Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Sven Beuchler, Arnd Meyer, Matthias Pester

SPC-PM3AdH v1.0 - Programmer's Manual

Preprint SFB393/01-08

Acknowledgment. The package *SPC-PM* has been developed in the *Sonderforschungsbereich 393* of the *Technische Universität Chemnitz* under the supervision of A. Meyer and Th. Apel. Other main contributors are G. Globisch, G. Kunert, D. Lohse, F. Milde, M. Pester, U. Reichel, and M. Theß.

Preprint-Reihe des Chemnitzer SFB 393

ISSN 1619-7178 (Print) **ISSN 1619-7186** (Internet)

SFB393/01-08 **March 2001, revised August 2003**

Author's addresses:

Sven Beuchler
TU Chemnitz
Fakultät für Mathematik
D-09107 Chemnitz
Germany

email: sven.beuchler@mathematik.tu-chemnitz.de

Arnd Meyer
TU Chemnitz
Fakultät für Mathematik
D-09107 Chemnitz
Germany

email: arnd.meyer@mathematik.tu-chemnitz.de

Matthias Pester
TU Chemnitz
Fakultät für Mathematik
D-09107 Chemnitz
Germany

email: mspester@mathematik.tu-chemnitz.de

http://www.tu-chemnitz.de/sfb393/

Contents

1 Overview

1.1 Introduction

SPC-PM are computer programs to solve the Poisson equation or the Lamé system of linear elasticity over a three-dimensional domain on a MIMD parallel computer.
The historical roots of the program are in several parallel programs for solving problems over two dimensional domains using domain decomposition techniques. These codes have been developed since about 1988 by A. Meyer, M. Pester, and other collaborators. On the other hand, Th. Apel, F. Milde, and U. Reichel [1] developed 1999/2000 an adaptive and parallel program, *SPC-PM3AdT*, for the solution of the Poisson equation or the Lamé system using tetrahedral meshes. Furthermore, A. Meyer developed an adaptive, parallel program *SPC-PM2AD* in two dimensions using triangles and quadrilaterals. The version 1.0 of *SPC-PM3AdH* documented here is a sequential, adaptive program using hexahedral meshes. A parallel version and a version for higher polynomial degrees (p-version) are planned. For an introduction of the capabilities of the program, its installation and utilization we refer to the User's Manual of version 3.3 of *SPC-PM Po 3D* [3]. The aim of this new Programmer's Manual for version 1.0 is to provide a description of the new data structures and to introduce new routines. It is written for those who are interested in a deeper insight into the code, for example for improving and extending.
The documentation is organized as follows: In the next section we describe the boundary value problems that can be solved and the finite elements that are used. Chapter 2 is concerned with the changed data structure. Chapter 3 shows the order of events of an adaptive program: adaptive mesh refinement, assembly of the stiffness matrix, solving of the linear system of equations and error estimation, which are described in the following chapters. In a last chapter, we give some numerical examples.
In this documentation we use *slanted style* for real existing paths and filenames, *emphasis style* for program parameters, sans serif style to characterize buttons and menu items of programs with a graphical interface, and `typewriter style` for the names of variables.

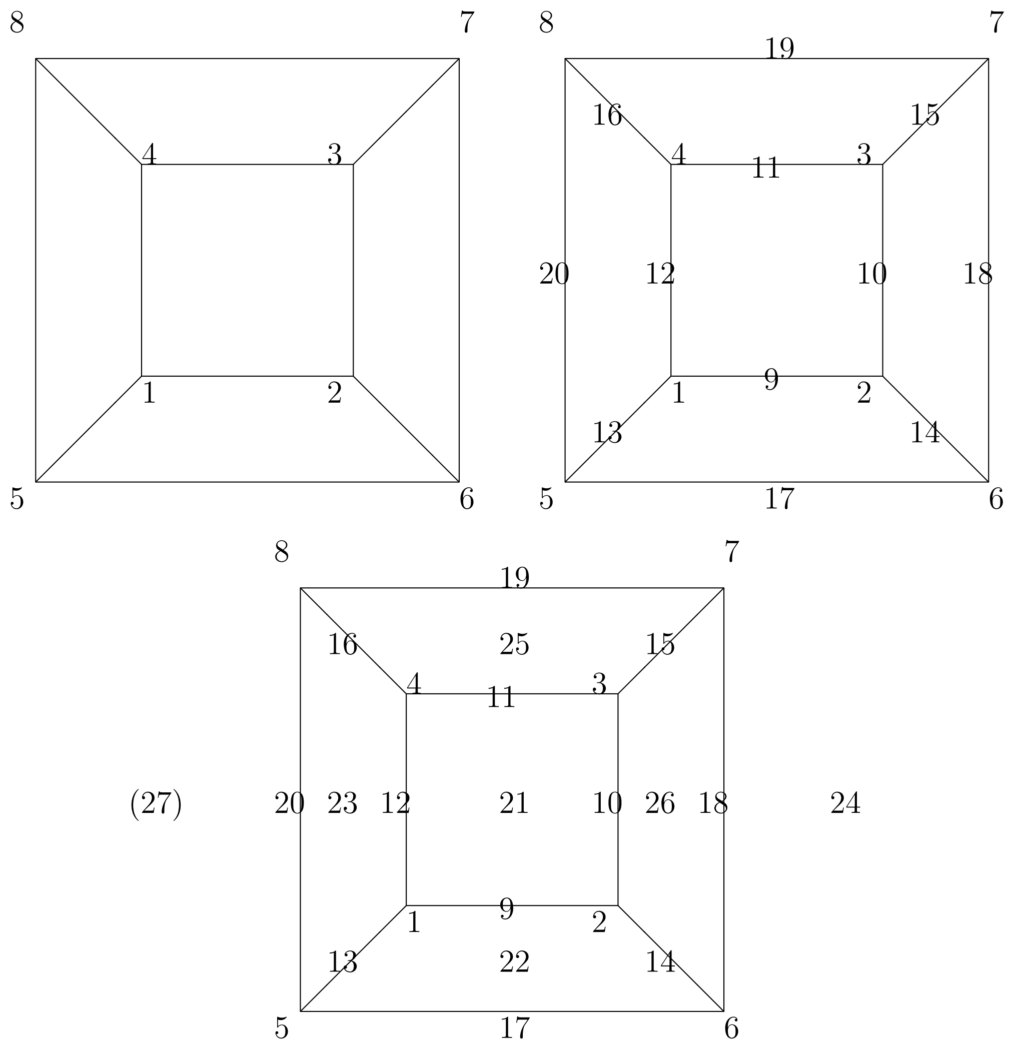


Figure 1: Finite elements implemented in the program *SPC-PM Po 3D* version 4.x.

1.2 The boundary value problems

We consider the 2nd order elliptic differential equation in the notation

$$\begin{aligned} Lu := -\nabla\cdot(A(x)\nabla u) + cu &= f \text{ in } \Omega\subset\mathbb{R}^3,\\ u &= u_0 \text{ on } \partial\Omega_1,\\ n^tA(x)\nabla u &= g \text{ on } \partial\Omega_2,\\ \frac{\partial u}{\partial n} &= 0 \text{ on } \partial\Omega\backslash\partial\Omega_1\backslash\partial\Omega_2, \end{aligned} \tag{1.1}$$

where $A(x) = \operatorname{diag}(a_i)_{i=1}^3$ or the Lamé problem for $\underline{u} = (u^{(1)}, u^{(2)}, u^{(3)})^t$

$$\begin{aligned} -\mu\triangle\underline{u} - (\lambda+\mu)\text{grad div}\underline{u} &= \underline{f} \text{ in } \Omega\subset\mathbb{R}^3,\\ u^{(i)} &= u_0^{(i)} \text{ on } \partial\Omega_1^{(i)}, i=1,2,3,\\ t^{(i)} &= g^{(i)} \text{ on } \partial\Omega_2^{(i)}, i=1,2,3,\\ t^{(i)} &= 0 \text{ on } \partial\Omega\backslash\partial\Omega_1^{(i)}\backslash\partial\Omega_2^{(i)}, i=1,2,3 \end{aligned} \tag{1.2}$$

where $\underline{t} = (t^{(1)}, t^{(2)}, t^{(3)})^t = S[u]\cdot\underline{n}$ is the normal stress. The stress tensor $S[u] = (s_{ij})_{i,j=1}^3$ is defined with $\underline{x} = (x^{(1)}, x^{(2)}, x^{(3)})^t$ by

$$s_{ij} = \mu\left[\frac{\partial u^{(i)}}{\partial x^{(j)}} + \frac{\partial u^{(j)}}{\partial x^{(i)}}\right] + \delta_{ij}\lambda\nabla\cdot\underline{u},$$

where $\underline{n}$ is the outward normal, and δ_{ij} is the Kronecker delta. The domain Ω must be bounded. In the present version curved boundaries are treated only by the refinement procedure. The boundary value problem is solved by a standard finite element method, using either hexahedral elements with linear, tri-quadratic or serendipity shape functions, see Figure 1. The discretization of (1.1,1.2) with a basis $(\phi_1^q, \ldots, \phi_n^q) \in V_q$ yields to the linear system

$$K^q\underline{u} = \underline{f}. \tag{1.3}$$

We obtain

$$u^q = \sum_{i=1}^n u_i^q\phi_i^q \tag{1.4}$$

as numerical solution of (1.1) or (1.2). The parameter q denotes the level of refinement.

