NETGEN/NGSolve Manual

Appendix to the master thesis
"Numerische Lösungen elliptischer und parabolischer Differentialgleichungen in zwei und drei Dimensionen mit NETGEN/NGSolve"

Noam Arnold

March 8, 2013

Contents

1 Introduction

This manual explains how to use NETGEN 4.9.13 and NGSolve 4.9.13 for the FEM calculations used in my master's thesis. We first introduce examples of elliptic PDEs in two and three dimensions to then look at a non-stationary parabolic PDE. The creation of the geometries and the calculation of elliptic PDEs are well explained in the official manual. The error calculation, however, is not mentioned, and we will hence explain it in detail. Since there is no predefined solver for non-stationary parabolic PDEs, some basic C++ knowledge is required to program a parabolic solver. We will explain the code in the last chapter.

2 What Is NETGEN/NGSolve?

NETGEN is a mesh generation tool. It is an open source software available at sourceforge.net [9]. NETGEN has a graphical user interface and is linked with NGSolve. The official NETGEN manual is available in the folder of the source code [9] at doc/ng4.pdf. It offers detailed information on the history of the program, includes an installation guide and explains how to create different geometries and meshes. To solve a PDE, NETGEN uses NGSolve. The solution provided by NGSolve can be visualized by NETGEN on the mesh. NGSolve is a finite element library, which must be connected to a mesh handler. In our case of course, NETGEN will be the mesh handler. NGSolve is an open source program written in C++ and can be downloaded at sourceforge.net [10], where you also find a forum, the official manual [8], and a useful wiki [6]. In addition, you find examples, which help you understand NETGEN/NGSolve, in the lecture notes [7].

3 2D Elliptic PDEs in NETGEN

To test NETGEN, we will use the Poisson's equation. Let $\Omega = (0,1)^2$ be an open domain with boundary $\Gamma = \partial\Omega$. We look for a solution of

$$\begin{aligned} -\operatorname{div}(A\nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \Gamma. \end{aligned} \tag{1}$$

We look at the weak formulation of the Poisson's equation and create a triangulation of Ω: Let $\mathcal{G}$ be a triangulation of Ω. We define $S := S_0^{k,0} \subset H_0^1$ as the finite element space for $\mathcal{G}$ with polinomial order k. Find a $u_S \in S$ such that

$$a(u_S, v) := \int_\Omega A\nabla u_S \cdot \nabla v \, dx = \int_\Omega f v \, dx =: l(v) \tag{2}$$

for all $v \in S$.

To implement the Poisson's equation in NETGEN, we need to write three different files:

i) a geometry file which describes the domain Ω,

ii) a mesh file which describes the triangulation $\mathcal{G}$ on the domain, and

iii) a PDE file which describes the weak formulation (2) of the PDE.

How to create the geometry file and then generate the mesh file is well documented in the official NETGEN manual [9]. How to write the PDE file is explained in the official NGSolve manual [8] and in the NGSolve wiki [6].

3.1 The Geometry File for 2D

To implement $\Omega = (0,1)^2$ as a geometry file in NETGEN, we use 2D spline geometry with the extension ".in2d". This format allows us to create complex geometries with elliptic arcs and different domains. We will only explain simple geometries without mesh refinement. You find further information on mesh refinements in the official NETGEN manual.
In your NETGEN directory (where you installed NETGEN) you find examples of geometry files. We need the file square.in2d for our square $\Omega = (0,1)^2$. Listing 1 shows us square.in2d slightly changed in order for it to fit our Poisson's equation in (1).

```
1  splinecurves2dv2
   #grading-parameter
3  5

5  points
   #number  x_n     y_n
7  1        0       0
   2        1       0
9  3        1       1
   4        0       1
11
   segments
13 #dil     dir     line/arc   P_1     P_2     boundary condition
   1        0       2          1       2       -bc=1
15 1        0       2          2       3       -bc=1
   1        0       2          3       4       -bc=1
17 1        0       2          4       1       -bc=1

19 materials
   #number  name
21 1        dome1
```

Listing 1: square.in2d

All lines starting with # are comments describing the respective columns below them.

Line 1	"splinecurves2dv2" describes this file as a 2D spline geometry file.
Lines 2-3	The grading parameter = 5 describes how fast the mesh size decreases. The gradient of the local mesh size $h(x)$ is bounded by $\lvert\Delta_x h(x)\rvert \leq \text{grading}^{-1}$. In our case we get $\lvert\Delta_x h(x)\rvert \leq \frac{1}{5}$.
Lines 5-10	Points in the plain are described. In the first column we give each point a number. These point numbers will be used for the definition of the segments. In the second and the third column we define the x- and y-coordinates of the point.

Lines 12-17 The first column described as dil (domain is left) tells us the domain number of the domain to the left of the segment and the second column dir (domain is right) tells us the domain number of the domain to the right. The segments are defined by the points described above. Each segment goes from the point with point number P_1 (column four) to the point with point number P_2 (column five). Figure 1 illustrates this with the big arrows showing the described segments and the small arrows indicating the domain to the left and right respectively. The line/arc column describes the number of points of the segment, with 2 points meaning the segment is a line and three points meaning it is an elliptic arc. In the official manual, you find further information on how the elliptic arc is built [9]. The last column of the segments states the boundary conditions. As we have Dirichlet boundary all over Γ, all segments have the same boundary condition and all belong to bc=1.

Lines 19-21 The domains are defined here. Since we have just one domain in our example, there is just one entry. The outside is always zero. The domain numbers have to be the same as used in dil and dir in Line 12-17.

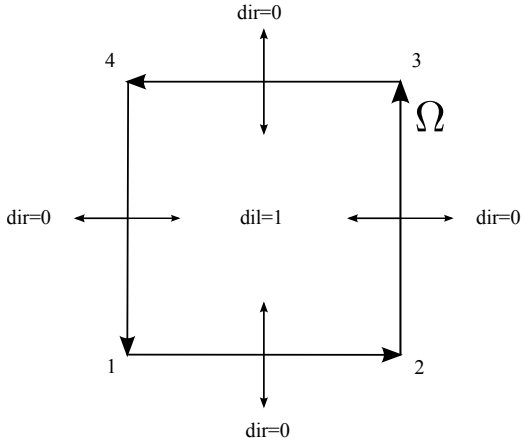

Figure 1: Explanation of square.in2d

3.2 The Mesh File in 2D

Once you have a geometry file, NETGEN can easily generate a mesh for you. You just open NETGEN, press "File/Load Geometry..." and choose the square.in2d file. Click on "Generate Mesh" and your first mesh is created. You can save it as square.vol with "File/Save Mesh...". If you want to refine your mesh, click on "Refinement/Refine uniform...".
The structure of the ".vol" format and all the special refinement options are explained in the official manual of NETGEN [9] and will not be explained here.

