PHP_FEM - a computational framework for **P**arallel **HP**-adaptive simulations with the **F**inite **E**lement **M**ethod

Krzysztof Banaś[1]

June 6, 2005

[1]pobanas@cyf-kr.edu.pl

Chapter 1

Introduction

PHP_FEM is a framework for parallel simulations with the adaptive finite element method. Its design is based on the modular architecture described in [1, 2, 3, 4]. The architecture should allow for easy management, modification and extension of the framework. The final aim of the development is to create a distributed environment for high performance parallel FEM simulations.

In the PHP_FEM framework it is assumed that any particular parallel FEM program consist of (at least) the following modules:

- mesh manipulation module
- field approximation module
- linear solver (or linear solver interface) module
- problem dependent module
- parallel execution module

The special feature of the proposed architecture is the fact that all modules except the parallel execution module are sequential modules modified as little as possible to make them fit into sequential programs as well as parallel programs.

The enclosed package contains several modules with interfaces conforming to the proposed architecture. These modules can be used to create prototype codes for solving two types of problems:

- Laplace's equation (a simple problem with small problem dependent part)
- stationary or time dependent linear convection-diffusion-reaction equations with Dirichlet, Neumann or mixed (Robin) boundary conditions

Apart from the two problem dependent modules there are the following modules:

- mesh manipulation module for 3D prismatic elements with mesh modification procedures allowing for full hp adaptivity
- field approximation module for discontinuous Galerkin approximations

- linear solver interface module that connects approximation and mesh modules with a linear solver module
- simple linear solver module that uses LAPACK dense matrix algorithms
- linear solver module that implements GMRES method with single and multi-level (multigrid) preconditioning
- parallel execution module in the form of the domain decomposition manager module
- parallel communication module providing interface with the PVM library routines

The prototype codes can be created in sequential and parallel versions. The last listed module is used only for parallel versions of the code. For sequential versions there exist a module with a single file containing dummy procedures conforming to the parallel communication interface.

The present report contains the user's manual for installing and running the code and briefly describes the particular modules with some information concerning their implementation. The detailed instructions for running several examples that test the capabilities of the code are included as well.

Chapter 2

The structure of the PHP_FEM code

The present chapter contains a sketch of the internal structure of the provided code. The code represents an implementation of certain algorithms and a particular version of the architecture both described in [4].

As it was stated in the introduction, the provided source files can be used to create several different executables for two different problems (simple elliptic and of convection-diffusion type). Each executable is built using object files from provided modules. Source files for each module (written in the C language) reside in separate subdirectories of the *src* directory. In the same subdirectory there is sometimes a header file for data structure and interface information internal to the module. For each module there is also an external interface in a header file contained in *include* subdirectory. The external interfaces are given exclusively in terms of constant parameters and function declarations with suitable parameter lists.

There is a naming convention adopted in the code. All external and global for a given module names of: directories, files, routines, data types, constant parameters and global (for a given module) variables start with two letters indicating the respective module, followed by a letter indicating the type of the named object: *d* for directories, *s* for source files, *h* for header files, *r* for routines, *t* for data types, *c* for constant parameters (written with all capital letters) or *v* for variables. Only the routine names and constant parameter names are made known to procedures external to a given module.

There are the following directories (in alphabetical order) with source and header files of the respective modules:

- *apd_dgscal_prism* – field approximation module for discontinuous Galerkin approximations using prismatic elements
- *ddd_manager* – domain decomposition manager providing the whole infrastructure for parallel computations
- *lsd_bliter* – linear solver module that implements GMRES method with single and multi-level (multigrid) preconditioning
- *lsd_lapack* – simple interface for the LAPACK library of linear algebra procedures (for dense matrices)

- *mmd_prism* – mesh manipulation module for 3D prismatic elements with mesh modification procedures allowing for full hp adaptivity
- *pcd_dummy* – dummy procedures imitating parallel communication routines for sequential versions of the code
- *pcd_pvm* – parallel communication module providing interface with the PVM library routines
- *pdd_conv_diff* – problem dependent module for stationary or time dependent linear convection-diffusion-reaction equations with Dirichlet, Neumann or mixed (Robin) boundary conditions (the module includes the submodule for the interface with linear solvers)
- *pdd_laplace* – problem dependent module for Laplace's equation with Dirichlet, Neumann or mixed (Robin) boundary conditions (the module includes the submodule for the interface with linear solvers)
- *utd_util* – simple utilities for all problem dependent modules

The particular feature of the proposed architecture is the use of the same mesh manipulation, field approximation and linear solver interface modules for sequential and parallel variations of the code. The details of the design of these modules is described in [5].

More complete specification of procedures from all modules is provided by header files in *include* and modules' subdirectories. All header files as well as all source files contains a lot of comments that explain details of the code.

2.1 Mesh manipulation module

In the current version of the code only prismatic elements are fully implemented. The master prismatic element is shown in Fig. 2.1. The figure presents local face, edge and node numbers together with local edge and face orientations. The numbering of faces and nodes corresponds to the array indexes in lists returned by data structure access routines. The orientations shown are used to specify the positions of faces with respect to elements and the positions of edges with respect to faces and elements. For each edge its orientation is given by the ordering of nodes in the data structure. For faces the orientations as well as local node and edge numbers are shown in Fig. 2.2. In the code it is assumed that the first node of a face has strictly defined position with respect to the element being its first neighbor. Namely, if the face has local number 1 than its first node is also the first node of the element. Subsequently for second face the first node is node 4, for the third face node 1, for the fourth face node 2 and for the fifth face node 3. However, it is allowed for the second neighbor of a face to have arbitrary relative position. Hence the data structure provides a possibility to indicate by how many positions (with respect to the numbering induced for the face by the local orientation of faces shown in Fig. 2.1) the first node of the face is shifted with respect to the position it would occupy if the element was the first neighbor. For example if the element presented in Fig. 2.1 is the second neighbor of its

fourth face and the first node of the face is the third element node then the shift is set to 1, if it is the sixth element node then the shift is set to 2, etc.

All elements are geometrically linear (or multi-linear), i.e. their position in space is obtained by a linear (or multi-linear) transformation from the master prismatic element shown in Fig. 2.1. The transformation is completely specified by coordinates of element's vertexes (these however has to be retrieved using faces' and edges' structures).

h-adaptivity implemented in the code consist of two ingredients: problem dependent methods to estimate or indicate the error of approximate solution and then apply adaptation strategy and mesh modification procedures that perform actual element divisions/clusterings. Depending upon adaptivity strategies and element types there may be different refinement as well as de-refinement kinds, such as: isotropic, anisotropic, regular, 1-irregular, irregular [6, 7].