2 Data structure

2.1 General remarks

In FORTRAN77 it is impossible to allocate memory during the run of the program, but there are several large arrays in our FEM program which are used only for a certain time. So it is necessary to have a dynamic memory management. To solve this problem in FORTRAN77 we have a very large workspace vector (as large as possible) in our program to use parts of it as arrays in the subroutines. There are several pointer variables which determine the array index on which data starts. We develop our own memory management and must take care of calculating these pointers to avoid overlaps. For an easier handling the *SPC-PM* package provides now a large set of functions and routines for the memory management.

Because of the adaptive mesh refinement we use now only the full data structure (FDS) with its greater variability. The reduced data structure does not longer exist.

There are a few general variables:

`NFG`	number of degrees per freedom per node,
`NEN2D`	number of nodes per face,
`NEIN`	number of unknowns per solid per d.o.f.,
`NE2IN`	number of unknowns per face per d.o.f..

We describe the arrays in the following general form:

1. name and dimension of the array,

2. general description of the array,

3. structure of a data block of the array,

4. additional information.

For some arrays there are pointers within the data blocks which determine the position of data. Most of the dimensions of the arrays are also variables/parameters which are located in COMMON blocks in the source file *adapmesh.inc*. It is better to use these variables because of possible evolution of the data structure.

2.2 Full data structure

2.2.1 Volumes

are stored in the array

1. `SOLID(MSOL,*):  MSOL=MELFAC+MELNOD+MELMAT=9+27+MELMAT`.
   So, each solid is a 'structure' of `MSOL` informations.

2. Each volume is described by its 6 faces.

3. (a) | face_1 | ... | face_6 | material | mid-node | polynom.degree |
   (b) | node_1 | ... | node_27 |
   (c) | element stiffness matrix | element right hand side |

4. The value 'material' is a pointer in the `MATERIAL` array. Opposite faces are face_1 and face_4, face_2 and face_5, face_3 and face_6. The ordering of the nodes is compatible with the ordering of the faces as given in Figure 2.

2.2.2 Faces

1. `FACE(MFACE,*):  MFACE=12`

2. Each face is described by its 4 edges.

3. | edge_1 | ... | edge_4 | son_1 | ... | son_4 | type | mark | mid-node | polynomial degree |

4. The value 'type' contains 4 informations, the geometry, the Dirichlet or Neumann data and the information about a boundary face, see 2.3.1:

1.Byte	pointer to geometry,
2.Byte	pointer to Neumann data,
3.Byte	pointer to Dirichlet data,
4.Byte	marking for boundary faces.

   Each face can be subdivided into 4 faces. Thus, we have possibly 4 sons. For son_1 we have the cases

> 0	face is subdivided, number of the first son,
< 0	face is hanging,
0	otherwise.

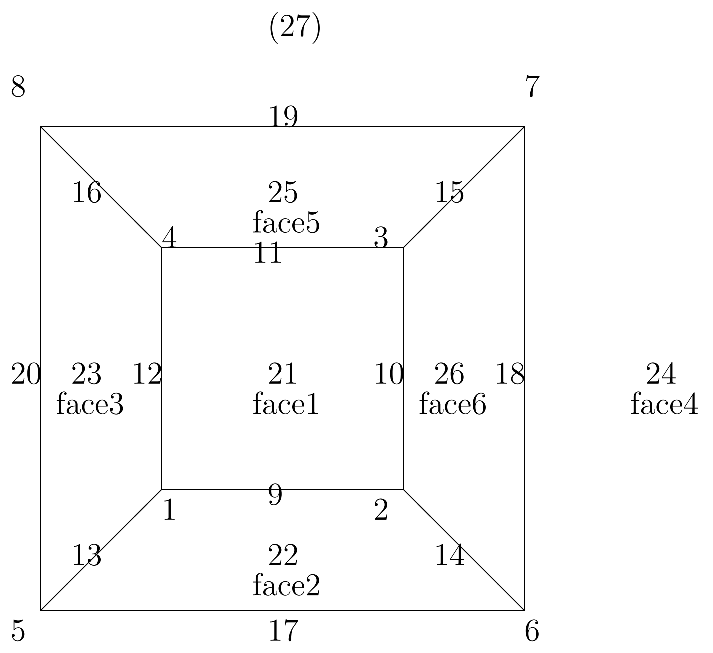


Figure 2: Ordering of the nodes in a given solid.

The value 'mark' is used for the error estimator:

1	face is marked,
0	face is unmarked.

Opposite edges are edge_1 and edge_3, and edge_2 and edge_4, respectively.

2.2.3 Edges

1. `EDGE(MEDGE,*):  MEDGE=6`

2. Each edge is described by its 2 vertices and the middle node.

3. | middle node | vertex_1 | vertex_2 | son1 | son2 | polynomial degree |

4. The 'values' son1 and son2 are used for this edge to find its sons being subdivided. We have the following cases for son1 and son2:

son1> 0	edge is subdivided, numbers of the two sons,
else:	edge is not subdivided, son2: geometry information,
son1< 0	edge is hanging,
son1$= 0$	otherwise,

2.2.4 Coordinates of the nodes

1. `X(NDIM+NDOF+1,*):NDIM=3`
2. Each node is represented by its three Euclidian coordinates.
3. | X_i | Y_i | Z_i | solution_1 | ... | solution_ndof | index of `U` |
4. The last entry stores the number j of shape function $\phi^q_{|j|}$ of the nodal basis associated to this node.

$j > 0$	node is vertex of one element,
$j = 0$	no shape function ϕ_j in the nodal basis associated with this node,
$j < 0$	else.

2.2.5 Dirichlet/Neumann data

1. `DIR(1+MDIRNEU*NDOF,*):  MDIRNEU=5`
   `NEUM(1+MDIRNEU*NDOF,*):  MDIRNEU=5`
2. The Dirichlet/Neumann data are associated with input faces of the coarse mesh readed from the input file. Both have the same data structure.
3. | urface | type, data for d.o.f. 1 | type, data for d.o.f. 2 | ... |
4. We store only the boundary condition on the coarse mesh. We have the following possibilities for data.

type	meaning	data
0	no boundary condition for this d.o.f.	unused
1	constant boundary condition	c, rest unused
2	linear boundary condition	a_1, a_2, a_3, b
100	special boundary condition	unused

In the case of a boundary condition of the type 1: $g(x) = c$, the constant value c is stored. In the case of a boundary condition of the type 2: $g(x) = a^t x + b$ the values $a \in \mathbb{R}^3$ and $b \in \mathbb{R}$ are given (in this order).

2.2.6 Hierarchical list

1. `HIER(NHIER-1)`

2. The hierarchical list connects all nodes with its father nodes.

3. Information per son:
   | son | index for BPX | $-i$ | father 1 | index for BPX | index for BPX | … | father i | index for BPX | index for BPX |

4. The information of sons are stored one behind the other in the hierarchical list. The integer i stores the number of fathers. We have 2, 4 or 8 fathers per son. In the case of the BPX preconditioner we have additional information. For more information, see 6.3. The integer `NHIER` describes the length of the hierarchical list. The information 'index for BPX' is unused for the Yserentant- and Jacobi- preconditioner.

2.2.7 Material information

1. `RMATE(NMAXIN+2,*):  NMAXIN=5`

2. The material information contains the local behaviour of (1.1) or (1.2).

3. For (1.1)
   | number | inf | a_1 | a_2 | a_3 | c | f |
   and for (1.2)
   | number | inf | $E = \frac{\mu}{\lambda+\mu}(3\lambda + 2\mu)$ | $\nu = \frac{\lambda}{2(\lambda+\mu)}$ | f_1 | f_2 | f_3 |

4. The value 'number' is the material number associated with input volumes of the coarse mesh read from the input file. The value 'inf' is the number of valid information of the `NMAXIN` possible material information.

2.2.8 Geometry data

1. `GEOMETRY(DIMGEOM,*):  DIMGEOM= 9`

2. Geometry data is taken from the `#FACEGEO` section in the mesh file and provides necessary parameters for all faces types.

3. | kind_of_face | data_1 | … | data_8 |

4. At the moment the following values for 'kind_of_face' are possible:

1	plane face, defined by a normal vector and a point on the plane,
2	plane face, defined by a point on the plane and a normal vector,
11	cylinder surface,
21	sphere surface,
31	cone surface,
41	ellipsoid, hyperboloid surface,
51	torus surface.

   For more details see [7]. The required parameters for the geometric correction are stored in data_1 to data_8.

2.2.9 `U`

1. `U(*)`

2. The array `U` stores the solution. It is required to compute a good start vector for the CG after adaptive mesh refinement.

3. | solution for node |

4. The index j of (1.4) does not correspond with the node number, see 2.2.4.

2.3 INCLUDE-Files/COMMON-Blocks

There is a number of COMMON-Blocks in our program. Most of them are located in INCLUDE-Files.

2.3.1 *adapmesh.inc*

This INCLUDE-File contains a number of variables/parameters which determines dimension of data.

