AMDiS tutorial

Simon Vey, Thomas Witkowski

June 10, 2011

Contents

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 be the application at runtime. The name of the parameter file is usually passed to the 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 with *ParaView*. 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

There are two different ways to install AMDiS, depending if you have a binary packages downloaded from www.amdis-fem.org, or you have a source package. In the first case, you just need to run the package manager of your Linux distribution and install this package. To install AMDiS from the sources, you need CMake with version at least 2.8. Follow the hints in the file AMDiS-/Howto_cmake.html.

2.2 Compilation of an example application

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

1. Change into the demo directory:
   ```
   > cd demo
   ```

2. call cmake:
   ```
   > ccmake .
   ```

3. if AMDiS is not found automatically, set `AMDIS_DIR` to the directory of `AMDiS/share/amdis`

4. Make the application example:
   ```
   > make <PROG-NAME>
   ```

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

To run the example, call:
```
> ./<PROG-NAME> <PARAMETER-FILE>
```

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
USE_OPENMP = 0
USE_UMFPACK = 1
USE_MKL = 0
USE_SERVER =
USE_COMPILER = gcc
DEBUG = 0

AMDIS_DIR = ../AMDiS
MPI_DIR =
```

If `USE_PARALLEL_AMDIS` is set to 1, parallel applications will be supported. If AMDiS is compiled with OpenMP supported, `USE_OPENMP` must be set to 1. If you want to make use of UMFPACK oder Intel's Math Kernel Library, set `USE_UMPFACK` and `USE_MKL`, repsectively, to 1. In both cases, AMDiS must be compiled with the corresponding options. If you run your computations on one of the high performance computers installed at the TU Dresden, insert the name of the corresponding system to `USE_SERVER`. Using the flag `USE_COMPILER` you may decide to use either the GNU C++ or the Intel C++ compiler. Note that your program must be compiled with the same compiler AMDiS has been compiled with. To the last you can enable the debuge mode in your code setting `DEBUG` to 1.

`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. If you have not changed the directory structure of your AMDiS installation, or you do not want to use some other AMDiS installation for the demo examples, you do not need to change the standard value. The value of `MPI_DIR` is only required for parallel computations. If you run your parallel computations on the high performance computers of the TU Dresden, do not set a value here.

Then the AMDiS makefile is included from the corresponding AMDiSinstallation:

```
#==== standard definitions and rules to compile AMDiS user programs ==========
include $(AMDIS_DIR)/other/include/Makefile_AMDiS.mk
```

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

# ===== 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

$$
\begin{aligned}
-\Delta u &= f \quad \text{in } \Omega \subset \mathbb{R}^{dim} && (4.1)\\
u &= g \quad \text{on } \partial\Omega && (4.2)
\end{aligned}
$$

with

$$
\begin{aligned}
f(x) &= -\left(400x^2 - 20dow\right) e^{-10x^2} && (4.3)\\
g(x) &= e^{-10x^2}. && (4.4)
\end{aligned}
$$

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.

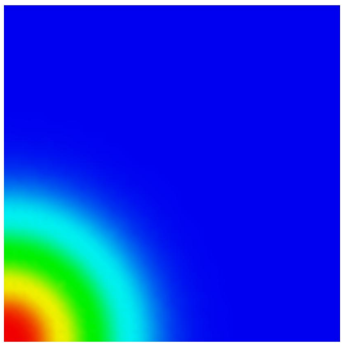


Figure 4.1: Solution of the Poisson equation on the unit square.

		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(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(argv[1]). The argument is the name of the parameter file.

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

```
  // ===== create and init the scalar problem =====
  ProblemStat 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, the required 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 stationary problem ellipt.

The operators now are defined as follows:

```
  // ===== create matrix operator =====
  Operator matrixOperator(ellipt.getFeSpace());
  matrixOperator.addSecondOrderTerm(new Simple_SOT);
  ellipt.addMatrixOperator(matrixOperator, 0, 0);

  // ===== create rhs operator =====
  int degree = ellipt.getFeSpace()->getBasisFcts()->getDegree();
  Operator rhsOperator(ellipt.getFeSpace());
  rhsOperator.addZeroOrderTerm(new CoordsAtQP_ZOT(new F(degree)));
  ellipt.addVectorOperator(rhsOperator, 0);
```

We define a matrix operator (left hand side operator) on the finite element space of the problem. The term $-\Delta u$ is added to it. Note that the minus sign isn't explicitly given, but implicitly contained in `Simple_SOT`. With `addMatrixOperator` we add the operator to the stationary problem definition. The both zeros represent the position of the operator in the operator matrix. As we are about to define a scalar equation, there is only the 0/0 position in the operator matrix. 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.

