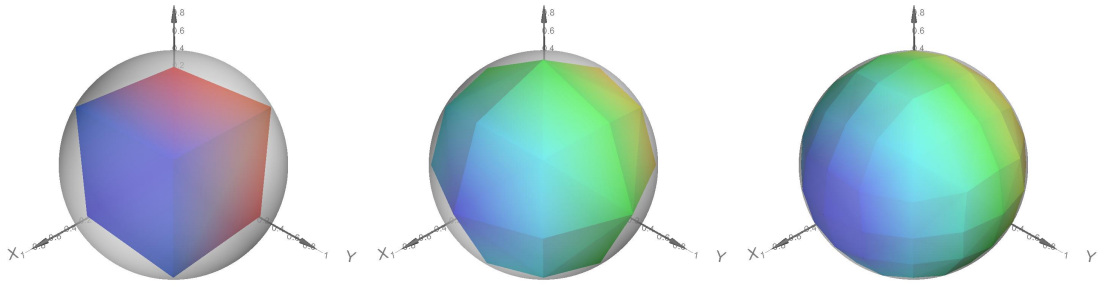


Figure 4.19: Surface of a cube successively refined and projected on the sphere.

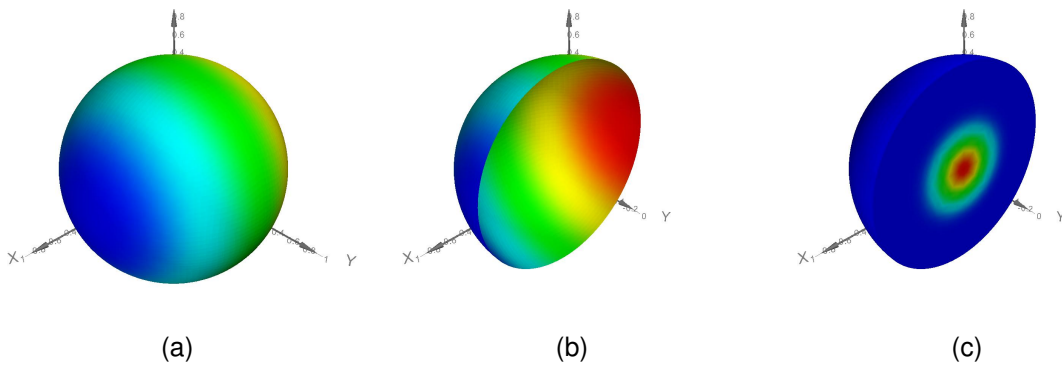


Figure 4.20: (a): Solution of the two dimensional problem on the surface of the sphere, (b): Halfed sphere, (c): Solution of the three dimensional problem (halfed ball).

1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0
1	1	0	0

Here, we use boundary projections. In the boundary block for each boundary side of an element the projection ID is given.

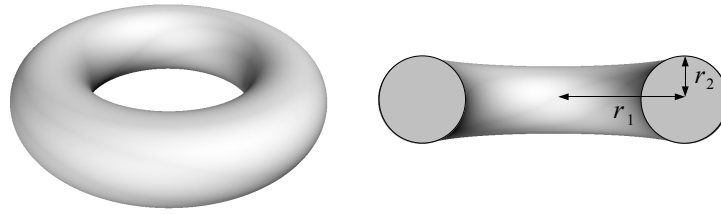
4.8.4 Output

In Figure 4.19, the solution of the two dimensional problem is shown on a successively refined mesh whose vertices are projected on the sphere. The finer the mesh, the better is the approximation to the sphere.

In Figure 4.20 (a), the final solution of the two dimensional problem is shown, Figure 4.20 (b) shows the halfed sphere to demonstrate that the solution is really defined on the sphere. The solution of the three dimensional problem is shown in Figure 4.20 (c).

4.9 Parametric elements

With parametric elements, problems can be solved on meshes which dimensions are not necessarily equal to the world dimension. Therefore, problems on arbitrary manifolds can be solved.

Figure 4.21: A torus and a halfed torus with the two radiuses r_1 and r_2 .