- polynomial degrees and nodes per element:

PSO	polynomial degree per solid	currently 1 or PFA,
PFA	polynomial degree per face	currently 1 or PED,
PED	polynomial degree per edge	currently 1 or 2,
NODES_IN_EL	nodes per element	currently 8, 20 or 27.

- parameter for array X:

MNODE	number of coordinates	currently 3.

- parameter for array EDGE:

MEDGE	information per edge	currently 6.

- parameters for array FACE:

MFACE	information per face	currently 12,
FRBCODE	position of the boundary code	currently 9.

- parameters for array SOLID:

MELFAC	information for material, faces, mid-node and polynomial degree	currently 9,
MELNOD	information nodes	currently 27,
MELMAT	element matrix and right hand side	currently 810, 3780 or 6804,
MSOL	information per solid	currently MELFAC+ MELNOD+ MELMAT.

- parameter for arrays DIR and NEUM:

MDIRNEU	information for boundary condition	currently 5.

- parameter for array GEOMETRY:

DIMGEOM	information for geometry	currently 9.

- parameter for boundary code FRBCODE:

`GEOBYTE`	byte for geometry	currently 255,
`NEUMBYTE`	byte for Neumann b.c.	currently 255*256,
`DIRBYTE`	byte for Dirichlet b.c.	currently 255*256*256,
`NEUMDIV`	factor for getting Neumann b.c.	currently 256,
`DIRDIV`	factor for getting Dirichlet b.c.	currently 256*256,
`RANDBIT`	factor for boundary code	currently 256*256*256.

2.3.2 *bpx.inc*

This file contains variables which are necessary for solving the linear system using the BPX-preconditioner [4]. All variables in this COMMON-block:

- `/BPX/`

`NBPX`	number of unknowns for all levels,
`NLEV`	number of levels,
`ZONE`	number of unknowns per level.

2.3.3 *GRAF3D.inc*

This file is a copy from further versions.

2.3.4 *filename.inc*

The file is necessary for reading the mesh files. All variables are in the COMMON-block `FILENAME`. For more information, see [6].

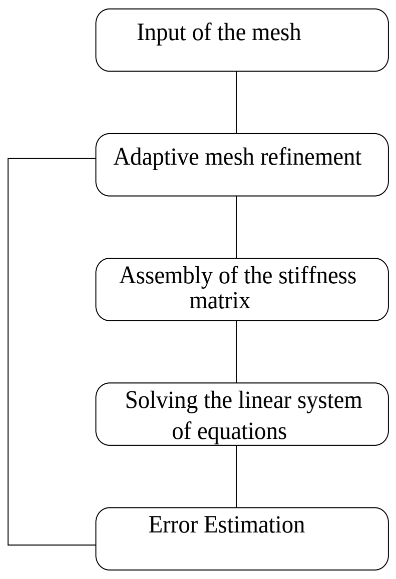

Input of the mesh
Adaptive mesh refinement
Assembly of the stiffness matrix
Solving the linear system of equations
Error Estimation

Figure 3: Overview over an adaptive program

3 Order of events of an adaptive program

In this chapter, we describe the main points of an adaptive program. A short description of the main program *test.f* is given in Figure 3. The two parts, mesh input and refinement of the mesh, we describe in chapter 4. For the mesh input and initialization of the data structure, we call the subroutines *set_pointers*, *grobnetzread*, *grobnetzkorr*, *correctnode* and *hieranf*. For the management of the adaptive refinement, we have the subroutines *korredg*, *netfine*, *correctnode* and *hangnod*. The assembly of the stiffness matrix is written in chapter 5 and done during the call of the subroutine *assem*. A detailed discussion of the subroutines handling with the solving of the linear system we find in chapter 6, for which we call the subroutines *chghier*, *stave*, and *ppcgm*. The error estimator is described in chapter 7 and called by the subroutines *efromu* and *errorest*.

4 Mesh refinement

4.1 General mesh handling

Unlike former versions of *SPC-PM Po 3D*, program *SPC-PM3AdH* constructs the mesh in several steps driven by the solution until the estimated local error of each volume is below a certain bound. Therefore the mesh generation consists also of several steps.

- Read in the user mesh data and generate the user mesh,
- generate new nodes on each mid-edge, mid-face and mid-volume,
- adaptive mesh refinement.

4.2 Mesh input

4.2.1 The procedure

The user mesh is read from a standardized file, compare [3]. These files are located in the subdirectory *./mesh4* (hexahedral meshes). We read the mesh in two steps. Firstly we read the numbers of nodes, edges, faces and volumes and estimate the size for each of the arrays `X, EDGE, FACE, SOLID, HIER` and for several vectors. We determine the starting pointers for each of these structures on the workspace vector `A`. Then we read the mesh.

4.2.2 Parameters of *set_pointers*

The subroutine *set_pointers* determines the start-pointers for all arrays.

```
SUBROUTINE SET_POINTERS (JX,JEDGE,JFACE,JSOLID,JMAT,JGEO,JDIR,
                      JNEUM,JHIER,JLCC,JCC,JH,NEIN,A,MAX,KMAX)
```

`A`	I/O	workspace vector
`JX`	O	start-pointer for the array `X`
`JEDGE`	O	start-pointer for the array `EDGE`
`JFACE`	O	start-pointer for the array `FACE`
`JSOLID`	O	start-pointer for the array `SOLID`
`JMAT`	O	start-pointer for the array `RMATE`
`JGEO`	O	start-pointer for the array `GEOMETRY`

`JDIR`	O	start-pointer for the array `DIR`
`JNEUM`	O	start-pointer for the array `NEUM`
`JHIER`	O	start-pointer for the array `HIER`
`JLCC`	O	start-pointer for the array `LCC`
`JCC`	O	start-pointer for the array `CC`
`JH`	O	start-pointer for the end
`NEIN`	O	number of unknowns per element
`KMAX`	O	maximal number of possible elements
`MAX`		maximal workspace in words

4.2.3 Parameters of *grobnetzread*

The subroutine *grobnetzread* generates the initial mesh. It reads the user mesh from a file. Then the routine shifts the arrays `X, EDGE, FACE and SOLID` to its positions estimated by *set_pointers*.

```
SUBROUTINE GROBNETZREAD(NN,X,NF,FACE,NE,EDGE,NVOL,SOLID,NDIR,
                          NNEU,NMAT,NMATINF,NGEO,GEO,A,LAENGE)
```

`NN`	O	number of nodes
`X`	I/O	array of node coordinates `X`
`NF`	O	number of faces
`FACE`	I/O	array of faces `FACE`
`NE`	O	number of edges
`EDGE`	I/O	array of edges `EDGE`
`NVOL`	O	number of volumes
`SOLID`	I/O	array of solids `SOLID`
`NDIR`	O	number of Dirichlet faces
`NNEUM`	O	number of Neumann faces
`NMAT`	O	number of materials
`NMATINF`	O	number of material informations
`NGEO`	O	number of geometries
`GEO`		array of geometries `GEOMETRY`
`A`	I/O	workspace vector
`LAENGE`	I	length of `A`

4.2.4 Parameters of *grobnetzkorr*

The subroutine corrects the initial mesh. It changes in the arrays `FACE` and `SOLID` the ordering of the edges and faces such that opposite edges and faces on opposite positions, see 2.2.2 and 2.2.1. Furthermore, it generates nodes on the middle of edges, faces and volumes. It initializes the polynomial degree for each edge, face and volume. Each face gets informations about the kind of boundary condition. Each edge obtains from its faces the kind of geometry and each node is projected to the existing geometry.

```
SUBROUTINE GROBNETZKORR(NN,X,NF,FACE,NE,EDGE,NVOL,SOLID,NDIR,
                        DIR,NNEUM,NEUM,NMAT,NMATINF,RMAT,GEO)
```

`NN`	I/O	number of nodes
`X`	I/O	array of node coordinates `X`
`NF`	I	number of faces
`FACE`	I/O	array of faces `FACE`
`NE`	I	number of edges
`EDGE`	I/O	array of edges `EDGE`
`NVOL`	I	number of volumes
`SOLID`	I/O	array of solids `SOLID`
`NDIR`	I	number of Dirichlet faces
`DIR`	I	the Dirichlet data `DIR`
`NNEUM`	I	number of Neumann faces
`NEUM`	I	the Dirichlet data `NEUM`
`NMAT`	I	number of materials
`NMATINF`	I	number of material informations
`RMAT`	I	the materials `RMAT`
`GEO`		array of geometries `GEOMETRY`

4.3 Adaptive mesh refinement

4.3.1 General remarks

Each volume is refined by subdividing into 8 sub-volumes, see Figure 4. We have the possibility of hanging nodes. Because of the refinement of solids into 8 sub-solids, we have only some cases of hanging nodes. We can have possible hanging nodes on edges and faces. We accept only hanging nodes in

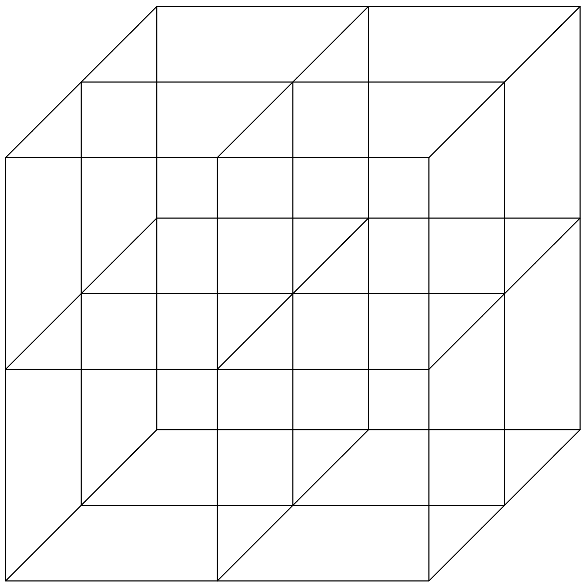


Figure 4: Refinement of a solid by subdividing into 8 sub-solids.

the finest level. For all cases of edge and face hanging nodes see Figure 5.