Now, we define boundary conditions:

```
  // ===== add boundary conditions =====
  ellipt.addDirichletBC(1, 0, 0, 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. As with adding operators to the operator matrix, we have to define the operator, on which the boundary condition will be applied. Thus we have to provide the matrix position indices after the boundary identifier.

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[0]:     1
ellipt->components:               1
```

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 1. The number of components, i.e. variables, in the equation is set to 1, since we are about to define a scalar PDE.

```
ellipt->solver:                        cg
ellipt->solver->max iteration:         1000
ellipt->solver->tolerance:             1.e-8
ellipt->solver->left precon:           diag
ellipt->solver->right precon:          no
```

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 left preconditioning, and no right preconditioning.

```
ellipt->estimator[0]:                     residual
ellipt->estimator[0]->error norm:         1
ellipt->estimator[0]->C0:                 0.1
ellipt->estimator[0]->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[0]->strategy:              2      % 0: no 1: GR 2: MS 3: ES 4:GERS
ellipt->marker[0]->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[0]->tolerance:              1e-4
ellipt->adapt[0]->refine bisections:      2

ellipt->adapt->max iteration:          10
```

The adaptation loop stops, when an error tolerance of 10^{-4} is reached, or after maximal 10 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->ParaView format:       1
```

The result is written in ParaView-format to the file `output/ellipt.vtu`.

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:

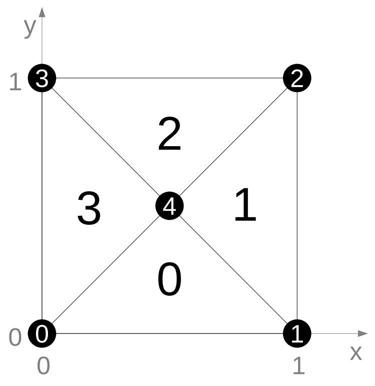


Figure 4.2: Two dimensional macro mesh

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

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.

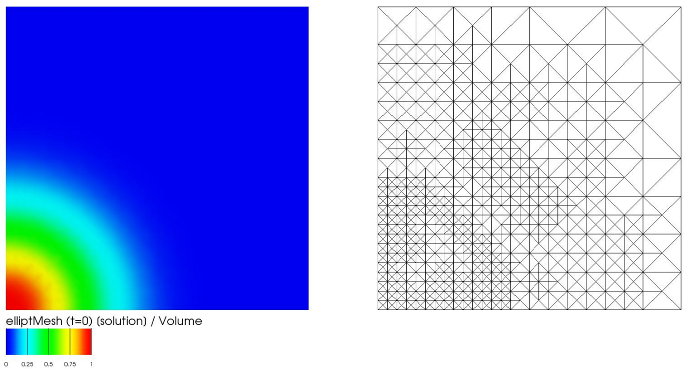
It follows the definition of boundary conditions:

```
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 neighborships. `-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
```



(a) (b)

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

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

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

$$\partial_t u - \Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^{dim} \times \left(t^{begin}, t^{end}\right) \tag{4.5}$$
$$u = g \quad \text{on } \partial\Omega \times \left(t^{begin}, t^{end}\right) \tag{4.6}$$
$$u = u_0 \quad \text{on } \Omega \times \left(t^{begin}\right). \tag{4.7}$$

We solve the problem in the time interval $\left(t^{begin}, t^{end}\right)$ 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

$$f(x,t) = \pi\cos(\pi t)e^{-10x^2} - \left(400x^2 - 20dow\right)\sin(\pi t)e^{-10x^2} \tag{4.8}$$
$$g(x,t) = \sin(\pi t)e^{-10x^2} \tag{4.9}$$
$$u_0(x) = \sin(\pi t^{begin})e^{-10x^2}. \tag{4.10}$$

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

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

$\tau = t^{new} - t^{old}$ 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. $\theta = 0.5$

		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.

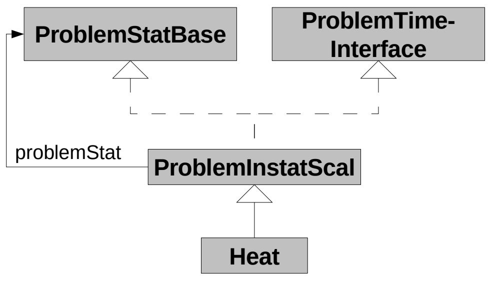


Figure 4.4: UML diagram for class Heat.

results in the Crank-Nicholson scheme. If we bring all terms that depend on u^{old} to the right hand side, the equation reads

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

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 ellipt 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 `ProblemInstat` 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 `ProblemInstat`. Actually, the initial solution is computed through the method `solveInitialProblem` of `ProblemTimeInterface`. But this method is implemented by `ProblemInstat` 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 ProblemInstat
{
public:
  Heat(ProblemStat &heatSpace)
    : ProblemInstat("heat", heatSpace)
  {
    theta = -1.0;
    Parameters::get(name + "->theta", theta);
    TEST_EXIT(theta >= 0)("theta not set!\n");
    theta1 = theta - 1.0;
  }
```

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 `ProblemInstat`. 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)
  {
    ProblemInstat::setTime(adaptInfo);
    rhsTime = adaptInfo->getTime() - (1 - theta) * adaptInfo->getTimestep();
  }

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

The method `setTime` is called by the adaptation loop to inform the problem about the current time. When this function is reimplemented, one should always call the function in the parent class, such that all time relavant variables in `ProblemInstat` will be updated. 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}, which is the current time.

The method `closeTimestep` is called at the end of each timestep by the adaptation loop. In the default implementation of `ProblemInstat::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 `ProblemInstat`. 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 *getRhsTimePtr() { return &rhsTime; };

private:
  double theta;
  double theta1;
  double rhsTime;
  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(argv[1]);
  Parameters::readArgv(argc, argv);

  // ===== create and init stationary problem =====
  ProblemStat 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 `ProblemStat`. `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 `ProblemStat`) to the adaptation loop. `heat` is interpreted as `ProblemTimeInterface` (implemented by class `ProblemInstat`).

The function g is declared in the following way:

```
// ===== create boundary functions =====
G *boundaryFct = new G;
boundaryFct->setTimePtr(heat.getTime());
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(heatSpace.getFeSpace());
A.addSecondOrderTerm(new Simple_SOT);
A.setUhOld(heat.getOldSolution(0));
if (*(heat.getThetaPtr()) != 0.0)
  heatSpace.addMatrixOperator(A, 0, 0, heat.getThetaPtr(), &one);
if (*(heat.getTheta1Ptr()) != 0.0)
  heatSpace.addVectorOperator(A, 0, 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(heatSpace.getFeSpace());
  C.addZeroOrderTerm(new Simple_ZOT);
  C.setUhOld(heat.getOldSolution(0));
  heatSpace.addMatrixOperator(C, 0, 0, heat.getInvTau(), heat.getInvTau());
  heatSpace.addVectorOperator(C, 0, heat.getInvTau(), heat.getInvTau());
```

The `Simple_ZOT` of operator `C` represents the zero order terms for the time discretization. On both sides of the equation u are added with the factor $\frac{1}{\tau}$ for both, assembling and error estimation. The inverse of the current timestep is returned by the function `getInvTau()`, which is a member of the class `ProblemInstat`.

```
  // create RHS operator
  Operator F(heatSpace.getFeSpace());
  F.addZeroOrderTerm(new CoordsAtQP_ZOT(rhsFct));
  heatSpace.addVectorOperator(F, 0);
```

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

Finally, the function g is created. This function is used for both, as the exact solution in the initial problem and as the Dirichlet boundary function. To the last, the adaption loop is started:

```
  // ===== create boundary functions =====
  G *boundaryFct = new G;
  boundaryFct->setTimePtr(heat.getTime());
  heat.setExactSolution(boundaryFct);
  heatSpace.addDirichletBC(1, 0, 0, boundaryFct);

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

Note that boundaries must be set after all operators were defined.

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 for the space and the time error are determined:

```
heat->adapt[0]->tolerance:               0.05
heat->adapt[0]->time tolerance:          0.05

heat->adapt->strategy:               1
```

If strategy is 0, an explicit time strategy with fixed timestep size is used. A value of 1 stands for the implicit strategy.

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[0]->coarsen allowed:         1
heat->adapt[0]->refine bisections:       2
heat->adapt[0]->coarsen bisections:      2
```

Now, the output behavior is determined:

```
heat->space->output->filename:                     output/heat
heat->space->output->ParaView format:              1
heat->space->output->ParaView animation:           1
heat->space->output->write every i-th timestep:    1
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 extension `.vtu`. 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 to the file `output/heat00.000.vtu`. If the parameter `ParaView animation` is enabled, AMDiS writes for the whole simulation one ParaView `pvd` file (in this case `output/heat.pvd`) including the names of all `vtu` files that were created. This file makes it very easy to view and analyze all the results of an instationary problem in ParaView.

Finally, we set parameter `WAIT` to 0. If the variable is set to 1, each call of the macro `WAIT` in the application will lead to an interruption of the program, until the `return` key is pressed.

```
WAIT:                                0
```

4.2.3 Macro file

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

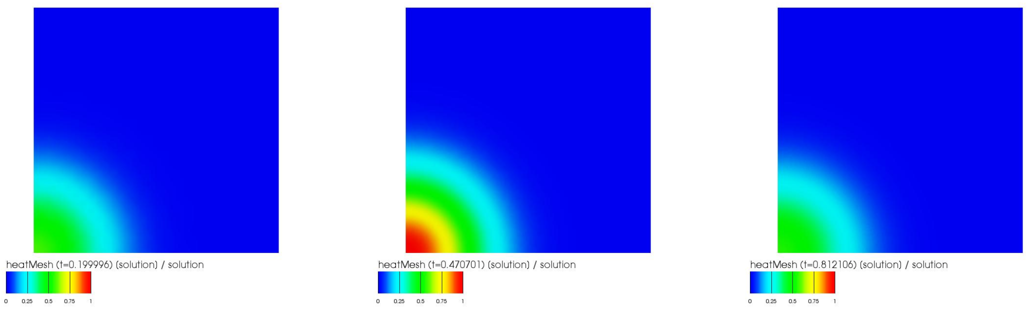
4.2.4 Output

As mentioned above, the output files look like `output/heat00.000.vtu`. 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

$$-\Delta u = f \tag{4.13}$$
$$u - v = 0. \tag{4.14}$$



$t \approx 0.2$ $t \approx 0.47$ $t \approx 0.81$

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.

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

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

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`:

```
ProblemStat 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(vecellipt.getFeSpace(0), vecellipt.getFeSpace(0));
matrixOperator00.addSecondOrderTerm(new Simple_SOT);
vecellipt.addMatrixOperator(&matrixOperator00, 0, 0);

int degree = vecellipt.getFeSpace(0)->getBasisFcts()->getDegree();
Operator rhsOperator0(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(vecellipt.getFeSpace(1), vecellipt.getFeSpace(0));
matrixOperator10.addZeroOrderTerm(new Simple_ZOT);
vecellipt.addMatrixOperator(matrixOperator10, 1, 0);

Operator matrixOperator11(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 both components.

```
vecellipt->components:              2

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

For most small and mid-size linear systems, direct solver perform much better than iterative ones. Therefore, we make use of the direct solver UMFPACK in this example.

```
vecellipt->solver:                  umfpack
```

All other solver parameters are than ommited because they need to be defined only for iterative solvers.

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.

		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 4.4: Files of the `neumann` example.

```
vecellipt->estimator[0]:            residual
vecellipt->estimator[1]:            0
```

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 10 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:    10

vecellipt->adapt[0]->tolerance:           1e-2
vecellipt->adapt[1]->tolerance:           1e-3
```

All components can be written into one file:

```
vecellipt->output->filename:              output/vecellipt
vecellipt->output->ParaView format:       1
```

As long as all FE spaces are equal, i.e., all components are discretized on the same mesh with the same basis functions, all components can and should be written to one file.