		1d	2d	3d
source code	src/			torus.cc
parameter file	init/	-	-	torus.dat.3d
macro file	macro/	-	-	torus.macro.3d
output files	output/	torus.mesh/.dat, rotation1.mesh/.dat, rotation2.mesh/.dat		

Table 4.10: Files of the torus example.

Furthermore, the vertex coordinates of the mesh can be flexible. Hence, moving meshes can be implemented.

In this section, we solve equation (4.1) with $f = 2x_0$ (x_0 is the first component of x) on a torus. Then we rotate the torus about the y -axis and solve the problem again.

The torus can be created by revolving a circle about an axis coplanar with the circle, which does not touch the circle. We call r_1 the radius of the revolved circle and r_2 the radius of the revolution, which is the distance of the center of the tube to the center of the torus. In Figure 4.21, a torus with its two radii r_1 and r_2 is shown.

We create a torus with center $(0; 0; 0)$ and the rotation axis in z -direction $(0; 0; 1)$. The projection of a point x_0 on the torus is implemented by the following steps:

1. x_1 is the projection of x_0 on the xy -plane
2. $x_2 = x_1 \frac{r_1}{\|x_1\|}$. Projection of x_1 on the sphere with radius r_1 with center 0. Thereby, x_2 is used as the center of a sphere with radius r_2 .
3. $x_3 = x_0 - x_2$. Move coordinate system into the center of the sphere with center x_2 . Thereby, x_3 contains the coordinates of x_0 in this new coordinate system.
4. $x_4 = x_3 \frac{r_2}{\|x_3\|}$. Thereby, x_4 is the projection of x_3 on the sphere with radius r_2 .
5. $x_5 = x_4 + x_2$. Thereby, x_5 contains the coordinates of x_4 in the original coordinate system. It is the projection of x_0 on the torus.

4.9.1 Source code

First, we define the rotation about the y -axis, which is used later to rotate the whole torus and the right hand side function.

```
class YRotation
{
public:
    static WorldVector<double>& rotate(WorldVector<double> &x, double angle)
    {
```

```

    double x0 = x[0] * cos(angle) + x[2] * sin(angle);
    x[2] = -x[0] * sin(angle) + x[2] * cos(angle);
    x[0] = x0;
    return x;
}
};

```

The right hand side function f has to follow the rotation of the torus.

```

class F : public AbstractFunction<double, WorldVector<double> >
{
public:
    F(int degree)
        : AbstractFunction<double, WorldVector<double> >(degree),
          rotation(0.0)
    {};

    double operator()(const WorldVector<double>& x) const
    {
        WorldVector<double> myX = x;
        YRotation::rotate(myX, -rotation);
        return -2.0 * myX[0];
    }

    void rotate(double r)
    {
        rotation += r;
    }

private:
    double rotation;
};

```

Every time, the mesh is rotated, the right hand side function will be informed over the method `rotate`.

Now, we implement the projection on the torus.

```

class TorusProject : public Projection
{
public:
    TorusProject(int id,
                 ProjectionType type,
                 double radius1,
                 double radius2)
        : Projection(id, type),
          radius1_(radius1),
          radius2_(radius2)
    {};

    virtual ~TorusProject() {}

    void project(WorldVector<double> &x)
    {
        WorldVector<double> xPlane = x;
        xPlane[2] = 0.0;
    }
};

```

```

double norm = std::sqrt(xPlane*xPlane);
TEST_EXIT(norm != 0.0)("can't project vector x\n");

WorldVector<double> center = xPlane;
center *= radius1_ / norm;

x -= center;

norm = std::sqrt(x*x);
TEST_EXIT(norm != 0.0)("can't project vector x\n");
x *= radius2_/norm;

x += center;
};

protected:
double radius1_;
double radius2_;
};