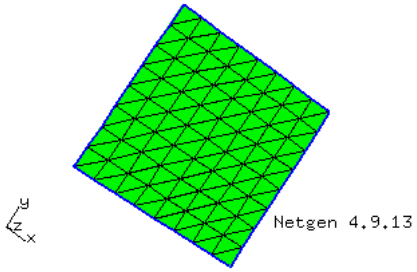

Figure 2: Mesh generated by NETGEN

3.3 The PDE File in 2D

How to write a PDE file is well documented in the official NGSolve manual [8]. It is strongly recommended to read the official NGSolve manual carefully. You find good examples of PDE files in the folder "pde-tutorial" in the downloaded zip file of the NGSolve source code [10]. Therefore, we will only sketchily describe the first part of the input file. The error calculation is not mentioned in the official NGSolve manual. We will hence focus on it. To be able to calculate the error, one must have an exact solution. For this purpose, we create an example for our Poisson's equation described in (1):

Example 1. *Let*

$$u_{exc} := xy(1-x)(1-y) \tag{3}$$

be the exact solution. For A we choose $A := 1 \cdot I$. Then we get

$$f = -\operatorname{div}(A\nabla u_{exc}) = -2(x^2 - x + y^2 - y).$$

We want to calculate the weak formulation (2) with the finite element space $S := S_0^{k,0} \subset H_0^1$ with order $k = 2$. With $A = 1$ and $f = -2(x^2 - x + y^2 - y)$ we can write the PDE file poisson.pde shown in Listing 2.

```
1  ###loading geometry and mesh#############################
   ## load geometry-file
3  geometry = square.in2d

5  ## load mesh-file
   mesh = square.vol
7  ##########################################################

9  ###defining bilinear form and linear form ################
   ## coefficient for the laplace-integral
11 define coefficient A
   1,
13
   ## coefficient for the source-integral
15 define coefficient f
   (-2*(x*x-x+y*y-y)),
17
   ## define a finite element space
19 define fespace v -type=h1ho -order=2 -dirichlet=[1]

21 ## the bilinear form of the weak formulation
   define bilinearform a -fespace=v -symmetric
23 laplace A

25 ## the linear form if the weak formulation
   define linearform l -fespace=v
```

```
27 source f
   ##########################################################
29
   #####creating solution#####################################
31 ## grid function to store the solution u
   define gridfunction u -fespace=v
33
   ## preconditioner "direct" means Cholesky factorisation
35 define preconditioner c -type=direct -bilinearform=a

37 ## numerical procedure to solve equation (boundary value problem)
   numproc bvp np1 -bilinearform=a -linearform=l -gridfunction=u -preconditioner=c -
       maxsteps=1000
39 ##########################################################
```

Listing 2: poisson.pde

In this example, we only need the *laplace* integrator for our bilinear form and the *source* for the integral of the linear form. Other integrals for the bilinear and the linear form are listed in the official NGSolve manual. You will also find other preconditioners and finite element spaces in the manual.

Lines 10-12 Since $A := 1 \cdot I$ there is no need to write A as a matrix in the PDE file. In fact, in NETGEN 4.9.13 there is no easy option to use a matrix for A. So far the only way is to code your own new Laplace integrator in C++.

Lines 18-19 The flag *-dirichlet=[n]* states that we have a Dirichlet boundary condition on the boundary segments with the flag -bc=n, defined in the geometry file square.in2d. In our case, as you can see in Listing 1, all boundary segments are bc=1, that means all of them have a Dirichlet boundary condition.

Line 38 The code *numproc bvp* (numerical procedure boundary value problem) gives NGSolve the order to calculate our PDE. We name it *np1*. The PDE with the bilinear form a and the linear from l is getting solved. The solution is stored in the grid function u. *numproc bvp* uses a CG-solver as default. The flag *-maxsteps* defines the maximal number of iterations (in our case the CG-iterations).

After we have written the PDE file, we let NETGEN read it. Open NETGEN, go to "Solve/Load PDE...", and choose poisson.pde. If the PDE is loaded, we should be able to see the mesh.
Now you only have to click on "Solve/Solve PDE". The solution is shown in colors, like in Figure 3. If your geometry is gray instead, you may need to click on "Visual" and there on "Scalar Function", and then choose your solution u. We will later provide additional information on the visualization options. Further information on the solution and the solving itself can be found in the terminal.

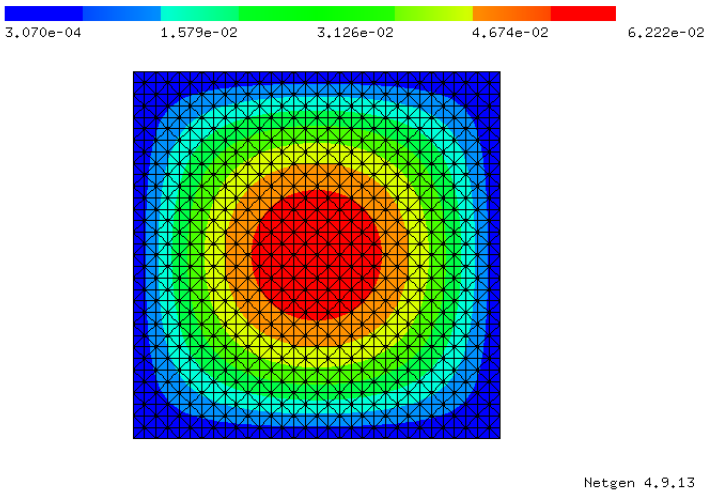

Figure 3: Solution of Poisson's equation (2) solved with NETGEN

The Error Calculation

We compute the error in the L^2-norm

$$\|u_{\text{exc}} - u_S\|_{L^2} = \sqrt{\int_\Omega (u_{\text{exc}} - u_S)^2 \, \mathrm{d}z}$$

and in the H^1-semi-norm

$$|u_{\text{exc}} - u_S|_{H^1} = \sqrt{\int_\Omega \left(\frac{\partial}{\partial x}(u_{\text{exc}} - u_S)\right)^2 \mathrm{d}z + \int_\Omega \left(\frac{\partial}{\partial y}(u_{\text{exc}} - u_S)\right)^2 \mathrm{d}z}.$$