4.3.2 Cases for subdividing a solid

A solid is subdivided if one of the following cases is satisfied:

- There are non acceptable hanging nodes on an edge of the solid.
- One face or this solid is marked for refinement by the error estimator.
- The number of subdivided edges is greater than `K3-1`.
- The number of subdivided faces is greater than `K4-1`.

The first condition is equivalent to the following. It exists an edge which has already been two times more refined in one solid than in the neighbor.

4.3.3 Parameters of *netfine*

The subroutine *netfine* controls the adaptive mesh refinement. It checks for each solid the conditions of refinement and calls the subroutine *div8solid* for

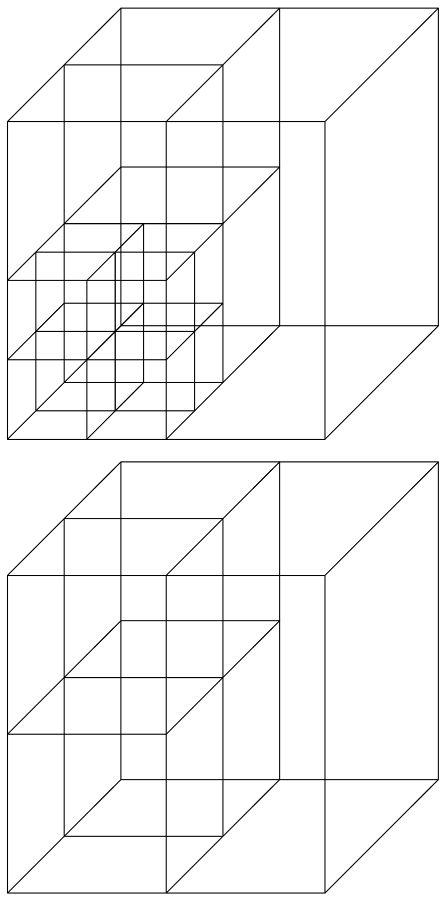


Figure 5: Cases of acceptable hanging nodes (below) and not acceptable hanging nodes(above).

subdividing the solid.

```
SUBROUTINE NETFINE(NN,X,NE,EDGE,NF,FACE,NVOL,SOLID,NHIER,HIER)
```

`NN`	I/O	number of nodes
`X`	I/O	array of node coordinates `X`
`NF`	I/O	number of faces
`FACE`	I/O	array of faces `FACE`
`NE`	I/O	number of edges
`EDGE`	I/O	array of edges `EDGE`
`NVOL`	I/O	number of volumes
`SOLID`	I/O	array of solids `SOLID`
`NHIER`	I/O	length of hierarchical list
`HIER`	I/O	array of the hierarchical list `HIER`

4.3.4 Parameters of *teiltest*

The INTEGER function *teiltest* checks for each solid the conditions of refinement.

0	solid may not be refined,
> 0	solid has to be refined.

```
INTEGER FUNCTION TEILTEST(THIS,NN,X,NE,EDGE,NF,FACE)
```

`THIS`	I	solid have to be checked
`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`

4.3.5 Parameters of *div8solid*

The subroutine *div8solid* exercises the refinement of a given solid into 8 new solids. The subroutine calls several subroutines, see 4.3.6.

```
SUBROUTINE DIV8SOLID(THIS,SOLID,NVOL,FACE,NF,EDGE,NE
                     ,X,NN,NHIER,HIER,GEOMETRY)
```

`THIS`	I	solid has to be refined
`NN`	I/O	number of nodes
`X`	I/O	array of node coordinates `X`
`NF`	I/O	number of faces
`FACE`	I/O	array of faces `FACE`
`NE`	I/O	number of edges
`EDGE`	I/O	array of edges `EDGE`
`NVOL`	I/O	number of volumes
`SOLID`	I/O	array of solids `SOLID`
`NHIER`	I/O	length of hierarchical list
`HIER`	I/O	array of the hierarchical list `HIER`
`GEOMETRY`	I	array of geometries `GEOMETRY`

4.3.6 Short description of further subroutines

- `SUBROUTINE EDORTOFA(KAN,DIR,NF,FACE,SOLID)`

  The subroutine determines the 4 edges of a given solid `SOLID` which do not bound a given pair of opposite faces. The parameter `DIR` has the values 1,2 or 3, for the first and fourth, the second and fifth, third and sixth face. The 4 edges are stored in the array `KAN`. The subroutine calls further subroutines, the subroutine *neigh* and the LOGICAL functions *bereits* and *edgoffac*.

  1. `SUBROUTINE NEIGH(DIR,JS,NGHBOUR)`
     The subroutines determines the local number $(1, \ldots, 6)$ for that 2 faces which neighbors of `JS` and which are not `DIR` and `DIR+3`.
  2. `LOGICAL FUNCTION EDGOFFAC(FACE,KA)`
     The function determines if the INTEGER `KA` is an element of the array `FACE`.
  3. `LOGICAL FUNCTION BEREITS(KA,ANZ,EDG)`
     The function is a generalization of `EDGOFFAC` in the case of an array with `ANZ` elements.

- `SUBROUTINE DIV4FACE(THFA,NE,EDGE,NF,FACE,NN,X,KA,FA,NHIER,`
  `HIER,NNEW,GEOMETRY)`

  The subroutine subdivides a given face `THFA` into 4 sub-faces. If the face has already been subdivided the subroutine does no division. In

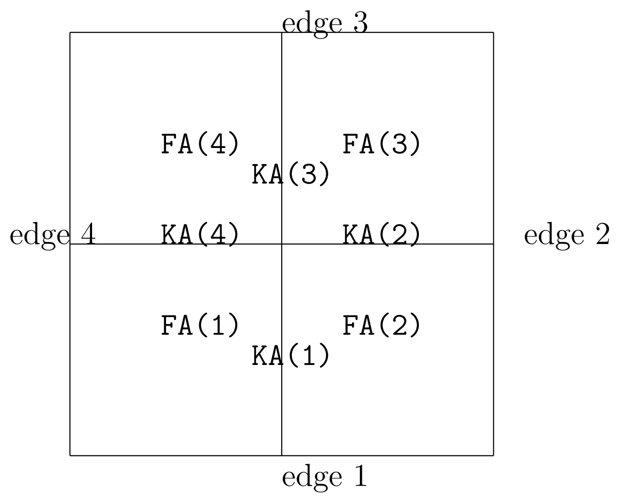


Figure 6: Old and new edges of an subdivided face.

both cases, the subroutine determines the 4 son faces `FA`, the mid-node `NNEW` and the 4 edges `KA` with `NNEW` as vertex. The ordering of the edges and faces in `KA` and `FA` is determined by the ordering of the edges in the array `THFA`, see Figure 6. Note, that this ordering is implicitly used for projecting the hanging nodes, see 6.4. The subroutine calls the procedures *divedg, cutedg, doedge, do12face, do9thnode, cutface* and *hierarch*.

-
  ```
  SUBROUTINE DIVEDG(THEDG,K1,K2,NOM,X,NN,
                                EDGE,NEDG,HIER,NHIER)
  ```

  The subroutine subdivides a given edge `THEDG` into 2 sub-edges. If the edge has already been subdivided the subroutine does no division. In both cases, the subroutine determines the 2 son edges `K1` and `K2` and the mid-node `NOM`. The subroutine calls the procedures *do2midnodes, do_edg* and *hierarch*.

- Subroutines located in *cutedg.f*:
  These subroutines are determine the cut of two sets.

  - `SUBROUTINE CUTEDG(NC,EDG1,EDG2,EDGE)`
    The subroutine determines the node, which are common vertex of the 2 edges. If the cut is empty then `NC=0`.
  - `SUBROUTINE CUTFACE(FA1,FA2,KA)`
    The subroutine determines the cut-edge between the two closed faces. If the cut is empty then `KA=0`.

- Subroutines located in *do27thnode.f*:
  These subroutines generate several nodes.

  - `SUBROUTINE DO27NODE` for the mid-node of a solid by the 20 nodes Serendipity mapping,
  - `SUBROUTINE DO2627NODE` for the mid-node of a solid by the 26 nodes Serendipity mapping,
  - `SUBROUTINE DO9ThNODE` for the mid-node of a face,
  - `SUBROUTINE DOMIDNODE` for the mid-node of an edge,
  - `SUBROUTINE DO2MIDNODES` for the mid-nodes of the two son faces.