```

In the main program, we create a torus projection as `VOLUME_PROJECTION` with ID 1. The values of r_1 and r_2 are chosen, such that the resulting torus is completely surrounded by the macro mesh that is defined later.

```

int main(int argc, char* argv[])
{
    FUNCNAME("torus main");

    // ===== check for init file =====
    TEST_EXIT(argc == 2)("usage: torus initfile\n");

    // ===== init parameters =====
    Parameters::init(false, argv[1]);

    // ===== create projection =====
    double r2 = (1.5 - 1.0/std::sqrt(2.0)) / 2.0;
    double r1 = 1.0/std::sqrt(2.0) + r2;

    new TorusProject(1, VOLUME_PROJECTION, r1, r2);

    ...

    adapt->adapt();

    torus.writeFiles(adaptInfo, true);
}

```

The problem definition and the creation of the adaptation loop are done in the usual way (here, replaced by `...`). After the adaptation loop has returned, we write the result.

Before we let the torus rotate, some variables are defined. We set the rotation angle to $\frac{\pi}{3}$.

```

double rotation = M_PI/3.0;
int dim = torus.getMesh()->getDim();
int dow = Global::getGeo(WORLD);

```

```

DegreeOfFreedom dof;
WorldVector<double> x;

const FiniteElemSpace *feSpace = torus.getFESpace();
const BasisFunction *basFcts = feSpace->getBasisFcts();
int numBasFcts = basFcts->getNumber();
DegreeOfFreedom *localIndices = new double[numBasFcts];
DOFAdmin *admin = feSpace->getAdmin();

WorldVector<DOFVector<double>*> parametricCoords;
for (int i = 0; i < dow; i++)
    parametricCoords[i] = new DOFVector<double>(feSpace, "parametric coords");

```

In the next step, we store the rotated vertex coordinates of the mesh in `parametricCoords`, a vector of DOF vectors, where the first vector stores the first component of each vertex coordinate, and so on. In the STL map `visited`, we store which vertices have already been visited, to avoid multiple rotations of the same point.

```

std::map<DegreeOfFreedom, bool> visited;
TraverseStack stack;
ElInfo *elInfo = stack.traverseFirst(torus.getMesh(), -1,
                                     Mesh::CALL_LEAF_EL |
                                     Mesh::FILL_COORDS);

while (elInfo) {
    basFcts->getLocalIndices(elInfo->getElement(), admin, localIndices);
    for (int i = 0; i < dim + 1; i++) {
        dof = localIndices[i];
        x = elInfo->getCoord(i);
        YRotation::rotate(x, rotation);
        if (!visited[dof]) {
            for (int j = 0; j < dow; j++)
                (*(parametricCoords[j]))[dof] = x[j];

            visited[dof] = true;
        }
    }
    elInfo = stack.traverseNext(elInfo);
}

```

We create an instance of class `ParametricFirstOrder` which then is handed to the mesh. Now, in all future mesh traverses the vertex coordinates stored in `parametricCoords` are returned, instead of the original coordinates.

```

ParametricFirstOrder parametric(&parametricCoords);
torus.getMesh()->setParametric(&parametric);

```

We rotate the right hand side function, reset `adaptInfo` and start the adaptation loop again. Now, we compute the solution on the rotated torus, which then is written to the files `rotation1.mesh` and `rotation1.dat`.

```

f.rotate(rotation);
adaptInfo->reset();
adapt->adapt();

DataCollector *dc = new DataCollector(feSpace, torus.getSolution());
MacroWriter::writeMacro(dc, "output/rotation1.mesh");

```

```
ValueWriter::writeValues(dc, "output/rotation1.dat");
delete dc;
```

We perform another rotation. All we have to do is to modify the coordinates in `parametricCoords` and to inform `f` about the rotation.

```
visited.clear();
ellInfo = stack.traverseFirst(torus.getMesh(), -1,
                              Mesh::CALL_LEAF_EL | Mesh::FILL_COORDS);
while (ellInfo) {
    basFcts->getLocalIndices(ellInfo->getElement(), admin, localIndices);
    for (int i = 0; i < dim + 1; i++) {
        dof = localIndices[i];
        x = ellInfo->getCoord(i);
        YRotation::rotate(x, rotation);
        if (!visited[dof]) {
            for (int j = 0; j < dow; j++)
                (*(parametricCoords[j]))[dof] = x[j];

            visited[dof] = true;
        }
    }
    ellInfo = stack.traverseNext(ellInfo);
}

f.rotate(rotation);
adaptInfo->reset();
adapt->adapt();

dc = new DataCollector(feSpace, torus.getSolution());
MacroWriter::writeMacro(dc, "output/rotation1.mesh");
ValueWriter::writeValues(dc, "output/rotation1.dat");
delete dc;
```

The solution is written to `rotation2.mesh` and `rotation2.dat`.