For the H^1-semi-norm, we need the derivatives of u_{exc} from (3)

$$\nabla u_{\text{exc}} = \begin{pmatrix} y - 2xy - y^2 + 2xy^2 \\ x - x^2 - 2xy + 2x^2 y \end{pmatrix}.$$

To compute the error, we include the code of Listing 3 below at the end of the PDE file poisson.pde from Listing 2.

```
#####error calculation#################################
## coefficient for L^2 and H^1 scalar product
define coefficient one
1,

## exact solution to calculate L^2 error
define coefficient uexc
(x*(1-x)*y*(1-y)),

## gradient for exact solution to calculate H^1 error
define coefficient graduexc
((y-2*x*y-y*y+2*x*y*y),(x-x*x-2*x*y+2*x*x*y)),

## scalar product for L^2 norm
define bilinearform al2 -fespace=v -nonassemble
mass one

```

```
## scalar product for  H_1 seminorm
define bilinearform ah1 -fespace=v -nonassemble
laplace one

## vetcor space for the error grid functions
define fespace verr -l2ho -order=0

## grid functions to store the errors
define gridfunction errl2 -fespace=verr
define gridfunction errh1 -fespace=verr

## L-2 norm of error
numproc difference npdiffl2 -bilinearform1=al2 -solution=u -function=uexc -diff=
    errl2

## H_1 seminorm of error
numproc difference npdiffh1 -bilinearform1=ah1 -solution=u -function=graduexc -
    diff=errh1
#########################################################
```

Listing 3: The code for the error calculation

Lines 14-20 These bilinear forms are defined to compute the error with *numproc difference*. Since we do not assemble a stiffness matrix with them, one must add the flag *-nonassemble*.

Lines 29-34 *numproc difference* and many other *numprocs* are described in the NETGEN help section. To find it, you click on "Solve/Help/Numprocs.../numproc difference" in the NETGEN menu. You will see that you can use *numproc difference* to calculate other differences as well.

After we have loaded the PDE file into NETGEN and solved it, the error will show up in the terminal. If we want to plot the error on each element, more specific $\|u_{\text{exc}} - u_S\|^2_{L^2(\tau)}$ or $\|u_{\text{exc}} - u_S\|^2_{H^1(\tau)}$ where τ is a triangle, we simply click on "Visual", then click on "Scalar Function", and choose the error *errl2* or *errh1*. Figure 4 shows us $\|u_{\text{exc}} - u_S\|^2_{L^2(\tau)}$ for every triangle τ.

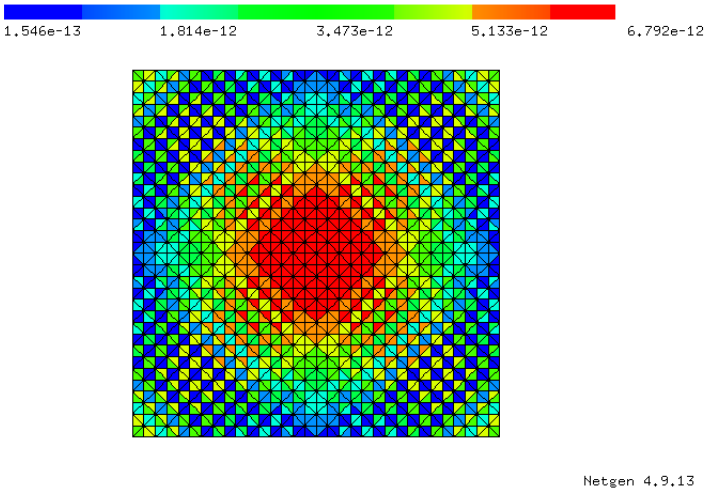

Figure 4: Square of the L^2 error on each triangle τ

4 3D Elliptic PDEs in NETGEN

In this chapter, we will look at the differences between a 2D and a 3D implementation. To test NETGEN, we will again use the Poisson's equation.

Let $\Omega = (-1,1)^3$ with $\Gamma = \partial\Omega$. The Neumann boundary Γ_N lies on the plane with the normal vector $\vec{n} := (1,0,0)^T$. The Dirichlet boundary is defined as $\Gamma_D = \Gamma \setminus \Gamma_N$. Let $\mathcal{G}$ be the triangulation of $\overline{\Omega}$. We define the finite element space as

$$S := S^{k,0} \cap \{u \in H^1(\Omega) : u|_{\Gamma_D} = 0\}$$

for $\mathcal{G}$ with polynomial order k.

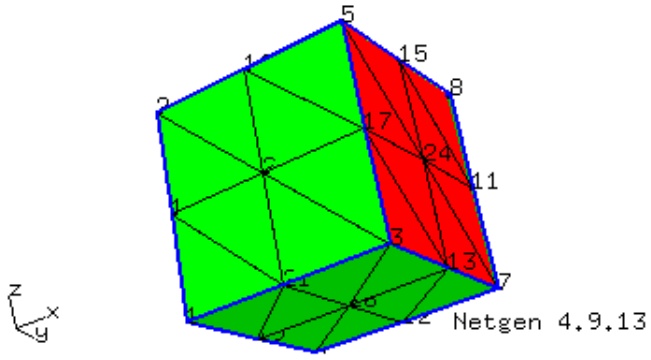

Figure 5: Ω with a mesh $\mathcal{G}$. Neumann and Dirichlet boundary

The weak formulation of our problem is as follows: Find $u_S \in S$ such that

$$a(u_S, v) := \int_\Omega A\nabla u_S \cdot \nabla v \, dx = \int_\Omega fv \, dx + \int_{\Gamma_N} \frac{g}{\beta} v \, ds =: l(v) \tag{4}$$

for all $v \in S$.