  The procedure *donode* gives the mid-node of an edge. In the case of a non existing node, the subroutine calls *do9thnode*.

- Subroutines located in *doedg.f*:
  This file stores procedures for generating an edge.

  - `SUBROUTINE DO_EDG` generates an edge with 6 informations.
  - `SUBROUTINE DOEDGE` determines the edge with a given mid-node and vertex. If the edge does not exist, the subroutine calls the procedure *do_edg* generating the edge.

- Subroutines located in *dosolid.f*:
  This file contains the procedures

  - `SUBROUTINE DO_SOLID` for generating a solid with 9 informations,
  - `SUBROUTINE DO12FACE` for generating a face with 12 informations.

- The subroutine *hierarch* collects the information for the preconditioner and is described in 6.3.

- Subroutines located in *gemnode.f*:

  - `SUBROUTINE GEMNODE(NF,FACE,NE,EDGE,FA1,FA2,`
    `FA3,I1,I2,I3)`

    We have a triple of 4 faces `FA1,FA2,FA3` which are the sons of subdivided faces. The father faces have a common node `I`. The indices `I1,I2,I3`$\in$ $\{1,\ldots,4\}$ are the numbers of that sons of `FA1,FA2,FA3` which have the common node `I`. The subroutine calls *cutface*. Notice, that this routine does not work in the case of general arrays of faces.

  - `SUBROUTINE GEMEDG(NE,EDGE,KA1,KA2,I1,I2)`

    We have two faces with a common edge, which have already subdivided. For the division we introduced for both faces 4 edges `KA1` and `KA2`, see Figure 6. We want to have the indices `I1` and `I2` of the two edges with a nonempty cut. The subroutine calls *cutedg*. The subroutine calls *edortofa*.

- Subroutines located in *getnodel.f*:

  - `SUBROUTINE GETNODEFA(THFA,EDGE,NE,FEMEL)` gives the 9 nodes `FEMEL` of the face `THFA`. The subroutine calls *cutedg*.

  - `SUBROUTINE GETNODE(THIS,EDGE,NE,NF,FACE,NN,X,NODES)`
    gives the 27 nodes `NODES` of the solid `THIS`. The ordering corresponds to Figure 2. The subroutine calls *cutface*, *change*, *cutedg* and

  - `SUBROUTINE GETMIDNODES(THFA,VNODES,MNODES,NE,EDGE)`.
    This subroutine determines for a given face `THFA` and a given ordering of the 4 vertex nodes `VNODES` the 4 nodes `MNODES` lying on the middle of the edges.

  - `SUBROUTINE GETNODEU(NE,THEL,ND,NK,X,NFG,U,XU)`
    This subroutine gives the coordinates and `NFG` solutions of the nodes of `THEL` in DOUBLE PRECISION.

4.4 Tree structures

Tree substructures of subroutines marked with the symbol $*$ are described before in the list.

4.4.1 SETPOINTER

```
SETPOINTER
```

4.4.2 GROBNETZREAD

```
GROBNETZREAD
```

4.4.3 GROBNETZKORR

```
GROBNETZKORR
    ↪ DO9THNODE
    ↪ CHAEDGE
        ↪ CHANGE
        ↪ CUTEDG
    ↪ EDGEGEO
    ↪ CHASOLID
        ↪ CHANGE
        ↪ CUTFACE
    ↪ DO2627NODE
    ↪ CORDIRNEU
    ↪ DOFACEBOUNDARY
    ↪ PCORECT
        ↪ PRO1FACE
        ↪ PRO2FACE
```

4.4.4 NETFINE

```
NETFINE
    ↪ TEILTEST
        ↪ EDORTOFA
            ↪ NEIGHB
            ↪ BEREITS
            ↪ EDGOFFAC
    ↪ DIV8SOLID
    ↪ DIV4FACE
        ↪ PCORECT∗
        ↪ DOEDGE
            ↪ DO_EDG
        ↪ DIVEDG
            ↪ DO_EDG
            ↪ HIERARCH
            ↪ DO2MIDNODES
                ↪ DOMIDNODE
        ↪ CUTEDG
        ↪ DO12FACE
        ↪ DO9THNODE
        ↪ HIERARCH
        ↪ CHANGE
        ↪ CUTFACE
    ↪ DOEDGE∗
    ↪ DIVEDG∗
    ↪ CUTEDG
    ↪ GEMEDG
        ↪ CUTEDG
    ↪ DO12FACE
    ↪ DO9THNODE
    ↪ HIERARCH
    ↪ GEMNODE
        ↪ CUTFACE
    ↪ DO_SOLID
    ↪ GETNODE
        ↪ CHANGE
        ↪ CUTFACE
        ↪ CUTEDG
        ↪ GETMIDNODES
    ↪ DO27NODE
    ↪ DO2627NODE
    ↪ PCORECT∗
```

5 Assembly of the equation system

5.1 Changes against version v2.x of *SPC-PM Po 3D*

5.1.1 General remarks

The assembly of the stiffness matrix has significantly changed since version v2.x, [2]. The same routines are used for the numerical integration and for the shape functions. The only change is the saving of the stiffness matrix element-wise. Here, we store the upper triangle. The main diagonal of the stiffness matrix and the right hand side have to be assembled.

5.1.2 Coarse grid matrix

In this version, the coarse grid matrix does not exist.

5.2 Short description of the subroutines

5.2.1 Subroutine in *assem.f*

The subroutine *assem.f* handles the generation of the element stiffness matrices, the assembly of the main diagonal of the global stiffness matrix and the right hand side.

```
SUBROUTINE ASSEM(NK,X,NEL,NEIN,SOLID,NEDG,EDG,NFACE,FACE,NDIR,
                 DIR,NNEUM,NEUMW,NMAX,NMATE,RMATE,K,MAXADR,F,
                 H,IER,N)
```

`NK`	I	number of nodes
`X`	I/O	pointer to array of node coordinates `X`
`NEL`	I	number of volumes
`NEIN`	I	number of unknowns per element
`SOLID`	I/O	array of solids `SOLID`
`NEDG`	I	number of edges
`EDG`	I	array of edges `EDGE`
`NFACE`	I	number of faces
`FACE`	I	array of faces `FACE`
`NDIR`	I	number of Dirichlet b.c
`DIR`	I	array of Dirichlet b.c.`DIR`

`NNEUM`	I	number of Neumann b.c
`NEUMW`	I	array of Neumann b.c.`NEU`
`NMATE`	I	number of material
`NMAX`	I	number of material informations
`RMATE`	I	array of materials `MAT`
`K`	O	main diagonal of stiffness matrix `K`
`MAXADR`		
`F`	O	right hand side `F`
`H`	H	additional vector
`N`	I/O	length of `K`,`F`, dimension of the linear system

This subroutine calls several subroutines, which are described below. Furthermore, this subroutines gives each node the information if this node is a vertex of a solid or not, see 6.3, 2.2.4 and determines the dimension of the linear system.

5.2.2 Subroutines in *element.f*

- `SUBROUTINE ELEMENT(NE,XT,AEL,REL,GAUSS,NGAUSS,A,C,Q,H)`

`NE`	I	number of unknowns per element
`XT`	I	coordinates `XT` of nodes of element `El`
`AEL`	O	element stiffness matrix `AEL`
`REL`	O	element right hand side `REL`
`GAUSS`	I	coordinates and weights of quadrature rule `GAUSS`
`NGAUSS`	I	number of Gaussian points
`A`	I	for (1.1) $(a_1, a_2, a_3)^t \,|_{El}$ and for (1.2) λ, μ
`C`	I	for (1.1) $c \,|_{El}$
`F`	I	for (1.1) $f \,|_{El}$ and for (1.2) $\underline{f}$
`H`	H	additional vector

The subroutine generates the element stiffness matrix using the Gaussian quadrature rule.

- `SUBROUTINE XTOCB(ND,X1,X2,X3,WLM,NB,BB)` builds for (1.2) the matrix

$$BB = \begin{pmatrix} x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & x_3 \\ \frac{x_2}{2} & \frac{x_1}{2} & 0 \\ 0 & \frac{x_3}{2} & \frac{x_2}{2} \\ \frac{x_3}{2} & 0 & \frac{x_1}{2} \end{pmatrix} + Wlm \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} (x_1, x_2, x_3) \, .$$

This routine saves in the case of `NDOF=3` a lot of operations from the following idea:
Usually the matrix

$$A_g = \hat{B}^t C \hat{B}, \; \hat{B} = (B(\nabla\phi_1), \ldots, B(\nabla\phi_{\texttt{NE}}))$$

is required for each Gaussian point (which means 39366 multiplies for `NE=27`)
From the idea

$$A_g = 2\mu(\tilde{C}^{\frac{1}{2}}\hat{B})^t(\tilde{C}^{\frac{1}{2}}\hat{B})$$

this reduced up to 25%, so this routine builds $\tilde{C}^{\frac{1}{2}}\hat{B}$.

5.2.3 Short description of further subroutines

- Subroutines in *e3intg.f* and *e2intg.f*
  The `SUBROUTINE E3INTG` and `SUBROUTINE E2INTG` determine the integration points and weights, in 3D and 2D. The 2D-routine is only used for the error estimator and the Neumann boundary condition, see 7.