4.3.3 Macro file

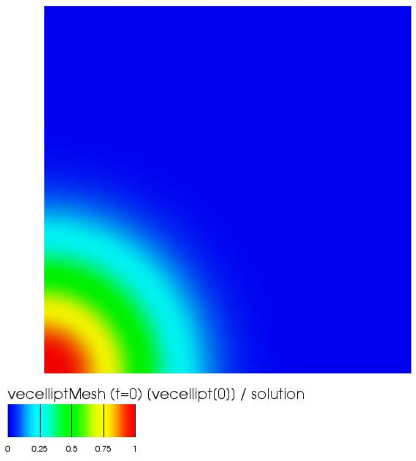
We again use the macro file `macro/macro.stand.2d`, which was described in Section 4.1.3.

4.3.4 Output

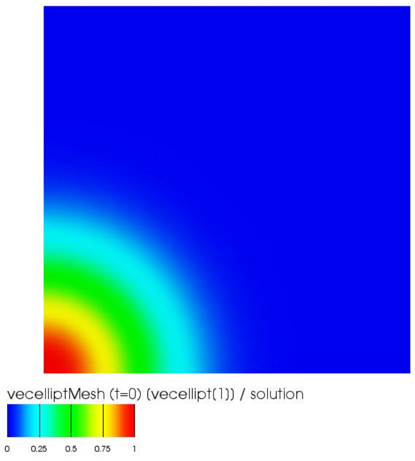
Component 0 of the solution (approximation of u) is written to the files `output/vecellipt0.mesh` and `output/vecellipt0.dat`. Component 1 of the solution (approximation of v) is written to the files `output/vecellipt1.mesh` and `output/vecellipt1.dat`. The two components are visualized in Figure 4.6.

4.4 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).



component 0



component 1

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

4.4.1 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, 0, 0, new N);
  neumann.addDirichletBC(2, 0, 0, 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.

4.4.2 Parameter file

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

4.4.3 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.

4.4.4 Output

In Figure 4.7, the solution is shown. At the Neumann boundaries, one can see the positive slope. At Dirichlet boundaries, the solution is set to $g(x)$.

4.5 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 infinte problem domain. The solution outside of Ω can be constructed by periodically continue the solution within Ω. In Figure 4.8 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 infinte 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 a infinte 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

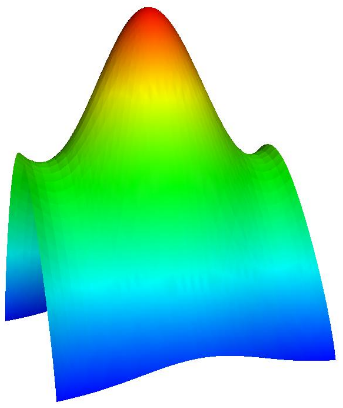


Figure 4.7: Solution of the problem with Neumann boundary conditions at two sides.

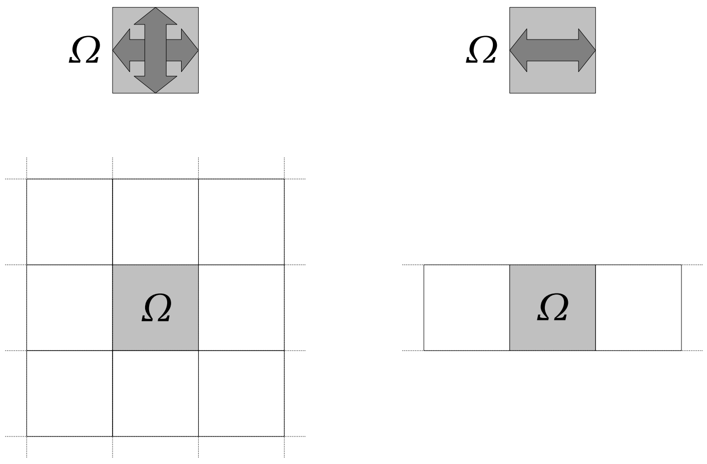


Figure 4.8: Two dimensional domain with periodic boundary conditions in both dimensions (left) and in only one dimension (right). In the first case, the solution at Ω can be propagated to the whole plane of $\mathbb{R}^2$. In the second case, the solution only describes a band within $\mathbb{R}^2$