4.1 The Geometry File for 3D

To implement $\Omega = (-1,1)^3$ as a geometry file in NETGEN, we need to use a different file format than in the 2D example. We will use *Constructive Solid Geometry (CSG)* with the extension ".geo". The NETGEN manual (/doc/ng4.pdf from [9]) explains the CSG format well and in detail. We will hence only present our cube.geo file with brief comments:

```
algebraic3d

# cube consisting of 6 planes:
solid p1 = plane (-1, -1 ,-1; -1, 0, 0) -bc=1;
solid p2 = plane (-1, -1, -1; 0, -1, 0) -bc=1;
solid p3 = plane (-1, -1, -1; 0, 0, -1) -bc=1;
solid p4 = plane (1, 1, 1; 1, 0, 0) -bc=2;
solid p5 = plane (1, 1, 1; 0, 1, 0) -bc=1;
solid p6 = plane (1, 1, 1; 0, 0, 1) -bc=1;

solid cube = p1 and p2 and p3 and p4 and p5 and p6;

tlo cube;
```

Listing 4: cube.geo

Lines 4-9 A plane is described with $p = (x_0, y_0, z_0; n_x, n_y, n_z)$, where $(x_0, y_0, z_0) \in p$ is a randomly choosen point in the plane and $\vec{n} = (n_x, n_y, n_z)'$ the normal vector of the plane p. Unlike in the 2D example, we do need to define different boundary conditions for the example (4). For this purpose we set for Γ_D the flag -bc=1 and for Γ_N the flag -bc=2.

Line 13 tlo cube declares the solid cube as a top-level object. This is necessary for meshing.

4.2 The Mesh File in 3D

The 3D mesh generation works exactly the same way as the 2D example. Since we look at a 3D figure, we may want to look beyond the surface. This can easily be done through clipping. Click on "Visual" to open the Visualization window. Then click on "Clipping". Figure 6 shows you an example.

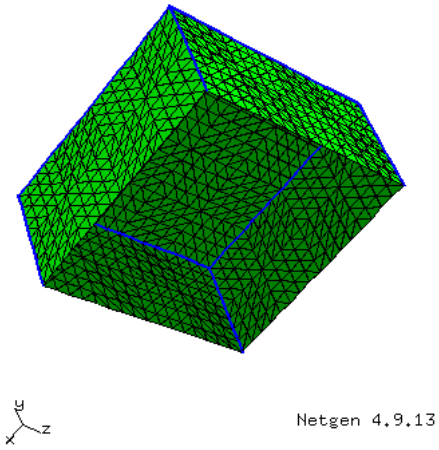

Figure 6: Mesh $\mathcal{G}$ cut with a plane

There is a video [2], which shows how the clipping works. NETGEN offers many tools to handle your meshes. You will find more general information in the NETGEN manual and in the NETGEN wiki [4], which also includes a video explaining how to modify a boundary condition in the graphic interface [1].

4.3 The PDE File in 3D

To be able to calculate the error, we create an example for our weak formulation of the Poisson's equation described in (4):

Example 2. *Let*

$$u_{exc} := (x+1)(y^2-1)(z^2-1)$$

be the exact solution for the chosen $\beta := 1, \quad A := 2 \cdot I$.
Then we get

$$\begin{aligned} f &= -\operatorname{div}(A \nabla u_{exc}) \\ &= -4(x+1)(x^2+y^2-2). \end{aligned}$$

We calculate g on Γ_N:

$$g = A\nabla u_{exc} \cdot \vec{n} = 2(y^2 - 1)(z^2 - 1).$$

We want to compute the weak formulation (4) with the finite element space $S := S^{k,0} \cap \{u \in H^1(\Omega) : u|_{\Gamma_D} = 0\}$ with order $k = 3$ and the parameters β, A, f, g as described above. For this purpose we write the PDE file poisson3D.pde shown in Listing 5:

```
geometry = cube.geo
mesh = cube.vol

## max memory usage
define constant heapsize = 100000000

## coefficient for laplace integral of bilinear form
define coefficient A
2,

## coefficient for source integral of linear form
define coefficient f
(-4*(x+1)*(z*z+y*y-2)),

## coefficient g/beta for Neumann integral of linear form
define coefficient g_beta
0, (2*(y*y-1)*(z*z-1)),

## finite element space with order=3 and Dirichlet-boundary at -bc=1
define fespace v -order=3 -dirichlet=[1]

## grid function to store solution
define gridfunction u -fespace=v

## linear form
define linearform f -fespace=v
source f
neumann g_beta

## bilinear form
define bilinearform a -fespace=v -symmetric
laplace A

## preconditioner
define preconditioner c -type=multigrid -bilinearform=a -smoothingsteps=1 -
    smoother=block -coarstype=cg -notest

## solving equation
numproc bvp np1 -bilinearform=a -linearform=f -gridfunction=u -preconditioner=c -
    maxsteps=500 -prec=1e-8

#######error computation#########################################
## coefficient for L^2 and H^1 scalar product
define coefficient one
1,

## exact solution to calculate L^2 error
define coefficient coef_uref
((x+1)*(y*y-1)*(z*z-1)),

## gradient of exact solution to calculate H^1 error
define coefficient coef_graduref
(((y*y-1)*(z*z-1)),(2*(x+1)*y*(z*z-1)),(2*(x+1)*(y*y-1)*z)),

## scalar product for L^2 norm
define bilinearform al2 -fespace=v -nonassemble
mass one

## scalar product for  H_1 seminorm
define bilinearform ah1 -fespace=v -nonassemble
laplace one

## vetcor space for the error grid functions
```

```
62 define fespace verr -l2ho -order=0

64 ## grid functions to store the errors
   define gridfunction errl -fespace=verr
66 define gridfunction errh -fespace=verr

68 ## L-2 norm of error
   numproc difference npdiffl2 -bilinearform1=al2 -solution=u -function=coef_uref -
       diff=errl
70
   ## H_1 seminorm of error
72 numproc difference npdiffh1 -bilinearform1=ah1 -solution=u -function=coef_graduref
       -diff=errh
   ######################################################################
74
   ## fits parameters for the visualization of the solution and error
76 numproc visualization npv1 -scalarfunction=u  -nolineartexture
```

Listing 5: poisson3D.pde

Lines 15-17 Since we have two boundaries (Γ_D and Γ_N) with two different boundary conditions, the integrals on the boundaries must have two entries. The first entry is for -bc=1 and the second entry for -bc=2. So the first entry is $\frac{g}{\beta}|_{\Gamma_D} = 0$ and the second entry is $\frac{g}{\beta}|_{\Gamma_N} = 2(y^2-1)(z^2-1)$.

Lines 40-70 The error calculation is the same as in the 2D example.

Lines 72-73 *numproc visualization* changes the visualization settings. You can also change these settings manually while looking at the solution. Click on "Visual" and you will find the settings in the Visualization window.