At the current stage, there is one type of mesh refinement and de-refinement implemented in the code - the breaking of a single prismatic element into eight sub-elements ("sons") and clustering back the sons into their father. The breaking is isotropic (see Figures 3.1 and 3.2), proportions of edges and faces remain the same for the father and its sons. The discontinuous Galerkin formulation does not impose any regularity constraints on the mesh, so any element can be divided without considering the status of the neighboring elements. This allows for the existence in the mesh of very small elements being the neighbors of much larger elements. However, to make the transition between refined and not refined parts of the mesh smoother (this may improve the quality of approximation, e.g. for transient problems when refined regions follow moving fronts) the strategy has been implemented that does not allow the difference in generations between neighboring elements to pass the prescribed limit (parameter *max_gen_diff* specified in the input file). Currently the breaking procedure works with any value of *max_gen_diff* while the clustering can enforce only *max_gen_diff* equals to one.

The breaking of an element (and similarly clustering back to an element) is done in a hierarchical way, making use of the way in which geometrical data is stored in the code. To divide an element, first, its sides are divided, then new nodes, edges, and faces are created. To divide a face, first its edges are divided, then new nodes and edges are created. The geometrical construction of new objects is separated from reassigning values of degrees of freedom for new elements. The latter step is done after modifying the whole mesh and can be done separately for each problem, in case several problems share the same mesh. Degrees of freedom for new elements are obtained by the L^2 projection of old degrees of freedom (L^2 projection is used also for obtaining new degrees of freedom after clustering back elements into their father).

2.2 Field approximation module

There are currently two types of shape functions implemented for the master prismatic element (users can select any of them using input file parameters). The first type, *APC_TENSOR*, uses tensor products of 2D polynomials forming basis for complete polynomials in xy planes multiplied by 1D polynomials in z direction (e.g. for second order polynomials in z as well as in xy subspaces, there are the following shape functions:

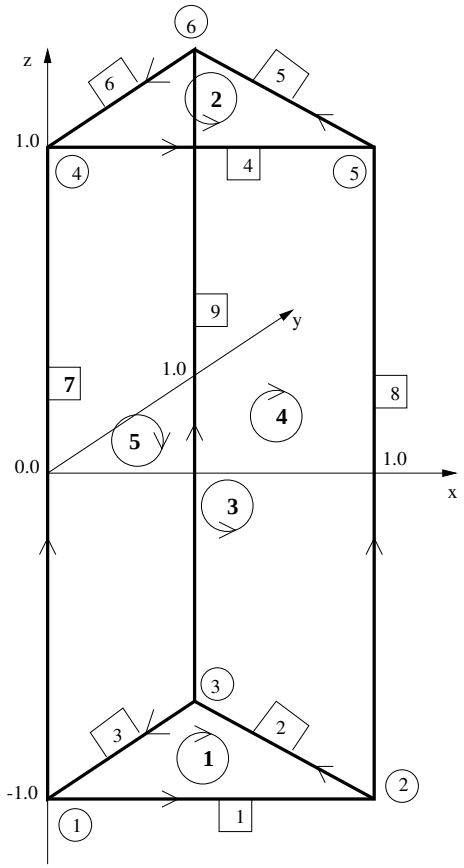

Figure 2.1: Master prismatic element

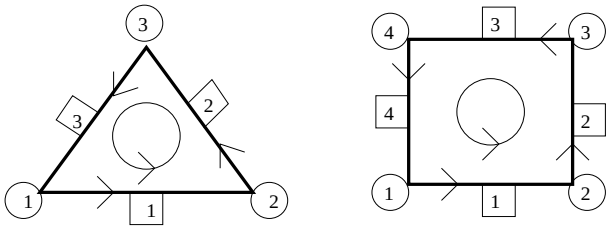

Figure 2.2: Triangular and quadrilateral faces

$1, x, y, z, xz, yz, xy, xyz, z^2, xz^2, yz^2, xyz^2, x^2, x^2z, x^2z^2, y^2, y^2z, y^2z^2$). For these shape functions the degree of approximation can be specified independently for xy plane and z direction. The second type of shape functions *APC_COMPLETE* uses polynomials forming a basis for complete polynomials of specified order in 3D (e.g. for second order polynomials there are the following shape functions: $1, x, y, z, xz, yz, xy, z^2, x^2, y^2$).

2.3 Problem dependent modules

It is assumed in the proposed architecture for FEM codes that problem dependent modules provide routines for time integration, nonlinear equations solution, adaptation strategy (with error estimation), interface with linear solver and returning the coefficients of a particular PDE solved. In the included release the convection-diffusion-reaction module is the one with richer structure and most of above mentioned ingredients.

The time integration routine in the basic setting of PHP_FEM consist of a loop over time steps with a constant time step length. At each time step a linear equations solver is called and then, at user specified intervals, slope limiting and/or the adaptation of the mesh is performed.

The linear equations solver calls routines that compute contributions to element stiffness matrices and load vectors. Formation of element stiffness matrices and load vectors consist of numerical integration of suitable terms from the discontinuous Galerkin formulations. In the code the integration is performed using the suitable approximation module routines separately inside elements, using 3D Gaussian quadratures, and over active element faces, using 2D quadratures for triangles and quadrilaterals. The 2D integration is performed for all active faces with no assumptions concerning the size of neighboring elements. In this sense meshes can be irregular (neighboring elements can have different sizes and vertexes) and approximation can be non-conforming. There is however a constraint, that initial meshes, although possibly fully unstructured, must be regular (without "hanging nodes").

At the current stage there is only one basic strategy implemented for h refinements/derefinements of the mesh. The strategy uses the value of an error indicator provided for each element. If the value exceeds the limit ϵ then the element is refined. The value of ϵ is computed as $\epsilon = (GLOB\_TOL)^2/N_{el}$ where $GLOB\_TOL$ is a user specified global tolerance for error, and N_{el} is the number of active elements in the mesh. If the sum of error indicators in a family of elements obtained by the division of a single element is less then some user specified fraction of ϵ, then the whole family is clustered back to the father element. The aim of the strategy is to obtain the global error indicator (the sum of element indicators) less than a prescribed limit with an approximate equidistribution of errors among elements. There is, additionally, a limit, the value of parameter max_gen, imposed on a maximal generation level for any element. There is no error estimation supplied in the default version of the code, all adaptations for test examples are done using the knowledge of the exact solution.

Even with the basic setting of the code for convection-diffusion-reaction equations it is possible to perform a broad range of simulations for, time-dependent or stationary, linear problems by modifying only internal routines that specify the coefficients of equations.

There exists a routine to read (if necessary) coefficients of a PDE for a particular run from a file in the working directory. There is also a mechanism to specify different material types and material coefficients for each element. Additional routines may be supplied to compute more complicated non-linear coefficients. It is envisaged that more complicated problems may require, apart from modifying coefficient procedures, also the changes in numerical integration routines as well as creation of additional procedures (e.g. for solving nonlinear equations). Finally, changes in finite element formulation may result in deeper modifications, including possible alterations to problem data structures.

PHP_FEM code is distributed with a suite of example problems. For some of them the exact solution is known. The procedure *pdr_error_test* computes the L^2 norm and the H^1 semi-norm (as the sum of element contributions, not taking into account discontinuities across elements) for several solutions to test problems (see Chapter 5).