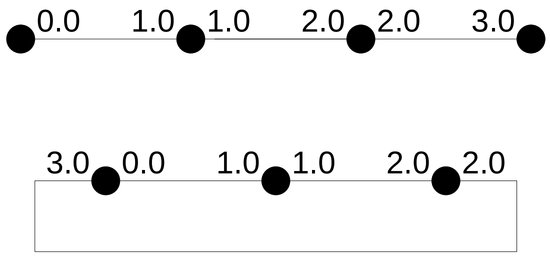


Figure 4.9: A one dimensional mesh with vertex coordinates stored at the elements and a corresponding periodic mesh with changed mesh topology. Note that the geometric data are not changed. The coordinates of the first vertex depend on the element it belongs to.

		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 4.5: Files of the `periodic` example.

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 geometry of the problem. Topological information, like element neighborhood, does change. The method is illustrated in Figure 4.9.

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.

4.5.1 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, 0, 0);
```

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

4.5.2 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

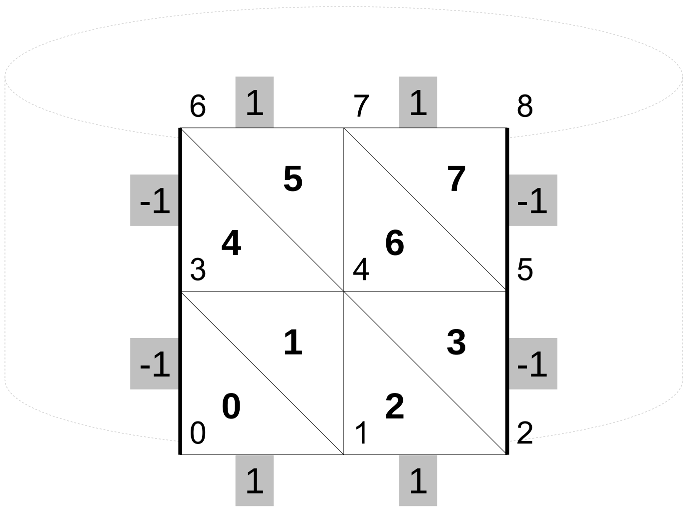


Figure 4.10: Macro mesh for the two dimensioanl periodic problem.

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.

4.5.3 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 4.10.

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.

4.5.4 Output

In Figure 4.11, 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 $0.1, 0.2, \ldots, 1.1$.

4.6 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 4.12, this is illustrated for a two dimensional

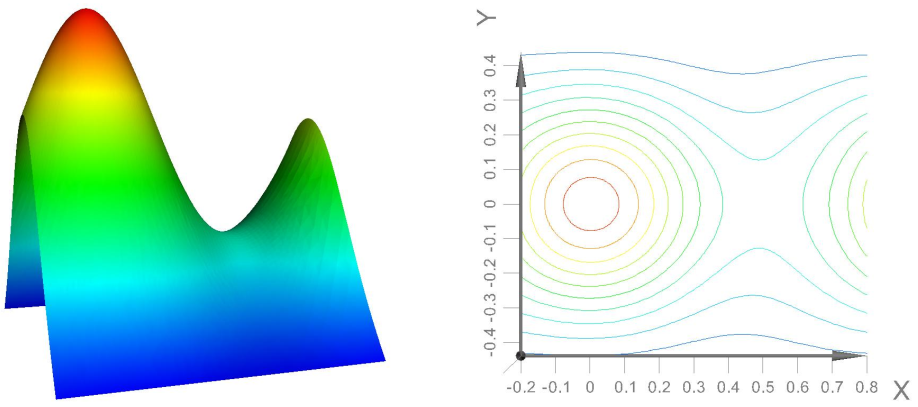


Figure 4.11: Solution of the problem with periodic boundary conditions at two sides (left) and iso lines of the solution for the values $0.1, 0.2, \ldots, 1.1$ (right).

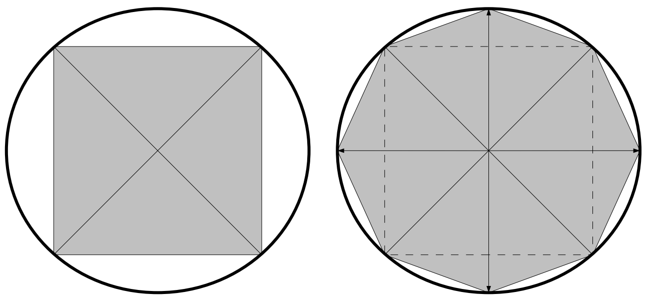


Figure 4.12: Boundary projection for a two dimensional mesh. The boundary vertices of the mesh are projected on the circle.

mesh which boundary vertices are successively projected to a circle. In Figure 4.13 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 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.

		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 4.6: Files of the `sphere` example.

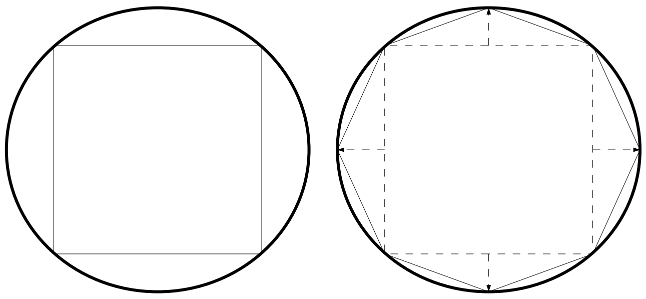


Figure 4.13: Volume projection for the one dimensional mesh. All vertices of this mesh are projected on the circle.

		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 4.7: Files of the `ball` example.

4.6.1 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)
  {}

  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);
```