To solve our Poisson's equation, we load our PDE file into NETGEN and click on "Solve". Figure 7 shows us the solution on the cube cut by a plane. As already mentioned, this can be done in the "Visualization" window by clicking on "Clipping".

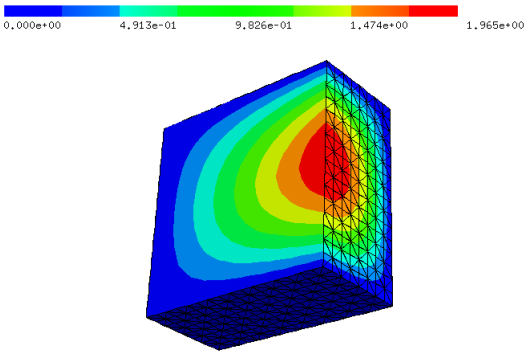

Figure 7: Solution of Poisson's equation (4) solved with NETGEN

5 2D Parabolic PDEs in NETGEN

In this chapter, we will look at the heat equation: Let Ω be an open domain and let $\Gamma = \partial\Omega$ be the boundary. Let $I = [0,T]$ be our time interval and $t \in I = [0,T]$ be the time variable. We look for the solution u of

$$\begin{aligned} \frac{\partial}{\partial t}u - \Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma \\ u|_{t=0} &= u^0. \end{aligned} \tag{5}$$

The weak formulation of the heat equation is:

Find a $u_S \in S = W_2^1(0,T;V,H)$ such that

$$\int_\Omega \frac{\partial}{\partial t}u_S v\,dx + \int_\Omega A\nabla u_S \cdot \nabla v\,dx = \int_\Omega f v\,dx \tag{6}$$

for all $v \in S$.

Attention: NGSolve does not offer a pre-built solution to solve this heat equation. To solve it, one must write a separate numerical procedure.

In our example, we will implement the simple backward Euler method [3, p. 137]. For this purpose we need to discretize the time interval. Split $I = [0,T]$ into p time intervals with constant length $\delta = t_{i+1} - t_i$. We will choose ϕ_i for the time-dependent linear basis functions, this means for $\{t_0,\dots,t_p\}$ we define

$$\phi_i(t) = \begin{cases} \frac{t-t_{i-1}}{t_i - t_{i-1}} & t \in [t_{i-1}, t_i) \\ \frac{t_{i+1}-t}{t_{i+1}-t_i} & t \in [t_i, t_{i+1}) \\ 0 & \text{else} \end{cases}.$$

We can describe $u(x,t)$ with the Rothe function

$$u_S^\delta(x,t) = \sum_{i=1}^{p} z_i(x)\phi_i(t).$$

For each time step t_i we will get $\frac{\partial}{\partial t}u_S = \frac{z_{i+1}-z_i}{\delta}$ and we can use the backward Euler method to solve the boundary value problem:

$$\left(\frac{z_{i+1}-z_i}{\delta}, v\right) + A(z_{i+1}, v) = (f_{i+1}, v),$$

Reordering the terms leads to the following recursion

$$\frac{1}{\delta}\int_\Omega z_{i+1} v\,dx + \int_\Omega \nabla z_{i+1}\nabla v\,dx = \int_\Omega f_{i+1} v\,dx + \frac{1}{\delta}\int_\Omega z_i v\,dx. \tag{7}$$

We will now look at the matrices we need to assemble. Under the assumption that the space discretization is independent from the time discretization, we define

$$z_i(x) = \sum_{j=1}^{N} \xi_{i,j}\varphi_j$$

with $\varphi_j \in V$ and $j \in \{1, \dots, N\}$, where N defines the number of inner grid points of the space discretization and V the the finite element space towards the space discretization. $\xi_{i,j} = z_i(x_j)$ with x_j as grid point of the space mesh. To create a implicit system of equations, we define the mass matrix

$$(M)_{kj} = (\varphi_k, \varphi_j)_{L^2(\Omega)},$$

the stiffness matrix

$$(A)_{kj} = (\nabla\varphi_k, \nabla\varphi_j)_{L^2(\Omega)},$$

and the load vector

$$(b_{i+1})_k = (f(t_{i+1}), \varphi_k)_{L^2(\Omega)},$$

with $k, j \in \{1, \dots, N\}$ and $\xi_i = (\xi_{i,j})_{j\in\{1,\dots,N\}}$. We then get for (7) the following implicit system of equations

$$(\frac{1}{\delta}M + A)\xi_{i+1} = b_{i+1} + \frac{1}{\delta}M\xi_i. \tag{8}$$

To code the iteration, we will choose an updater $\omega_{i+1} = \xi_{i+1} - \xi_i$. Therefor we add $\pm A\xi_i$:

$$\begin{aligned}
(\frac{1}{\delta}M + A)\xi_{i+1} &= b_{i+1} + A\xi_i + \frac{1}{\delta}M\xi_i - A\xi_i \\
(\frac{1}{\delta}M + A)\xi_{i+1} &= b_{i+1} + (A + \frac{1}{\delta}M)\xi_i - A\xi_i \\
(\frac{1}{\delta}M + A)\xi_{i+1} - (A + \frac{1}{\delta}M)\xi_i &= b_{i+1} - A\xi_i \\
(\frac{1}{\delta}M + A)(\xi_{i+1} - \xi_i) &= b_{i+1} - A\xi_i \\
\omega_{i+1} &= (\frac{1}{\delta}M + A)^{-1}(b_{i+1} - A\xi_i)
\end{aligned} \tag{9}$$