2.4 Linear equations solver

The linear equations solver is an iterative solver built around procedures acting on the global stiffness matrix stored in the block format described in [4]. The data structure used by the solver is specially designed for solving systems of linear equations arising from finite element discretizations using multi-level methods.

The global stiffness matrix is stored in elementary block structures. Since in the discontinuous Galerkin approximations degrees of freedom are associated exclusively with elements, there is a one-to-one correspondence between elements and elementary blocks. In essence a single block structure stores data corresponding to several rows of the system of linear equations. The implied storage scheme for the system matrix is the block compressed row scheme.

For the purpose of preconditioning the solver defines another block data structure, designated to store preconditioner matrices. Each preconditioner block may correspond to a single or multiple elementary blocks, i.e. to a single element or a small subdomain. The subarrays of the stiffness matrix are either the same as for elementary blocks (in the former case), or need additional assembling for the subdomains case. Preconditioner blocks may have common parts corresponding to overlapping subdomains.

2.5 Domain decomposition module

This module takes care of the parallel execution of the code. It provides routines for mesh partition, load balancing, mesh transfer and the interface with parallel linear equations solvers. The details of its structure and functions are described in [1, 2, 3, 4].

2.6 Parallel communication module

There exist one true parallel communication module for PVM environment. The model of parallel execution is however in the style of SPMD, making the use of MPI an easy to implement alternative.

There is only one parallel executable to be run on each processor of the parallel machine. The processes exchange data using messages that are first packed into buffers and then sent to their destination. There are special procedures for group communication (gather, scatter, etc.)

In order to allow for SPMD style with PVM a special executable *pvm_run* is created with a purpose of starting the proper executables on processors of the parallel system. The source code for the *pvm_run* command is in the subdirectory *pvm_run* of the PVM parallel communication directory *pcd_pvm*. The command has the following usage:

```
usage:  pvm_run -np X program_name
```

with X denoting the number of processes. More complex (and default in the code) execution mode is provided by the command *pvm_run_debug*. Running this command opens a terminal window for each spawned process and starts a debugger (`gdb` in the default setting). The debugger begins the execution of a specified executable using a file in the working directory. The command has the following usage:

```
usage:  pvm_run -np X program_name debugger_command_file
```

with X denoting as before the number of processes.

Chapter 3

User's guide

3.1 Installing the code

PHP_FEM code is provided in gzip compressed tar file *PHP_FEM_xx.yyyy.tgz*, where *xx* stands for the version number and *yyyy* is the year of the release. When uncompressed and unpacked, by e.g.

```
$ gunzip PHP_FEM_xx_yyyy.tgz
$ tar xvf PHP_FEM_xx_yyyy.tar
```

(or a single '`tar xvzf PHP_FEM_xx_yyyy.tar`' command), it creates at first its own directory *PHP_FEM_xx_yyyy*, then three subdirectories *src*, *examples*, and *doc* and finally the *README* file.

The *README* file provides information similar to included in the introduction and in the next section of the present document: a brief description of the code structure and instructions on how to compile and execute the code. The *doc* directory contains this document and its source files in the *tex* subdirectory. In subdirectories of the *examples* directory reside input files for test examples described in Chapter 3 of this document.

REMARK. For Linux systems with no lapack and blas libraries installed there is a separate, larger distribution file *PHP_FEM_xx_yyyy_linux.tar.gz* containing an additional directory, *lib*, with blas and lapack libraries).

3.2 Making the code

To build the code the current directory has to be changed to the *src* subdirectory of the *PHP_FEM_xx_yyyy* directory:

```
$ cd PHP_FEM_xx_yyyy/src
```

All files necessary for making the code are in this directory. The source files for the code reside in subdirectories beginning with three letters indicating the module of the code. The *include* subdirectory contains interface header files with declarations of parameters and external procedures. The remaining files from *src* are used for compiling and linking the code.

The files for making the code are designed to enable one to work on several systems simultaneously. Different systems are distinguished by different architecture names. To

build the code for a particular architecture the value of two environmental variables should be set:

- `PHP_FEM_DIR` - the home directory of the code (e.g. *some_path/PHP_FEM_xx_yyyy*)
- `PHP_FEM_ARCH` - a unique symbol specifying the particular architecture (possible examples: `PC_LINUX`, `IBM_RS6K` or `SGI_R10K`)

To specify the necessary values of environmental variables the following lines can be included into the shell initialization file (the example is for the csh and tcsh shells):

```
setenv  PHP_FEM_DIR   /my_home_directory/PHP_FEM/PHP_FEM_01_2005
setenv  PHP_FEM_ARCH  PC_LINUX
```

For running the code with the PVM communication library in a debug mode the environmental variable `PHP_FEM_DIR` should be exported using `PVM_EXPORT` mechanism (e.g. for *.cshrc* file: `setenv PVM_EXPORT PHP_FEM_DIR`).

Before compiling the individual procedures of the code two directories have to be created:

- *bin* - for storing executable files for all architectures
- *obj/$(PHP_FEM_ARCH)* - for storing obj files for a given architecture.

The directories can be created explicitly or by running '`make config`' command (this command should be run every time the code is built for a new architecture). The command confirms the value of `PHP_FEM_ARCH` variable, creates directory *obj/$(PHP_FEM_ARCH)* for object files (and directories *bin* and *obj* if necessary), and gives information on possible options for making particular variations of the code. Then, while making the code, the object files for a given architecture are stored in the proper subdirectory of *obj* and separate binaries are created for each architecture (e.g. if $(PHP_FEM_ARCH) is set to LINUX then the subdirectory for object files is *obj/LINUX* and all binaries' names end with `LINUX`.

The supplied *Makefile* contains all dependencies and commands for creating executables. The code for a particular architecture is compiled and linked using included architecture dependent options. These options are stored in *make.$(PHP_FEM_ARCH)* files. An example of such a file for `LINUX` architecture is provided as *make.LINUX*. The following values are specified in architecture dependent files:

`CC` - C compiler

`LD` - linker for C and Fortran (blas and lapack) procedures

`INC` - placement of header files (local, i.e. `-I$(PHP_FEM_DIR)/include` and for standard libraries if necessary)

`LIB` - placement and names of libraries (see example *make.xxx* files)

`CFL` - C optimization and other flags

`LDFL` - linker optimization and other flags

To make the code a particular option to the `make` command should be specified. This option implies the selection of particular modules that comprise the created code. The name of the built executable file reflects the choices. For example running:
`$ make conv_diff_dg_prism_bliter_pvm`
will create the executable for the current architecture ($(PHP_FEM_ARCH) architecture) named *PHP_FEM_cdpb_pvm_'$(PHP_FEM_ARCH)* that uses the following modules:

- c - convection-diffusion-reaction problem dependent module
- d - discontinuous Galerkin approximation module
- p - mesh manipulation module with prismatic elements
- b - block based multigrid preconditioned GMRES iterative solver for linear equations
- pvm - domain decomposition parallel execution module with PVM communication library

The output of the '`make config`' command gives other possible options for '`make`', corresponding to different possible combinations of modules.