4.6.2 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[0]:     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.

4.6.3 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
```

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:
  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.

4.6.4 Output

In Figure 4.14, 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 approxi-

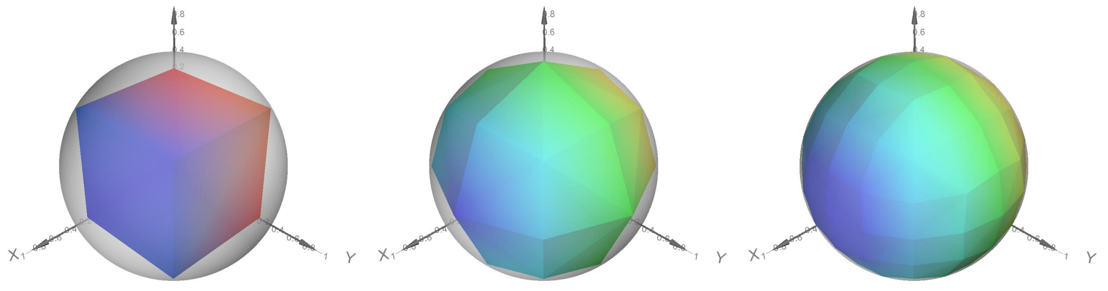


Figure 4.14: Surface of a cube successively refinend and projected on the sphere.

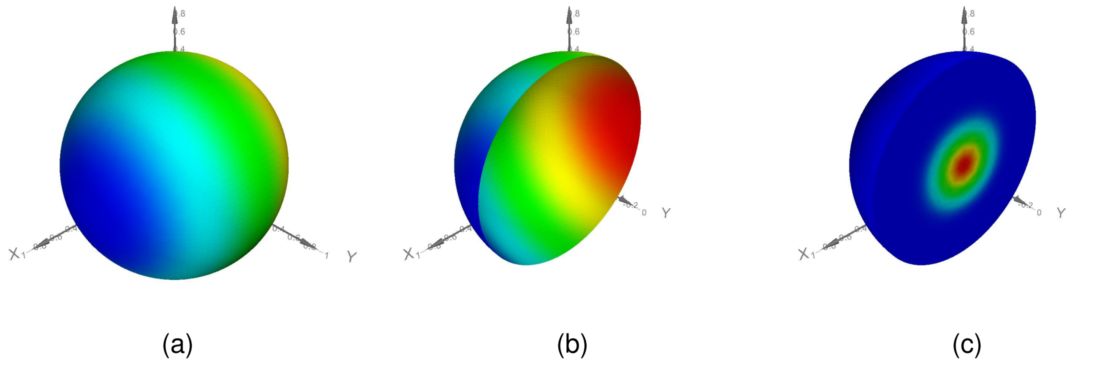


Figure 4.15: (a): Solution of the two dimensional problem on the surface of the sphere, (b): Halfed sphere, (c): Solution of the three dimensional problem (halfed ball).

mation to the sphere.

In Figure 4.15 (a), the final solution of the two dimensional problem is shown, Figure 4.15 (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 4.15 (c).

4.7 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. 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 4.16, 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.

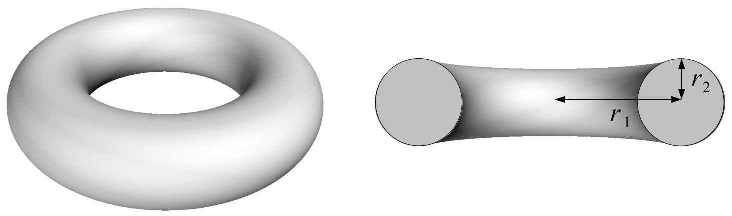


Figure 4.16: A torus and a halfed torus with the two radiuses r_1 and r_2.

		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 4.8: Files of the `torus` example.

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.

4.7.1 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(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();
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);
}

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;
}
```