5.1 Coding with NGSolve

To code with NGSolve, it is recommended to download MyLittleNGSolve [5]. We will look at demo_instat.cpp and adapt it for our purpose. Listing 6 shows us an excerpt of instat_rothe.cpp, where changes have been made.

```
class NumProcParabolicRothe : public NumProc
{
protected:
  // bilinear form for the stiffness matrix
  BilinearForm * bfa;
  // bilinear form for the mass matrix
  BilinearForm * bfm;
  // linear form providing the right-hand side
  LinearForm * lff;
  // solution vector
  GridFunction * gfu;

  // time step
  double dt;
  // total time
  double tend;

public:

  /*
    In the constructor, the solver class gets the flags from the pde - input file.
    the PDE class apde constains all bilinear-forms, etc...
```

```
   */
24 NumProcParabolicRothe (PDE & apde, const Flags & flags)
     : NumProc (apde)
26 {
     // in the input file, you specify the bilinear forms for the stiffness and for
         the mass term
28   // like "-bilinearforma=k". Default arguments are 'a' and 'm'

30   bfa = pde.GetBilinearForm (flags.GetStringFlag ("bilinearforma", "a"));
     bfm = pde.GetBilinearForm (flags.GetStringFlag ("bilinearformm", "m"));
32   lff = pde.GetLinearForm (flags.GetStringFlag ("linearform", "f"));
     gfu = pde.GetGridFunction (flags.GetStringFlag ("gridfunction", "u"));
34
     dt = flags.GetNumFlag ("dt", 0.001);
36   tend = flags.GetNumFlag ("tend", 1);
   }
38
   virtual ~NumProcParabolicRothe()
40 { ; }


42
   // creates a solver object
44 static NumProc * Create (PDE & pde, const Flags & flags)
   {
46   return new NumProcParabolicRothe (pde, flags);
   }
48
   // solve at one level
50 virtual void Do(LocalHeap & lh)
   {
52   cout << "solve parabolic pde (rothe-method)" << endl;

54   // reference to the matrices provided by the bi-forms
     // will be of type SparseSymmetricMatrix<double> for scalar problems
56
     const BaseMatrix & mata = bfa->GetMatrix();
58   const BaseMatrix & matm = bfm->GetMatrix();
     const BaseVector & vecf = lff->GetVector();
60   BaseVector & vecu = gfu->GetVector();

62   // creates a matrix of the same type:
     BaseMatrix & summat = *matm.CreateMatrix();
64
     // creates additional vectors:
66   BaseVector & d = *vecu.CreateVector();
     BaseVector & w = *vecu.CreateVector();
68


70   // matrices matm and mata have the same memory layout. The arrays of values
     // can be accessed and manipulated as vectors:
72
    summat.AsVector() = (1.0/dt) * matm.AsVector() + mata.AsVector();
74
     // A sparse matrix can compute a sparse factorization.
76   BaseMatrix & invmat = * dynamic_cast<BaseSparseMatrix&> (summat).InverseMatrix
         ( gfu->GetFESpace().GetFreeDofs()  );

78   // implicite Euler-method
     vecu = 0;
80   for (double & t = pde.GetVariable("t",true); t <= tend; t += dt)
       {
82 cout << "t = " << t << endl;
       lff->Assemble(lh);
84 d = vecf - mata * vecu;
   w = invmat * d;
86 vecu += w;

88 // update visualization
   Ng_Redraw ();
90     }
   }
```

Listing 6: excerpt of instat_rothe.cpp

We explain the changes that have been made to demo_instat.cpp and explain the iteration of (8):

Lines 1, 24, 39	The name of the class in the code has been changed to NumProcParabolicRothe for our purpose.
Lines 43-47	The program is creating a solver object. NumProcParabolicRothe can now be used in the PDE file.
Lines 57-67	The program is assembling the matrices and creating the vectors.
Line 73	The program is calculating $(\frac{1}{\delta}M + A)$ from (9) as vectors.
Line 76	The program is calculating $(\frac{1}{\delta}M + A)^{-1}$ from (9). Since we have Dirichlet boundaries in our heat equation (5), we just want the free degrees of freedom (FreeDofs) to compute the inverse of the matrix.
Line 80	The "for" loop is for the iteration of (9) with the time step dt= δ and the end time T, which are described in the PDE file. We also get the variable t from the PDE file since we want to be able to use t as a variable for $f(x,t)$.
Line 84	$(b_{i+1} - A\xi_i)$ is calculated of (9).
Line 85	The updater $\omega_{i+1} = w$ is created.
Line 86	The updating $\xi_{i+1} = \xi_i + \omega_{i+1}$.
Line 89	Since we calculate a new solution in each loop step, we want to visualize the solution in each time step.

In the folder my_little_ngsolve we find the "Makefile" file. It facilitates an easy compilation of instant_rothe.cpp. Unix users can simply type in "make" into the terminal to create the shared library libmyngsolve.so. When we write our PDE file, this shared library will be included.

The Code for the Error Calculation

To implement the error calculation, we write an additional numeric procedure. We copy the code of *numrpoc difference* and simply add it to our time loop. We find the code of *numproc difference* in the source of NGSolve [10] at "solve/numprocee.cpp:171". The code of instat_rothe.cpp merged together with the code of *numproc differences* gives us Listing 7, a code for a new numerical procedure that we call *numproc parabolicdifference*. This procedure is calculating our heat equation and the error at the same time.

```
1  #include <solve.hpp>

3  using namespace ngsolve;

5  class NumProcParabolicDifference : public NumProc
   {
7  protected:
     // bilinear form for the stiffness matrix
9    BilinearForm * bfa;
     // bilinear form for the mass matrix
11   BilinearForm * bfm;
     // linear form providing the right-hand side
13   LinearForm * lff;
     // solution vector
15   GridFunction * gfu;

17   // time step
```

```
    double dt;
    // total time
    double tend;

    // bilinear form to calculate the difference
    BilinearForm * bfad;
    // coefficient function for the difference
    CoefficientFunction * coefd;
    // difference grid function
    GridFunction * gfdiff;

      // output to file
      string filename;
      ofstream * file;

public:
    NumProcParabolicDifference (PDE & apde, const Flags & flags)
      : NumProc (apde)
    {
      bfa = pde.GetBilinearForm (flags.GetStringFlag ("bilinearforma", "a"));
      bfm = pde.GetBilinearForm (flags.GetStringFlag ("bilinearformm", "m"));
      lff = pde.GetLinearForm (flags.GetStringFlag ("linearform", "f"));
      gfu = pde.GetGridFunction (flags.GetStringFlag ("gridfunction", "u"));

      dt = flags.GetNumFlag ("dt", 0.001);
      tend = flags.GetNumFlag ("tend", 1);

      bfad = pde.GetBilinearForm (flags.GetStringFlag ("bilinearformd", "adiff"));
      coefd = pde.GetCoefficientFunction(flags.GetStringFlag("function",""));
      gfu = pde.GetGridFunction (flags.GetStringFlag ("gridfunction", "u"));
      gfdiff = pde.GetGridFunction (flags.GetStringFlag ("diff", ""), 1);

      filename = pde.GetDirectory()+dirslash+flags.GetStringFlag("filename","");
      if (filename.length())
        file = new ofstream (filename.c_str());
      else
        file = 0;
    }

    virtual ~NumProcParabolicDifference()
    { ; }

    // creates a solver object
    static NumProc * Create (PDE & pde, const Flags & flags)
    {
      return new NumProcParabolicDifference (pde, flags);
    }

    // solve at one level
    virtual void Do(LocalHeap & lh)
    {
      cout << "solve parabolic difference pde" << endl;

    if (file)
        {
    (*file) <<"Domain   Points   Time  Difference " << endl;

        }

    // reference to the matrices provided by the bi-forms
      // will be of type SparseSymmetricMatrix<double> for scalar problems

      const BaseMatrix & mata = bfa->GetMatrix();
      const BaseMatrix & matm = bfm->GetMatrix();
      const BaseVector & vecf = lff->GetVector();
      BaseVector & vecu = gfu->GetVector();

      // creates a matrix of the same type:
      BaseMatrix & summat = *matm.CreateMatrix();

      // creates additional vectors:
      BaseVector & d = *vecu.CreateVector();
      BaseVector & w = *vecu.CreateVector();