Running '`make clean`' will remove, for the current architecture, the binary executables as well as the object files.

3.3 Running the code

The code can be executed from any directory using binaries from *bin*. Each executable accepts the optional argument denoting the working directory for the code. If none is specified the current directory from which the command to run is issued is used. When running the code from the subdirectories of the *examples* directory the call is
`$ ../../bin/name_of_the_executable current_directory`
or simply
`$ name_of_the_executable current_directory`
if *$(PHP_FEM_DIR)/bin* directory has been included into the path.

To run any version of the code two files have to be specified in the working directory. These are:

- mesh initialization file
- field initialization file

3.3.1 Mesh data input files

There are two types of mesh files accepted: files that store all data from the code data structure (input files *mesh.dat* and mesh files in the format from "gradmesh" mesh generator[1], input files *mesh_jk.dat*. The first format is used also for dump files (*mesh.dmp*)

[1] jkucwaj@zms.pk.edu.pl

created by the program itself. The user selects the type of input file at the beginning of program's execution.

For both kinds of mesh file formats the first line in the file contains four parameters specifying dimensions for allocating arrays of structures corresponding to: nodes, edges, faces and elements, respectively. These sizes constitute limits for indexes that will not be passed during the execution of the program. There is however the way to increase the maximal size of the problem, by dumping out the contents of the data structure, modifying properly the *mesh.dmp* file and running the code again.

An example 3D mesh file of the first kind, containing all data from the data structure and used for creating the mesh composed of two elements depicted in Fig. 3.1, looks as follows:

```
300000 1300000 1600000 600000
8 9
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
0.0 1.0 1.0
14 15
1 1 2
1 2 3
1 3 4
1 4 1
1 3 1
1 5 6
1 6 7
1 7 8
1 8 5
1 7 5
1 2 6
1 3 7
1 4 8
1 1 5
9 10
3 3 1 0
-5 -2 -1
3 3 2 0
-4 -3 5
3 3 1 0
6 7 10
3 3 2 0
-10 8 9
```

```
4 2 1 0
1 11 -6 -14
4 2 1 0
2 12 -7 -11
4 2 2 0
3 13 -8 -12
4 2 2 0
4 14 -9 -13
4 -1 1 2
5 14 -10 -12
2 3
5 0 2 0
1 3 5 6 9
5 0 2 0
2 4 -9 7 8
```

The meaning of subsequent parts of the file is:

- `300000 1300000 1600000 600000`

  - size parameters for allocating structures

- `8 9`

  - the number of node structures, the index for the next created node,

- ```
  0.0 0.0 0.0
  1.0 0.0 0.0
  1.0 1.0 0.0
  0.0 1.0 0.0
  0.0 0.0 1.0
  1.0 0.0 1.0
  1.0 1.0 1.0
  0.0 1.0 1.0
  ```

  - coordinates for subsequent nodes

- `14 15`

  - the number of edge structures, the index for the next created edge,

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

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

- for each edge structure: type, two end nodes' numbers

- `9 10`

  - the number of face structures, the index for the next created face,

- ```
  3 11 1 0
  -5 -2 -1
  3 11 2 0
  -4 -3 5
  3 11 1 0
  6 7 10
  3 11 2 0
  -10 8 9
  4 1 1 0
  1 11 -6 -14
  4 1 1 0
  2 12 -7 -11
  4 1 2 0
  3 13 -8 -12
  4 1 2 0
  4 14 -9 -13
  4 -1 1 2
  5 14 -10 -12
  ```

  - for each face structure: type of face (3 - triangle, 4 - quadrilateral), kind of boundary condition (1 - Dirichlet, 11 - Neumann, < 0 - inter-element boundary, the value denotes the shift in nodes positions for two neighboring elements), numbers of two equal size neighbors (-1 - large neighbor, 0 - boundary of the domain), in the second line: numbers of three or four edges constituting sides of the face ("+" sign indicates the same, "-" sign the opposite orientation)

- `2 3`

  - the number of element structures, the index for the next created element,

- ```
  5 0 0 0
  1 3 5 6 9
  5 0 0 0
  2 4 -9 7 8
  ```

  - for each element structure: type (the number of vertexes), material number, father's number, refinement type; in the second line: global numbers of faces forming sides of the element (sign indicates the same or opposite orientation)

The file above describes an initial (not adapted) mesh, the meaning of subsequent fields for adapted meshes may be different (see the section describing the data structure).

Input files for the generator of 3D layered meshes (modified output files produced by the "gradmesh" mesh generator) have different structure, as in the simple example file below:

```
300000 1300000 1600000 600000
4 0.0 1.0 100 1 2
0.  0.        1.  0.      1. 1.        0. 1.
2
 1 1 2 3
 -11 -11  2
 1 1 3 4
 1 -11 -11
```

The meaning of input data is as follows:

- `300000 1300000 1600000 600000`

  - size parameters for allocating structures

- `4 0.0 1.0 100 1 2`

  - the number of nodes in a 2D mesh, z coordinate of the bottom plane of the domain, z coordinate of the top plane of the domain, the number of layers of elements, boundary flag for bottom layer, boundary flag for top layer

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

  - x and y coordinates of subsequent nodes

- `2`

  - the number of 2D elements

- ```
  1 1 2 3
  -11 -11  2
  1 1 3 4
  1 -11 -11
  ```

  - for each element: material number or element number if material data is not specified (sign indicates the type of 2D element, > 0 - triangle, < 0 - quadrilateral), node numbers (three or four numbers), in the second line: for each edge, either neighboring element number (> 0) or boundary condition flag (< 0)

3.3.2 Field data input files

The field initialization file specifies the parameters for finite element approximation. Both original files (*field.dat* and dump files *field.dmp* have the same form.