4.7.2 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[0]:        1

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
```

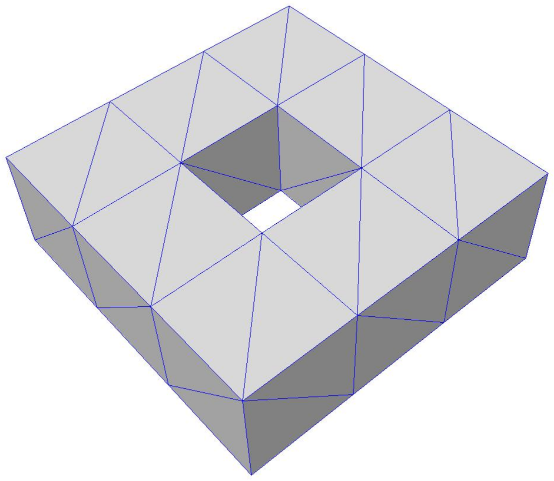


Figure 4.17: Macro mesh of the torus problem.

4.7.3 Macro file

The macro mesh defined in `./macro/torus_macro.3d` is shown in Figure 4.17.

4.7.4 Output

In Figure 4.18, the solutions on the three tori are shown.

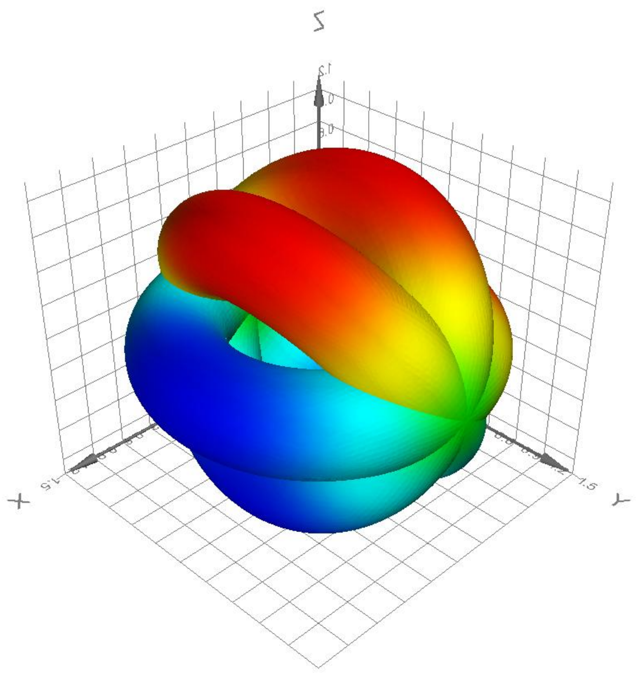


Figure 4.18: Solution on the original torus and on the two rotated tori.