- Subroutine in *getmaterial.f*
  The `SUBROUTINE GETMATERIAL` determines the coefficients and right hand side of the boundary value problem on a given element.

- Subroutines in *matmoduls.f*
  The `SUBROUTINE MATTMUL` does the matrix multiplication $A = B^t C$, where B is stored transposed. The `SUBROUTINE MATMULADU6` does the matrix operation $A = A + \alpha B^t C$ where $B \in \mathbb{R}^{NB,6}$ and $C \in \mathbb{R}^{NC,6}$. This procedure calls the special `DOUBLE PRECISION FUNCTION DSCAPR6` for calculating the scalar product of 2 vectors $a, b \in \mathbb{R}^6$.

- Subroutines in *solve33.f*
  The `SUBROUTINE SOLV33T` solves the $3 \times 3-$(matrix-)system $TX = Y$,

where Y is stored transposed using the rule of Cramer. The `SUBROUTINE SOLV33` solves the $3 \times 3-$(matrix-)system $TX = Y$. The `DOUBLE PRECISION FUNCTION DET3` calculates the determinant of a 3×3 matrix T, given by its rows.

- Subroutines in *mastervalls.f*
  The `SUBROUTINE DOMASTERVAL` gives for several elements the values of shape functions and their first and (for hexahedral elements) second derivatives in the point `XYZ`. It calls one routine of *phi2d.f* or *phi3d.f*.

- Subroutines in *phi2d.f* and *phi3d.f*
  For quadrilaterals, we have `SUBROUTINE PHI2BQ` for 9-node quadrilaterals, `SUBROUTINE PHI2L` for 4-node quadrilaterals and `SUBROUTINE PHI2Q` for 8-node quadrilaterals. In the case of hexahedral elements, we have `SUBROUTINE PHI3L` for linear, `SUBROUTINE PHI3Q` for serendipity and `SUBROUTINE PHI3TQ` for tri-quadratic elements. The second derivatives `D2` are stored in the order $(u_{xx}, u_{yy}, u_{zz}, u_{xy}, u_{xz}, u_{yz})$.

- Subroutines in *fakku.f*
  The `SUBROUTINE FAKKU` accumulates the global vectors.

- Subroutines in *smamvekd.f*
  The `SUBROUTINE SMAMDIA` gives the main diagonal of an element stiffness matrix. The `SUBROUTINE SMAMVEKD` is for the matrix vector multiplication.

- Subroutines in *nodeglobal.f*
  These subroutines handle the last entry of the array of coordinates `X`, see 2.2.4. The `SUBROUTINE NODEINIT` gives `NEIN` nodal shape functions its index ϕ_j. The `SUBROUTINE CHGGLOBSIG(NEIN,NODES,X)` changes the signum of the last entry of `X` for the `NEIN` nodes `NODES`. The `SUBROUTINE NODEGLOBAL` gets for a group of nodes the indices of their shape functions ϕ_j.

- Furthermore, we need additional procedures from *getnodel.f*.

5.2.4 Boundary conditions

- Neumann boundary conditions: The routines for building in the Neumann boundary conditions are located in *neumann.f*.

Subroutine *neumann* does the numerical integration

$$r_j = \int_{face} (a^t \cdot x + b)\phi_j \, \mathrm{d}x \tag{5.1}$$

using Gaussian quadrature for a face. This subroutine does the main part for implementing

$$\vec{n}^t A \nabla u = g.$$

```
SUBROUTINE NEUMANN(ND,NODES,X,NN,NGAUSS,GAUSS,A,B,
                   INTEGRAL,H,NE2IN)
```

`ND`	I	dimension of the face, currently 2
`NODES`	I	array `NODES` of the `NE2IN` nodes of the face
`X`	I	array of node coordinates `X`
`NN`	I	number of nodes
`NGAUSS`	I	number of points for Gaussian quadrature
`GAUSS`	I	coordinates and weights of quadrature rule `GAUSS`
`A`	I	a of (5.1)
`B`	I	b of (5.1)
`INTEGRAL`	O	$r = (r_j)_{j=1}^{Ne2in}$ of (5.1)
`H`	H	additional vector
`NE2IN`	I	number of unknowns per face

Further subroutines are `SUBROUTINE DETER22` for calculating the determinant of the matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ and `SUBROUTINE CROSSPRO` for calculating $b = v_1 \times v_2$.

- Dirichlet boundary conditions: The Dirichlet boundary conditions are built when determining the starting vector for the PPCG method. This is done by subroutine *stave* located in the file *stave.f* described in 6.6.

5.3 Tree Structures

Tree substructures of subroutines marked with the symbol $*$ are described before in the list.

```
ASSEM
  ↪ E3INTG
  ↪ GETNODE∗
  ↪ GETMATERIAL
  ↪ GETNODEU∗
  ↪ ELEMENT
      ↪ DOMASTERVAL
          ↪ PHI2L
          ↪ PHI2Q
          ↪ PHI2BQ
          ↪ PHI3L
          ↪ PHI3Q
          ↪ PHI3TQ
      ↪ MATTMUL
      ↪ SOLV33T
          ↪ DET3
      ↪ XTOCB
      ↪ MATTMULADU6
          ↪ DSCAPR6
  ↪ NODEINIT
  ↪ CHGGLOGSIG
  ↪ NODEGLOBAL
  ↪ FAKKU
  ↪ SMAMDIA
  ↪ E2INTG

     ↪ GETNODEFA∗
  ↪ RANDKOOR
  ↪ NEUMANN
      ↪ MATTMUL∗
      ↪ CROSSPRO
          ↪ DETER22
      ↪ GETNODEU∗
      ↪ DOMASTERVAL∗
```

6 Solving the linear system using the Preconditioned Conjugate Gradient method

6.1 The solver

The linear system of equations is solved using the PCG-method with a preconditioner. The start vector is the solution of the preceeding level. The user can choose between a simple diagonal preconditioning, the hierarchical basis preconditioning [8] and the BPX-preconditioner [4]. A coarse grid solver is not implemented. For applying the solver we have several difficulties. The first problem is the existence of hanging nodes. For these non real existing unknowns we have to build a projection from the space of all shape functions to the space without shape functions corresponding to hanging nodes. Furthermore, we need the adjoint projector. The subroutines for applying the projection and extension are described in 6.4.

We wanted to have a data structure which works for linear, tri-quadratic and serendipity elements. An edge oriented data structure is not useful for these elements. We have nodes, midpoints of faces and volumes, which have 4 or 8 sons. For this purpose we introduced the hierarchical list. Subroutines for the management of the hierarchical list can be found in 6.3.

6.2 Control parameters of the solver

The user can choose several parameters of the solver. We give a short overview over these parameters.

Option	Description		
`v`	variant of preconditioning:	=0/1	Jacobi
		=2/3	Yserentant
		=4/5	BPX
		without/with coarse grid solver	
`i`	`iter`, maximal number of iterations		
`e`	`epsilon`, termination criterion for the relative error norm in the CG algorithm		
`z`	control of the amount of screen output, see `ion` in [3], Table 2.		

6.3 Subroutines for the management of the hierarchical list

A completely new tool of *SPC-PM3AdH* is the hierarchical list. A short description was given in 2.2.6. We consider the three cases 2, 4 and 8 fathers for one son. We created a data structure which is similar for a certain number `NFATH` of fathers. The hierarchical list stores all hierarchies of nodes and is a generalization of the array `LISTE` of further versions, see 5.4. in [3]. Due to the varying number of informations (9, 15 or 27) for each son, a structure of the form `HIER(NINF,*)` is avoided. For each son, the node number of the son, the node numbers of the fathers and number of fathers are stored. Additionally, the indices of fathers in the preceeding levels are necessary to store for the BPX-preconditioner. The order of this information is described in 2.2.6.

- For initialization of the hierarchical list, we have

```
SUBROUTINE HIERANF(NE,EDGE,NF,FACE,NVOL,SOLID,NN,X,
                                          NHIER,HIER).
```

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NVOL`	I	number of volumes
`SOLID`	I	array of volumes `SOLID`
`NHIER`	O	length of hierarchical list
`HIER`	O	array of the hierarchical list `HIER`

After the reading of the mesh, a first refinement is done. The mesh contains elements with 8 nodes. This program works with elements having 27 nodes. Hence, we introduce new nodes on each midpoint of edges, faces and volumes, see 4.2.4. These hierarchies are the first entries of the hierarchical list and entered by the subroutines *hieranf*. This subroutine calls *hierarch* which we will now describe.

- `SUBROUTINE HIERARCH(SON,NFATHER,FATHER1,FATHER2,FATHER3,`
  `      FATHER4,FATHER5,FATHER6,FATHER7,FATHER8,NHIER,LISTE)`

  This procedure enters a son and the possibly 8 fathers to `LISTE`. The parameter `NFATHER` contains the number of fathers. The parameter `NHIER` will increased up to `3(NFATHER+1)`.