For the examples included in the current release of the code the discontinuous Galerkin approximation is employed. For this approximation the format of field input files is the following. In the first line the four parameters are specified:
  ```

- the number of equations (1 for scalar problems, ¿1 for vector problems)
- the number of copies of data structures for degrees of freedom necessary to perform calculations (for time dependent and nonlinear problems more than one copy of degrees of freedom may be necessary, for e.g. values at the previous time step or at the previous iteration).
- the kind of finite element basis functions (3D complete polynomials or tensor products of 1D and 2D complete polynomials)
- the encoded default degree of approximation for coarse spaces in multigrid linear solvers

Then two options are possible: either the file contains one negative number indicating the encoded degree of approximation for all elements or for each element there exist one line with the encoded (positive) degree of approximation. The field input file can also contain the values of degrees of freedom for all elements. This option is usually utilized for dump files, while original initialization is done using either the zero value or the field initialization function.

There are several example input files provided in subdirectories of the *examples* directory. The contents of these files is described in Chapter 3.

3.3.3 Control data input files

For convection-diffusion-reaction problems (and possibly all other more complicated problems) the execution of the program is governed by a set of control parameters. These parameters are stored in the file *problem.dat*. The convection-diffusion-reaction problem dependent module has a special procedure that reads the control data file. The file has a rigid structure with keywords and parameters in specified places. Each keyword is followed by either another keyword or an exact number of parameters (floating point parameters are indicated by the floating point or scientific notation). The meaning of parameters will be explained using an example *control.dat* file for convection-diffusion-reaction problems solved by the discontinuous Galerkin approximation:

```
INTERACTIVE
1            1 - yes, 0 - no
NUM_OF_PROB
1            1 - single problem
PROBLE_NAME
101          101 - pure convection in x direction
MESHFIL_TYP
2            2 - 2D unstructured mesh data
CONTROL_PAR
INIT
1            1 - initial data applied at t=0 and supplied for linear solver
NRMA
0            0 - no material data specified (coefficients given in the code)
```

```
SLOP
0             0/1 - no/yes default slope limiter
IMPL
0.5           0.5 - Crank-Nicholson time integration scheme
PENA
0             0 - no penalty terms => NDG formulation
FREE
0.7           0.7 - strong slope limiting
BCON
1             1 - single set of BC data
1             1 - number for the first set (<10 - Dirichlet data)
1.0             value
ICON
1             1 - single set of initial data
0.0           0.0 - initialize solution to zero in all elements
TIME_PARAMS
1 0           1 - default time integration scheme, 0 - output level
STEP
0 20          1 - current step number, 20 - final step number
TIME
0 600         0 - current time, 600 - final time
DTIM
5.0 5.0       5 - current time step length, 5 - previous time step length
CONV
0 0.0         0 - convergence type, 0 - convergence treshold
INTV
0 0           0 - no automatic dumpouts, 0 - no automatic graphics output
NONL_PARAMS
0 0           0 - no nonlinear solver, 0 - output level: no output
CONV
0 0 0         0 - max_iter, 0 - convergence type, 0 - convergence treshold
LINS_PARAMS
1 2           1 - single level GMRES solver, 2 - output level
CONV
100 1 1.e-6   100 - max_iter, 1 - convergence type, 1.e-6 - treshold
ADAPT_PARAM
1 2           1 - default strategy for adaptation, 2 - output level
CTRL
1 2 1         1 - interval between adapting, 2 - max_gen, 1 - max_gen_diff
TOLE
0.001 0.01    0.001 - global tolerance level, 0.01 - ratio for de-refinements
```

The first parameter (after keyword `INTERACTIVE`) specifies the mode of operation for the program. The value is substituted to the local variable `interactive` in the `main` procedure. `interactive` equal to 1 denotes the interactive mode, for which a user chooses interactively actions to be performed by the program. In the other mode (`interactive`

equal to 0) the program realizes a sequence of operations encoded in `main` and quits. In the basic setting the sequence comprises of solving the problem and computing the error of approximation (the last only when the exact solution is known).

The next parameter (after keyword `NUM_OF_PROB` specifies the number of problems solved simultaneously. Then the loop over problems begins and, from that moment, all program operations are performed referencing to a particular problem or to the interface between problems (discretizations). Each problem has its own number.

The rest of the example *control.dat* file corresponds to a single discretization. If there would be more problems solved simultaneously, then this part should be repeated with suitable values of parameters for other problems.

The value after keyword `PROBLE_NAME` denotes the current problem identifier. This identifier is used e.g for choosing the appropriate coefficients of the solved partial differential equations, types and values for boundary conditions and the exact solution. There is a convention employed to use numbers less then 100 for static problems and greater then 100 for time dependent problems.

After keyword `MESHFIL_TYP` there is an identifier of the type of mesh file: 1 denotes mesh files with all data for a 3D mesh specified, 2 corresponds to files containing data only for 2D unstructured meshes, used by the internal mesh generator to produce 3D meshes composed of layers of elements.

There is also an option to specify `MESHFIL_TYP` ≤ 0. This indicates that the current problem should share its mesh with the problem having the number `-MESHFIL_TYP` (problems are numbered consecutively, starting from 0).

Several parameters read next (after `CONTROL_PAR` keyword) control the execution of the problem dependent part of the code. After keyword `INIT` there is an indicator whether the problem uses initial condition or not. More precisely there are three possible values:

0 - boundary value problem, do not use initial condition

1 - initial and/or nonlinear boundary value problem, apply initial condition at the beginning of execution and get initial guess from data structure for each solution of linear equations

-1 - special option: boundary value problem pure Neumann boundary conditions

The number after `NRMA` indicates how to input element material numbers. If it is 0 then no material data is specified and the code should supply all necessary coefficients for the partial differential equations. When the number is greater than zero than it indicates the number of materials. The code will look for the file *material.dat* in the calling directory with the specified number of data records (the form of data records is problem dependent). Each element should have its material number specified in the input mesh file. When the number after `NRMA` is equal to -1 than it indicates that there is one material, for all elements, with the data specified in *material.dat* file. Material number fields for all elements, supplied in the mesh file, are overwritten by the number 1.

After the keyword `SLOP` there is an off/on (0/1) switching parameter indicating whether slope limiting will be used or not. The value after `IMPL` specifies the implicitness parameter for the time integration routine. For stationary problems 1.0 should be specified, although the code will substitute 1.0 whenever time integration is switched off.

The next parameter, after PENA, switches on the NIPG formulation of the discontinuous Galerkin method whenever it is not equal to zero. In such a case its value denotes the penalty coefficient in the NIPG formulation.

After the keyword FREE there is a parameter that can be used by different problem dependent modules. Currently it is utilized by the slope limiting procedure to control the strength of slope limiting.

The group of parameters following the keyword BCON specifies the boundary data. First of them is the number of two-line records of data for each boundary condition case. In each record the first line sets the numerical symbol of the particular boundary condition. This number can be retrieved for each face by calling the routine *mmr_fa_bc* (note that this is different from the boundary condition type returned by *pdr_get_bc_type* which specifies only whether the face is internal, Dirichlet, Neumann or Robin). The boundary condition number is send as one of input parameters for the procedure *pdr_fa_coeff* returning the values for specific boundary conditions. The number of values for each boundary condition is problem dependent. In the example file, corresponding to a scalar problem, one Dirichlet boundary condition with the number 1 and the value 1.0 is specified (the convention adopted in the code is to assign to Dirichlet conditions numbers from 1 to 9, to Neumann conditions numbers from 11 to 19 and to Robin conditions numbers from 21 to 29.

In a similar way to the boundary data, the initial data for the problem are supplied. After the keyword ICON there is the number of initial condition's one-line records. Each line contains NREQ values that are subsequently stored. To retrieve these values from the i-th array the function *pdr_ic_val* should be used with i send as one of the arguments.

The next several groups of parameters are designed to control the execution of different (sub-)modules of the program. Some of these parameters correspond to modules or options not yet implemented, but expected in fully developed applications. For all modules there is one parameter that has the same meaning, namely the parameter specifying the output level. It can have four values:

PDC_SILENT=0 - no information is printed to output

PDC_ERRORS=1 - only error information printed

PDC_INFO=2 - important (e.g. convergence) information printed

PDC_ALLINFO=3 - all available information printed

The first group of parameters is used for controlling different aspects of time integration algorithm:

- keyword TIME_PARAMS, parameters in order:
  - type (identifier) of time integration scheme (0 - no time integration, 1 - default scheme)
  - output level
- keyword STEP, parameters in order:
  - current time step number

  - final time step number (to stop execution)

- keyword `TIME`, parameters in order:
  - current time instant
  - final time (to stop execution)

- keyword `DTIM`, parameters in order:
  - the length of the current time step
  - the length of the previous time step

- keyword `CONV`, parameters in order:
  - type (identifier) of a convergence in time criterion
  - threshold value for assessing convergence in time

- keyword `INTV`, parameters in order:
  - interval (in time steps) between dumpouts of data
  - interval (in time steps) between graphics output

The next group (after `NONL_PARAMS`) corresponds to, not implemented in the basic setting, nonlinear equations solver module. There are the following parameters (in order, separated by the keyword `CONV`):

- type (identifier) of a nonlinear equations solver
- output level
- maximal number of nonlinear iterations
- type (identifier) of a nonlinear convergence criterion
- threshold value for assessing nonlinear convergence

Similar parameters are specified as basic control data for linear equations solver execution. It is assumed that most often iterative solvers will be used and the data reflects it. The parameters are (again separated by the keyword `CONV`):

- identifier of a solver (in the basic setting: 0 - direct dense solver, 1 - single level GMRES, 2 - multi-level GMRES)
- output level
- maximal number of iterations
- type (identifier) of a linear convergence criterion
- parameter for assessing linear convergence

The options in the control input file may be overwritten by the options read from the solver input file *solver.dat*.

The last group of parameters corresponds to mesh adaptation. They are used both by the problem dependent adaptation module and the problem independent mesh modification module. The parameters and the keywords are the following (in order):

- `ADAPT_PARAM`
  - type (identifier) of an adaptation strategy
  - output level
- `CTRL`
  - the number of time steps between adaptations
  - maximal generation number allowed for elements (initial mesh elements have generation 0)
  - maximal generation difference allowed between two neighboring elements
- `TOLE`
  - tolerance level used by the adaptation strategy
  - the ratio controlling thresholds for de-refinements

3.3.4 Solver data input files

The file *solver.dat* presents a solution for the situation when a certain module uses too many parameters to be convenient to include in the general control file *control.dat*.

The GMRES method with multi-level preconditioning requires several control parameters for every level. An example file shows parameters for some particular solver, together with the explanation of all possible options:

```
FINE_LEVEL    (the only level for single-level solvers)
0 20          0-standard GMRES_type, 20-Krylov vectors)
2 1 1         2-block_GS preconditioner, 1-internal sweep, 1-small blocks

COARSE_LEVEL  (the coarsest level solver)
1 0 20        1-GMRES solver, 0-type: standard, 20-Krylov vectors
2 1 1         2-block_GS preconditioner, 1-internal sweep, 1-small blocks
100 1 1e-12   100-max_iter, 1-rel_ini conv type, 1e-12 conv treshold
-1 0          -1-p_coarse=p_fine, 0-output level: no output

INTER_LEVELS  (smoothers/preconditioners for intermediate levels)
1 1           1-pre-smooth sweep, 1-post-smooth sweep
2 1 1         2-block_GS preconditioner, 1-internal sweep, 1-small blocks
-1 0          -1-p_coarse=p_fine, 0-output level: no output
```

```
**************************************************************************

Options for GMRES solver (on fine level a,g,h,i,k provided also by fem code):
a b c: a-solver: 1-single level GMRES, 2-multi-level/multigrid GMRES
       b-type: 0-standard, 1-matrix free (not implemented yet)
       c-number of Krylov vectors
d e f: d-preconditioner (only at the current level):
         0-no, 1-block Jacobi, 2-block Gauss-Seidel (both as smoothers)
         3-additive Schwarz (block Jacobi as preconditioner)
         4-ILU(0) (as smoother), 5-ILU(0) (as preconditioner)
       e-number of internal sweeps for block Jacobi and Gauss-Seidel
       f-type of blocks (most useful options):
         1-one-element blocks
         2-larger blocks with no overlap (for ILU(0))
         4-larger blocks with overlap (for block smoothers)
g h i: g-maximal number of single iterations (not restarts!)
       h-convergence criterion for the norm of residual:
         1-rel_ini - its ratio to the norm of initial residual
         2-abs_val - its value
         3-rel_rhs - its ratio to the norm of the rhs vector
       i-convergence treshold
j k:   j-indicator of the degree of approximation (on the coarsest level)
         (-1 - the same as for the finest level)
       k-output level
         0-no information is printed to output
         1-only error information printed
         2-important (convergence) information printed
         3-all available information printed

Options for standard iterations solver (fine level a,g,h,i,k as above):
a b c: a-solver: 10-standard iterations, 20-multigrid
       b-number of pre-smooth steps (for multigrid on the finest level)
       c-number of post-smooth steps (for multigrid on the finest level)
d e f: d-actual solver/smoother (only at the current level):
         0-no, 1-block Jacobi, 2-block Gauss-Seidel (both as smoothers)
         3-additive Schwarz (block Jacobi as preconditioner)
         4-ILU(0) (as smoother), 5-ILU(0) (as preconditioner)
       e-number of internal sweeps for block Jacobi and Gauss-Seidel
       f-type of blocks (the same options as for the GMRES solver):
g h i: g-maximal number of single iterations
       h-trash (max norm of update used as convergence criterion)
       i-convergence treshold
j k:   j-indicator of the degree of approximation (on the coarsest level)
         (-1 - the same as for the finest level)
       k-output level (the same options as for the GMRES solver)
```