```

```
    // matrices matm and mata have the same memory layout. The arrays of values
93  // can be accessed and manipulated as vectors:

95  summat.AsVector() = (1.0/dt) * matm.AsVector() + mata.AsVector();

97  // a sparse matrix can compute a sparse factorization
    BaseMatrix & invmat = * dynamic_cast<BaseSparseMatrix&> (summat).InverseMatrix
        ( gfu->GetFESpace().GetFreeDofs() );
99
    // implicite Euler-method
101 vecu = 0;
    for (double & t = pde.GetVariable("t",true); t <= tend; t += dt)
103   {
  cout << "t = " << t << endl;
105     lff->Assemble(lh);
  d = vecf - mata * vecu;
107 w = invmat * d;
  vecu += w;
109
// calculate the difference
111 cout << "Compute difference for t=" <<t << endl;

113 if (bfad->NumIntegrators() == 0)
      throw Exception ("Difference: bilinearformd needs an integrator");
115
  BilinearFormIntegrator * bfid = bfad->GetIntegrator(0);
117   FlatVector<double> diff =
      dynamic_cast<T_BaseVector<double>&> (gfdiff->GetVector()).FV();
119   diff = 0;

121   int ndom = ma.GetNDomains();

123   {
  for (int k = 0; k < ndom; k++)
125   CalcDifference (ma, *gfu, *bfid, coefd, diff, k, lh);
    }
127
    double sum = 0;
129 for (int i = 0; i < diff.Size(); i++)
      sum += diff(i);
131
    cout << " total difference  = " << sqrt (sum) << endl;
133


135 if (file)
      {
137 (*file)
      << ma.GetNLevels()
139   << "   " << bfad->GetFESpace().GetNDof()
      << "   " << t
141   << "   " << sqrt(sum) << endl;
      }
143


145
  // update visualization
147 Ng_Redraw ();
    }
149 }

151 virtual string GetClassName () const
  {
153   return "parabolic solver including error calculation";
  }
155
  virtual void PrintReport (ostream & ost)
157 {
    ost << GetClassName() << endl
159 << "Bilinear-form A = " << bfa->GetName() << endl
  << "Bilinear-form M = " << bfm->GetName() << endl
161 << "Linear-form     = " << lff->GetName() << endl
  << "Gridfunction    = " << gfu->GetName() << endl
163 << "dt              = " << dt << endl
  << "tend            = " << tend << endl
```

```
165   << "bilinearfromd    = " << bfad->GetName() << endl
      << "Gridfunction Difference  = " << gfdiff->GetName() << endl;
167   }

169   static void PrintDoc (ostream & ost)
      {
171     ost <<
          "\n\nNumproc Parabolic:\n" \
173       "-------------------\n" \
          "Solves a parabolic partial differential equation by an implicite Euler
               method\n\n" \
175       "Required flags:\n"
          "-bilinearforma=<bfname>\n"
177       "     bilinear-form providing the stiffness matrix\n" \
          "-bilinearformm=<bfname>\n"
179       "     bilinear-form providing the mass matrix\n" \
          "-linearform=<lfname>\n" \
181       "     linear-form providing the right hand side\n" \
          "-gridfunction=<gfname>\n" \
183       "     grid-function to store the solution vector\n"
          "-dt=<value>\n"
185       "     time step\n"
          "-tend=<value>\n"
187       "     total time\n"
          "-bilinaerformd=<bfname>\n"
189       "     bilinear-form for the norm of the error calculation\n"
          "-function=<coefficientfunction>\n"
191       "     coefficient function providing the referent solution\n"
          "-diff=<gfname>\n"
193       "     gridfunction to store the errors\n"
          "-filename=<filename>\n"
195       "     name of the file to store the errors\n"
      << endl;
197   }
};
199
// declare the numproc 'parabolicdifference'
201
namespace
203 #ifdef MACOS
demo_parabolic_cpp
205 #endif
{
207   class Init
      {
209   public:
        Init ();
211   };

213   Init::Init()
      {
215     GetNumProcs().AddNumProc ("parabolicdifference", NumProcParabolicDifference::
            Create, NumProcParabolicDifference::PrintDoc);
      }
217
      Init init;
219 }
```

Listing 7: excerpt of instat_rothe_difference.cpp

Line 23	Additional bilinear form for the L^2 or H^1 error is defined.
Line 25	Additional coefficient function for the exact solution u_{exc} is defined.
Line 27	Additional grid function to store the error is defined.
Lines 45-48	We define and store the additional *numproc parablicdifference* flags for the error calculation.
Lines 50-54	To be able to store the error in every time step we store a file if the *-filename* flag is used.

Lines 71-73	The first row of our table with the errors of each time step is written.
Lines 111-132	We are in the time loop and are calculating the difference for the time step t.
Lines 135-141	In every time step, we add the number of the domain, the number of points, the time and the error to the file created in lines 50-54.

5.2 The PDE File

The PDE file to solve the parabolic PDE differs only slightly from the other examples we looked at. Listing 8 shows us a PDE file without an error calculation. Once again, we need an example to implement.

Example 3. *Let $\Omega = [0, \pi]^2$ be our geometry. We choose as exact solution*

$$u_{exc} := (1 - e^{-2t}) \sin(x) \sin(y). \tag{10}$$

We verify that we have Dirichlet boundary conditions as in (5) and we get

$$f = 2 \sin(x) \sin(y).$$

As in (5) we choose

$$u|_{t=0} = 0$$

and for the time interval we choose $I = [0, 3]$.