- For applying the BPX-preconditioner to this adaptive program, a changing of the order of the hierarchical list is necessary:
  For the BPX-preconditioner [4] for serendipity or tri-quadratic elements, a last refinement from the level $q-1$ to the level q consists of the refinement from linear to tri-quadratic or serendipity elements. For an adaptive program, it can be happen that one edge will not be refined for the levels $1, 2, \ldots, l$, $l \leq q-1$. Hence, this node remains in each level a mid-node of this edge. But, in the hierarchical list, the refinement history will entered during the call of the subroutine *hieranf*, which corresponds to a refinement from level 1 to level 2 and not from level $q-1$ to level q!
  The subroutine *chghier* is implemented for the reordering of the hierarchical list. Figure 7 shows ordering of the hierarchical list corresponding to the nodes.

  `SUBROUTINE CHGHIER(NHIER,HIER,NN,X,NC,NCOARSE,H)`

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NC`	I/O	number of vertices of the mesh
`NCOARSE`	I	number of vertices in the coarse mesh
`H`	H	array of volumes `SOLID`
`NHIER`	I	length of hierarchical list
`HIER`	I/O	array of the hierarchical list `HIER`

  This subroutine moves all nodes not being a vertex of a solid to the end of the hierarchical list. From the subroutine *assem*, it is known which node is a vertex and which not. This information is saved in the last entry of the array of node coordinates of the array `X`, see 2.2.4.

These subroutines are located in *hierarch.f*.

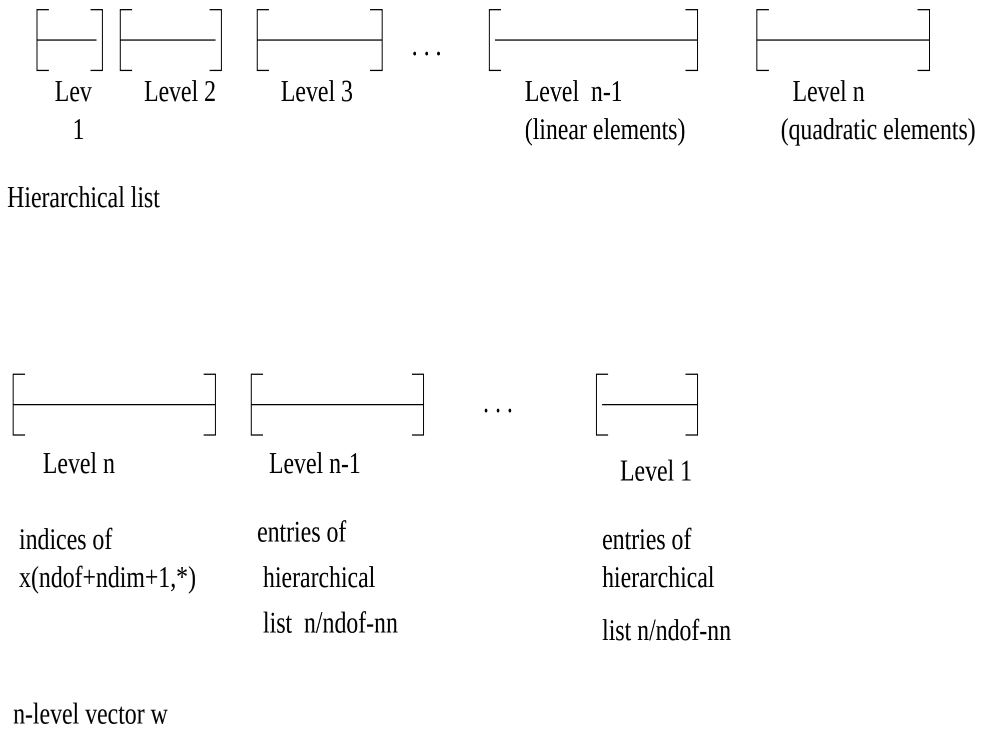

Lev 1
Level 2
Level 3
...
Level n-1 (linear elements)
Level n (quadratic elements)
Hierarchical list
Level n
Level n-1
...
Level 1
indices of x(ndof+ndim+1,*)
entries of hierarchical list n/ndof-nn
entries of hierarchical list n/ndof-nn
n-level vector w

Figure 7: Ordering of the hierarchical list.

Now, we give a short description of the forwards and backwards handling with the hierarchical list `HIER(NHIER-1)`:

- forwards:

  1. Set `IH=1`,
  2. If `(IH≥NHIER)` then return,
  3. Set `NUMBER-OF-FATHERS=-HIER(IH+2)`,
  4. Do operation for the son `HIER(IH)`,
  5. Set `IH=IH+(NUMBER-OF-FATHERS+1)*3`,
  6. goto 2

- backwards:

  1. Set `IH=NHIER`,
  2. If `(IH≤2)` then return,
  3. If `(HIER(IH-7)<0)` then `NUMBER-OF-FATHERS=2`,
  4. Else if `(HIER(IH-13)<0)` then `NUMBER-OF-FATHERS=4`,
  5. Else if `(HIER(IH-25)<0)` then `NUMBER-OF-FATHERS=8`,
  6. Do operation for the son `HIER(IH)`,
  7. Set `IH=IH-(NUMBER-OF-FATHERS+1)*3`,
  8. goto 2

Note, that we need `HIER(IH+1), HIER(IH+4), HIER(IH+5), HIER(IH+7), HIER(IH+8),` . . . only for the BPX-preconditioner. On these entries we have stored additional information.

But because of memory reasons, we have to move these numbers corresponding to nodes on coarser levels up to `N/NFG-NN` for applying the q-level vector, see Figure 7. Otherwise, the indices `NN+1,...,N/NFG` are used in the q-level vector w. For a better understanding, see 6.5.5.

6.4 Subroutines for projecting hanging nodes

For projecting and the extension of hanging nodes, we have three subroutines. The first is *projhangnodbpx1* for projecting hanging nodes on edges, the second is *projhangnodet* for projecting hanging nodes on faces and the third is *projhangnode* for applying the adjoint operator of the first two projections. Note, that the operators for projecting hanging nodes on edges and faces are commute. The marking of hanging nodes on faces is done during the refinement of the mesh. A negative entry of `FACE(5,*)` depicts the mid-node as hanging node of this face. The marking of hanging nodes on edges is more difficult. For this purpose, we have

- `SUBROUTINE HANGNOD(NF,FACE,NE,EDGE,NVOL,SOLID)`

  for marking all hanging nodes and

- `SUBROUTINE KORREDG(NE,EDGE)`

  for removing the markings of hanging nodes.

The marking is done on `EDGE(4,*)`, see 2.2.3. Both subroutines are located in *gemnode.f*.

6.4.1 Projection of hanging nodes on edges

The projection of hanging nodes of edges is rather simple. We have to distinguish between quadratic (for 20 and 27 node elements) and linear projection (for 8 node elements), see Figure 8. This projection is done by

`SUBROUTINE PROHANGNODBPX1(NEIN,NE,EDGE,W,NN,X).`

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NEIN`	I	number of nodes per element
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`W`	I/O	preconditioned residuum vector $\underline{w}$

This subroutine is located in *projectors.f*.

6.4.2 Projection of hanging nodes on faces

This projection is very difficult and is different for 8-, 20- and 27-node elements and done by

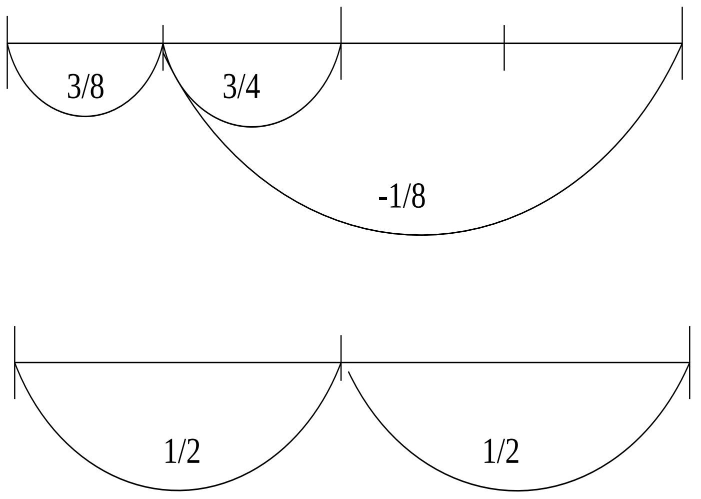


Figure 8: Projection of hanging nodes for quadratic (above) and linear elements (below).

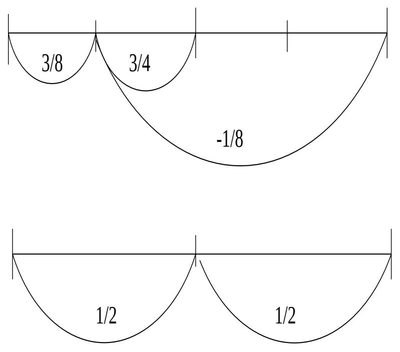


Figure 9: Numbering of the nodes for the sons.

`SUBROUTINE PROHANGNODET(NEIN,NFG,W,EDGE,NE,FACE,NF,X,NN).`

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NEIN`	I	number of nodes per element
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`W`	I/O	preconditioned residuum vector $\underline{w}$

This subroutine uses a certain ordering of the edges and nodes of the son faces j related to father face, see Figure 9.