```
Options for intermediate level solvers/smoothers:
b c:   b-number of pre-smooth steps
       c-number of post-smooth steps
d e f: d-smoother at the current level:
         0-no, 1-block Jacobi, 2-block Gauss-Seidel (both as smoothers)
         3-additive Schwarz (block Jacobi as preconditioner)
         4-ILU(0) (as smoother), 5-ILU(0) (as preconditioner)
       e-number of internal sweeps for block Jacobi and Gauss-Seidel
       f-type of blocks (the same options as for the GMRES solver):
j k:   j-indicator of the degree of approximation
         (-1 - the same as for the finest level)
       k-output level (the same options as for the GMRES solver)
```

3.4 Examples

There are three subdirectories of the *examples* directory containing input files for three example problems and solution strategies. Running the examples consist in calling the suitable executables (different for Laplace's equation and for convection equation) and selecting the proper options from the menus. For each problem the proper mesh input files are indicated. Field input files and solver input files can be modified to chose the desired solution strategies.

3.4.1 Laplace's equation approximated on a structured mesh – directory *data_std*

This example provides the basic setting for testing sequential capabilities of the code. It can be used to check h and p convergence rates for the discontinuous Galerkin formulations for a pure diffusion problem with smooth solution [4]. Also, different sets of shape functions can be tested: tensor product polynomials (TENSOR basis) and complete order polynomials (COMPLETE basis).

The setting for the solved problem and the solution strategy is the following:

- Equation:
$$\Delta u = \Delta u_{ex}$$
$$u_{ex} = \exp\left(-x^2 - y^2 - z^2\right)$$
- Domain:
$$[0,1] \times [0,1] \times [0,1]$$
- Boundary conditions:
$z = 0$ and $z = 1$ – Neumann BC: $\frac{\partial u}{\partial n} = \frac{\partial u_{ex}}{\partial n}$
$x = 0$, $x = 1$, $y = 0$ and $y = 1$, – Dirichlet BC: $u = u_{ex}$

- Meshes: initial mesh (Fig. 3.1) read from the mesh input file *mesh.dat* . Subsequent meshes obtained by consecutive isotropic refinements of two initial elements. The second mesh with 16 elements, 56 faces, 66 edges and 27 nodes shown in Fig. 3.2.

- Degree of approximation p: uniform, specified in the field input file.

Fig. 3.4 shows the solution obtained with the mesh from Fig. 3.2 and degree of approximation $p = 3$.

3.4.2 Laplace's equation approximated on an unstructured mesh – directory *data_box*

This example provides the basic setting for testing parallel capabilities of the code's version utilizing the PVM library. To make running of the codes easier two files are provided:

- *go_lapl* - script for starting *pvm_run_debug* command with the suitable arguments

- *gdb_file* - debugger input file

Sequential runs can be started as usual using executable names. For parallel runs the use of scripts consist of simply calling:

```
$ go_lapl
```

The rest consist of choosing the proper options from the menus. The mesh input file is the, so called in the code, "gradmesh" input file *mesh_jk.dat*. The first step of execution should consist in partitioning the mesh. Otherwise different processors run the same code with the same not-decomposed data structure.

3.4.3 Time dependent convection equation approximated on an unstructured mesh – directory *data_conv_x*

This test case shows the possibility of performing parallel dynamic adaptivity and load balancing. The mesh file with dumped data should be chosen. Again, the first step of execution should consist in partitioning the mesh (otherwise different processors run the same code with the same not-decomposed data structure).

Problem description:

- Equation:
$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0$$

- Domain: rectangular box in 3D space

- Initial condition:
$$u = 0$$

- Boundary conditions:
$x = 0$, $x = 1$, $y = 0$, $y = 1$, $z = 0$ and $z = 1$ – Dirichlet BC: $u = u_{exact}$

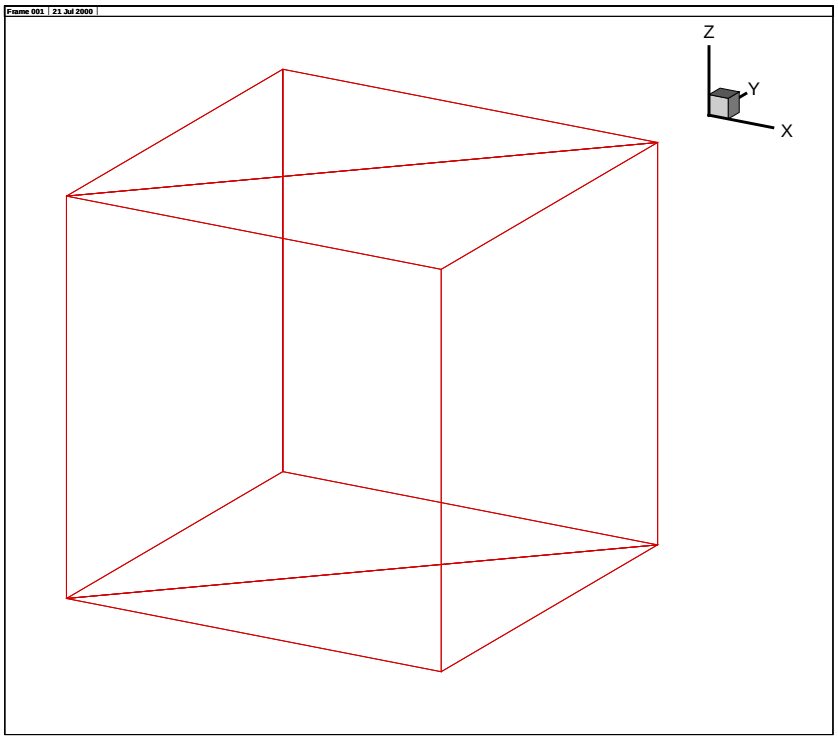

Figure 3.1: Test example 1 - initial mesh with 2 prismatic elements

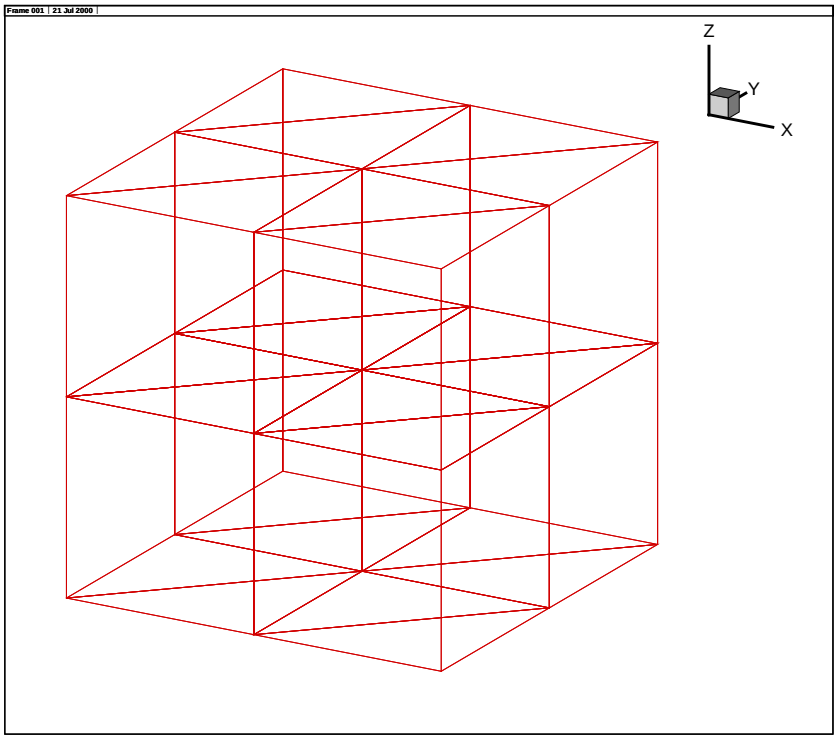

Figure 3.2: Test example 1 - once refined mesh with 16 prismatic elements

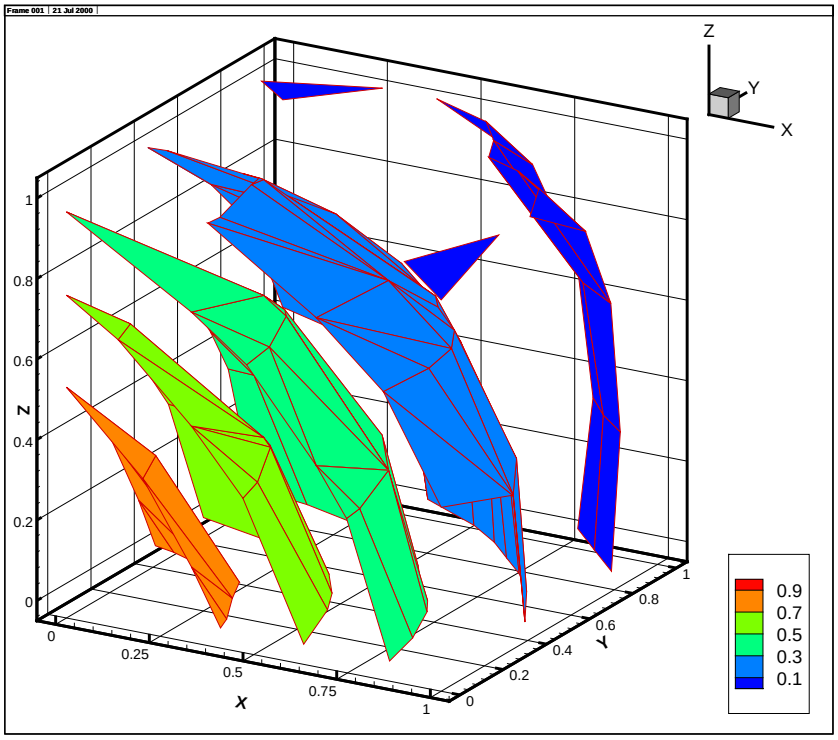

Figure 3.3: Test example 1 - solution for initial mesh and degree of polynomial $p = 2$

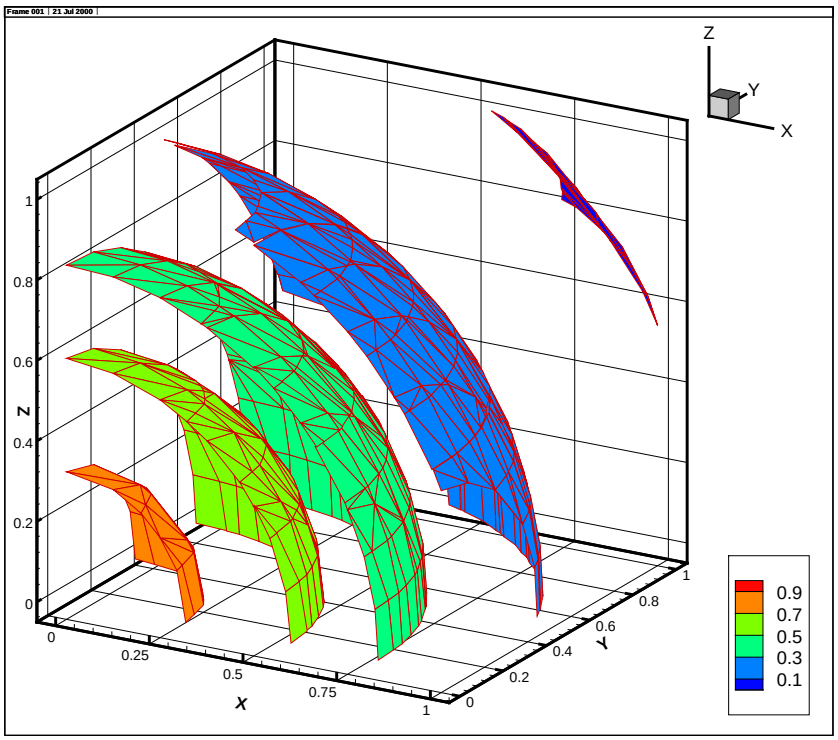

Figure 3.4: Test example 1 - solution for once refined mesh and degree of polynomial $p = 3$

- The exact solution is a square wave traveling along the x direction with the speed 1.
- Meshes : obtained by adaptive refinements and de-refinements of an initial unstructured mesh.
- Adaptive strategy: to divide elements with the H^1 norm of the true error exceeding the limiting value until the generation level max_gen is reached, to cluster back families of elements with the sum of H^1 norms less then a specified ratio of the limiting value.
- Degree of approximation: $p = 1$.
- Time integration: second order Crank-Nicholson (trapezoidal rule) scheme, constant time step length.

Bibliography

[1] K. Banaś, 'A modular design for parallel adaptive finite element computational kernels', in *Computational Science — ICCS 2004, 4th International Conference, Kraków, Poland, June 2004, Proceedings, Part II*, eds., M. Bubak, G.D. van Albada, P.M.A. Sloot, and J.J. Dongarra, volume 3037 of *Lecture Notes in Computer Science*, pp. 155–162. Springer, (2004).

[2] K. Banaś, 'Parallelization of large scale adaptive finite element computations', in *Parallel Processing and Applied Mathematics, Proceedings of Vth International Conference, PPAM 2003, Częstochowa, Poland, 2003*, eds., R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wa'sniewski, volume 3019 of *Lecture Notes in Computer Science*, pp. 431–438. Springer, (2004).

[3] K. Banaś, 'A model for parallel adaptive finite element software', in *Domain Decomposition Methods in Science and Engineering*, eds., R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O.Widlund, and J. Xu, volume 40 of *Lecture Notes in Computational Science and Engineering*, pp. 159–166. Springer, (2004).

[4] K. Banaś, *Zastosowanie adaptacyjnej metody elementów skończonych do obliczeń wielkiej skali*, Wydawnictwo Politechniki Krakowskiej, Kraków, 2004.

[5] K. Banaś, 'On a modular architecture for finite element systems. I. Sequential codes', *Computing and Visualization in Science*, **8**, 35–47, (2005).

[6] L. Demkowicz, J.T. Oden, W. Rachowicz, and O. Hardy, 'Towards a universal h-p adaptive finite element strategy, Part.1 Constrained approximation and data structure', *Computer Methods in Applied Mechanics and Engineering*, **77**, 79–112, (1989).

[7] W. Rachowicz, 'An anisotropic *h*-type mesh refinement strategy', *Computer Methods in Applied Mechanics and Engineering*, **109**, 169–181, (1993).

[8] B. Cockburn and C.W. Shu, 'The local discontinuous Galerkin finite element method for convection diffusion systems', *SIAM Journal on Numerical Analysis*, **35**, 2440–2463, (1998).