The PDE file for the heat equation:

```
1  geometry = square_pi.in2d
   mesh = square_pi.vol
3
   ## reads the shared library "libmyngsolve.so" that you created
5  shared = libmyngsolve

7  ## starting time
   define variable t = 0
9
   define coefficient lam
11 1,

13 define coefficient rho
   1,
15
   define coefficient coef_source
17 (2*sin(x)*sin(y)),

19 define fespace v -type=h1ho -order=1 -dirichlet=[1]

21 define gridfunction u -fespace=v

23 define bilinearform a -fespace=v
   laplace lam
25
   define bilinearform m -fespace=v
27 mass rho

29 define linearform f -fespace=v
   source coef_source
31
   ##here the new programmed numproc is used
33 numproc parabolicrothe np1 -bilinearforma=a -bilinearformm=m -linearform=f -
       gridfunction=u -dt=0.00001 -tend=3
```

```
35 numproc visualization npv1 -scalarfunction=u -subdivision=2 -nolineartexture -
       deformationscale=1 -minval=0 -maxval=1
```

Listing 8: instat_rothe.pde

Lines 1-2	The geometry file and the mesh file are included as in the PDE files shown before. We just have to make sure that they fit for our new $\Omega = [0, \pi]^2$.
Line 5	This line is necessary to use the created shared library.
Line 8	In our simple example this line is not necessary. It is necessary if you want to use t as a variable, for instance for the right-hand side $f(x, t)$. Also, if we have a time interval $I = [t_0, T]$, we can set the starting time $t = t_0$ there.
Line 13	We use our self-written numproc from Listing 6.

Figure 8 shows us the solution of the heat equation (5) with f and $T = 3$ from example 3 at t=0.5, t=1, t=2, and t=3. At "Visual" we switch "Autoscale" off and set "Min-value"=-1 and "Max-value"=1. To have a 3D view, we switch "Deformation" on and set the view on the y-z plane.

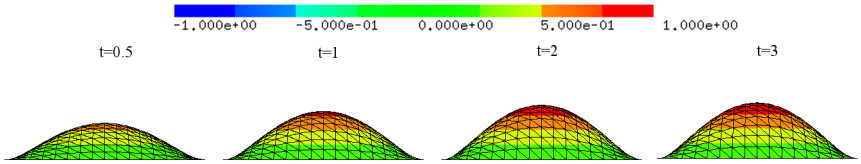

Figure 8: Solution of the heat equation (5) at t=0.5, t=1, t=2, t=3

The Error Calculation

This is how the PDE file which uses *numproc parabolicdifference* looks like:

```
1  geometry = square_pi.in2d
   mesh = square_pi.vol
3
   shared = myngsolve
5
   define variable t = 0
7
   define coefficient lam
9  1,
   28
11 define coefficient rho
   1,
13
   define coefficient coef_source
15 (2*sin(x)*sin(y)),

17 define fespace v -type=h1ho -order=1 -dirichlet=[1]

19 define gridfunction u -fespace=v

21 define bilinearform a -fespace=v
   laplace lam
23
   define bilinearform m -fespace=v
```

```
25 mass rho

27 define linearform f -fespace=v
   source coef_source
29
   #######error calculation#########################
31 ## coefficient for scalar products
   define coefficient one
33 1,

35 ## exact solution to calculate L^2-error
   define coefficient coef_uexc
37 ((1-exp(-2*(t)))*sin(x)*sin(y)),

39 ## scalar product for L^2-norm
   define bilinearform al2 -fespace=v -nonassemble
41 mass one

43 define fespace verr -l2ho -order=0
   define gridfunction errl2 -fespace=verr
45 ###################################################

47 numproc parabolicdifference np1 -bilinearforma=a -bilinearformm=m -linearform=f -
       gridfunction=u -dt=0.001 -tend=3 -bilinearformd=al2 -function=coef_uexc -diff
      =errl2 -filename=solutions.txt
```

Listing 9: instat_rothe_difference.pde

Lines 1-28	These lines are identical with the ones in Listing 8.
Lines 32-44	The coefficient function *coef_uref* is our exact solution from (10). The bilinear form *al2* defines the integrator for the L^2-norm of the error.
Line 47	We add the flag *-bilinearformd* to calculate the error in the requested norm, the flag *-function* to calculate the exact solution, the flag *-diff* for the grid function to store the solution, and the flag *-filename* to store the error of each time step in one file.

References

[1] Modifying Boundary Conditions of a Mesh. Available online at http://netgen-mesher.sourceforge.net/Docs/Screencasts/netgen_bc_001.swf; visited on December 28th 2012.

[2] Gerhard Gruber. A Simple Example. Available online at http://netgen-mesher.sourceforge.net/Docs/Screencasts/twocyl-example.swf; visited on December 28th 2012.

[3] Thomas Richter. Die Finite Elemente Methode für partielle Differentialgleichungen. Available online at numerik.uni-hd.de/~richter/SS11/numerik2/num2.pdf ; visited on October 10th 2012.

[4] Joachim Schöberl. NETGEN Wiki - Main Page. Available online at http://sourceforge.net/apps/mediawiki/netgen-mesher/index.php?title=Main_Page; visited on December 28th 2012.

[5] Joachim Schöberl. NGSolve - Aditional Files. Available online at http://sourceforge.net/projects/ngsolve/files/ngsolve/Additional%20Files/; visited on December 28th 2012.

[6] Joachim Schöberl. NGSolve Wiki - Main Page. Available online at http://sourceforge.net/apps/mediawiki/ngsolve/index.php?title=Main_Page; visited on December 28th 2012.

[7] Joachim Schöberl. Scientific Computing - Software Concepts for Solving Partial Differential Equations . Available online at www.asc.tuwien.ac.at/~schoeberl/wiki/lecture/scicomp.pdf; visited on August 17th 2012.

[8] Joachim Schöberl. Ngsolve 4.4 - Reference Manual, 2009. Available online at http://NGSolve.sourceforge.net/pderef/; visited on August 14th 2012.

[9] Joachim Schöberl. Netgen 4.X, 2010. Available online at http://sourceforge.net/projects/netgen-mesher/files/netgen-mesher/; visited on December 28th 2012.

[10] Joachim Schöberl. Ngsolve 4.X, 2010. Available online at http://sourceforge.net/projects/ngsolve/files/ngsolve/; visited on December 28th 2012.