Finally, we free some memory and finish the main program.

```
for (int i = 0; i < dow; i++)
    delete parametricCoords[i];
delete [] localIndices;
}
```

4.9.2 Parameter file

In the parameter file, we set the macro file to `./macro/torus_macro.3d`. This two dimensional mesh is 8 times globally refined and successively projected on the torus.

```
dimension of world:      3
```

```
torusMesh->macro file name:      ./macro/torus_macro.3d
torusMesh->global refinements:   8
```

```
torus->mesh:                torusMesh
torus->dim:                  2
torus->polynomial degree:    1
```

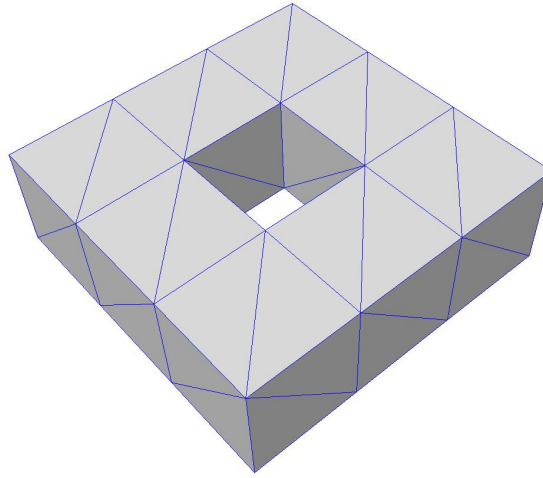


Figure 4.22: Macro mesh of the torus problem.

```

torus->solver:                cg
torus->solver->max iteration: 1000
torus->solver->tolerance:      1.e-8
torus->solver->left precon:    diag
torus->estimator:              no
torus->marker:                  no

torus->output->filename:        output/torus
torus->output->AMDiS format:    1
torus->output->AMDiS mesh ext:  .mesh
torus->output->AMDiS data ext:  .dat

```

4.9.3 Macro file

The macro mesh defined in `./macro/torus_macro.3d` is shown in Figure 4.22.

4.9.4 Output

In Figure 4.23, the solutions on the three tori are shown.

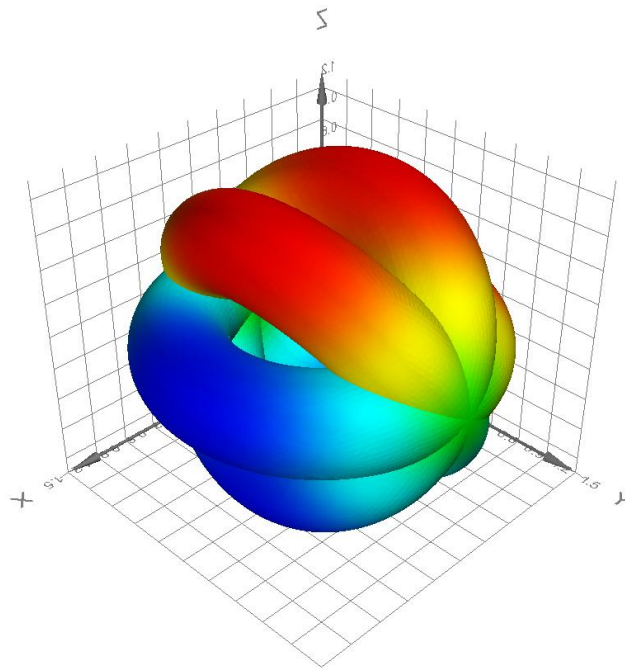


Figure 4.23: Solution on the original torus and on the two rotated tori.