The edges `FACE(1,J)` and `FACE(4,J)` connect a vertex of the father with the midpoint of an edge of the father and `FACE(2,J)` and `FACE(3,J)` connect edge midpoints with the face midpoint. This ordering is guaranteed during the call of *div4face*. The ordering of the nodes of a face depends only on the ordering of the edges. Hence, a call of *getnodefa* for a son face ensures that the first node is the vertex of the father, the second and fourth are midpoints of edges and the third is the midpoint of the father face. The fifth to eighth node are midpoints of the edges between first and second, second and third, third and fourth, fourth and first node.

6.4.3 The adjoint projector

The interpolation to hanging nodes is done by

`SUBROUTINE PROHANGNODE(NEIN,NFG,W,EDGE,NE,FACE,NF,X,NN).`

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NEIN`	I	number of nodes per element
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`W`	I/O	preconditioned residuum vector $\underline{w}$

This subroutine uses the same orderings of the edges of the sons as *projhangnodet*. This subroutines calls in the cases `NEIN=20` and `NEIN=27`

```
SUBROUTINE QUADINT(NFG,W,SON,VL,VM,VR)
```

for the quadratic interpolation of Figure 8. This subroutine is located in *projhangnode.f*.

6.5 Subroutines for applying the preconditioner

The application of the preconditioner C

$$\tilde{P}C^{-1}\tilde{P}^t\underline{r} = \underline{w}$$

is done in the

```
SUBROUTINE PRLOES(NN,X,NEIN,N,NEDG,EDGE,NF,FACE,A,LA,C,P,NCC,
                    CC,LCC,KETTE,IGLOB,HIER,NHIER,W,R,V).
```

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NEIN`	I	number of nodes per element
`N`	I	number of unknowns of the linear system
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NEDG`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`A`	I	stiffness matrix, element per element
`LA`	I	number of volumes and parameter `NEIN`
`C`	I	main diagonal of stiffness matrix
`P`	I	permutation vector for the coarse-grid nodes
`NCC`	I	number of coarse-grid nodes
`CC`	I	array of the coarse-grid matrix `CC`
`LCC`	I	array of the index-vector for the coarse-grid matrix
`R`	I	residuum vector $\underline{r}$
`W`	O	preconditioned residuum vector $\underline{w}$
`V`	H	additional vector
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`

This subroutine is located in *prloes.f*. The array `C` stores in the case of the BPX-preconditioner the main diagonals for all levels. For preparing this main diagonal an additional subroutine, *prevor*, is necessary. This subroutine is described in 6.6. The subroutine calls the subroutines for the preconditioniers and the subroutines for projecting hanging nodes induced by $\tilde{P}$.

6.5.1 Coarse grid solver

In the case of meshes with more nodes the absolute numbers of iterations of the pcg-method can be reduced by involving a coarse grid solver in the preconditioner. The subroutine

```
SUBROUTINE COARS(NEL,NEIN,SOLID,NN,X,NF,FACE,NE,EDGE,
                NCC,CC,LCC,N,D,H,MAX).
```

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NEIN`	I	number of nodes per element
`N`	I	number of unknowns of the linear system
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NEL`	I	number of elements
`SOLID`	I	array of elements `SOLID` with element stiffness matrices
`NCC`	O	dimension of the coarse-grid matrix
`CC`	O	Cholseky-decomposition of the coarse-grid matrix saved in VBX-format
`LCC`	O	index vector for the array `CC`
`D`	I	main diagonal of the stiffness matrix
`H`	H	additional vector
`MAX`	O	length of the array `CC`

computes the stiffness matrix A_0 on level 0 for linear elements. Furthermore, it computes the Cholesky decomposition

$$A_0 = R_0 R_0^T \tag{6.1}$$

and stores the matrix R_0 in VBZ-format. This subroutine is located in *coars.f*.

6.5.2 Jacobi preconditioner

The Jacobi preconditioner is the simplest of all. It only consists of a multiplication of the residual vector $\underline{r}$ with inverse D^{-1} of the main diagonal of the stiffness matrix. The condition number of $D^{-1}K$ equals $\mathcal{O}(h^2)$, where K is the stiffness matrix of our global problem and h is the discretization parameter.

6.5.3 Yserentant preconditioner

The Yserentant preconditioner [8] is based on the hierarchy of the finite element meshes. It can be written in the form

$$C^{-1} = SS^t. \tag{6.2}$$

Here, S is the basis transformation matrix which transfers the usual nodal basis to the h-hierarchical basis. For more details, see [1].

If we use the coarse grid solver, we get the following form:

$$\begin{aligned} C^{-1} &= SA_0S^t, \quad \text{with} \\ A_0 &= \begin{cases} \delta R_0 R_0^t & \text{on the coarse grid} \\ I & \text{else.} \end{cases} \end{aligned}$$

$R_0R_0^t$ is the Cholesky-decomposition of the coarse grid matrix A_0 (6.1). The parameter δ have been found empirically.

If we have jumping coefficients in the differential equation, a Jacobi modification of the form

$$C^{-1} = SD^{-1}S^t$$

with the main diagonal D of the stiffness matrix whose elements are scaled with the mesh-size h_l of the level l of the point it belongs to is useful. The condition number of $C^{-1}K$ is equal to $\mathcal{O}(h^{-1})$ in the three-dimensional case. This is an improvement in comparison to the Jacobi preconditioner, but it still cannot satisfy. For applying the Yserentant preconditioner the subroutine *prloes* calls

```
SUBROUTINE NHSTMUL(NFG,W,NHIER,HIER,NN,X)
```

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`

`W`	I/O	preconditioned residuum vector $\underline{w}$
`NFG`	I	degrees of freedom per node
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`

for the operation $S^t\underline{w}$ and

```
SUBROUTINE NHISMUL(NFG,W,NHIER,HIER,NN,X)
```

`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`W`	I/O	preconditioned residuum vector $\underline{w}$
`NFG`	I	degrees of freedom per node
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`

for $S\underline{w}$. Both subroutines are located in *nhieyser.f*. Note that the number of nodes and index for the nodal shape functions associated to this node are different. Hence, it is necessary to call *nodeglobal*. Before we can use the subroutine *prloes* for the Yserentant preconditioner one initialization step is necessary for obtaining the multiple diagonal D. This generation of the diagonal D does

```
SUBROUTINE DIA3YS(NFG,N,NHIER,HIER,C,NK,SC,NEIN,X).
```

`NK`	I	number of nodes
`N`	I	dimension of linear system
`X`	I	array of node coordinates `X`
`C`	I/O	main diagonal for all levels
`NFG`	I	degrees of freedom per node
`SC`	H	additional vector
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`
`NEIN`	I	number of nodes per elements

Additionally, this subroutine does the different scalings for the level-jumps from linear to serendipity and tri-quadratic elements on level q.

6.5.4 BPX-Preconditioner

The BPX-preconditioner [4] is also a hierarchical preconditioner. It can be written in the form

$$w = \sum_{l=0}^{q} \sum_{i \in \mathcal{N}_l} \phi_i^l \langle r, \phi_i^l \rangle,$$

where $\mathcal{N}_l$ is the index set of all nodes which are sons on level l or fathers having a son on this level a subset of all nodes of level l. Here

$$\langle r, v \rangle = a(u,v) - \langle f, v \rangle \ \forall v \in (H_0^1(\Omega))^d$$

defines 'the residual functional'. Note that $\underline{r} = (\langle r, \phi_i^q \rangle)_{i=1}^{N_q}$ is given. In matrix notation, we obtain

$$\underline{w} = \sum_{l=0}^{q} \sum_{i \in \mathcal{N}_l} Q_l \begin{pmatrix} I \\ \mathbf{0} \end{pmatrix} \tilde{y}^{(l)}$$

with

$$\tilde{y}^{(l)} = (\langle r, \phi_i^l \rangle) = \begin{pmatrix} I & \mathbf{0} \end{pmatrix} Q_l^t \underline{r}.$$

Thus, we obtain

$$\begin{aligned} \underline{w} &= \sum_{l=0}^{q} Q_l \begin{pmatrix} I \\ \mathbf{0} \end{pmatrix} \begin{pmatrix} I & \mathbf{0} \end{pmatrix} Q_l^t \underline{r} \\ &= \sum_{l=0}^{q} Q_l \begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} Q_l^t \underline{r} \\ &= \sum_{l=0}^{q} Q_l P_l Q_l^t \underline{r} \end{aligned}$$

with the projection matrix $P_l = (p_{ij})$ from all nodes on level l onto the subset $\mathcal{N}_l$. The matrix Q_l^t is the transformation matrix from the usual basis of the space V_q to the space V_l. Therefore, we have for $l < q$

$$Q_l = Q_{q-1}^q \cdot \ldots \cdot Q_l^{l+1}$$

and

$$Q_q = I$$

with

$$(Q_{l-1}^l)_{ij} = \begin{cases} 1 & \text{(a)} \quad \text{if } i = j, \quad i,j = 1,2,\ldots,\mathtt{N}_{l-1} \\ \frac{1}{2} & \text{(b)} \quad \text{if } j = i_1 \text{ and } j = i_2 \text{, where } P^{(i)} \text{ is the midpoint between } P^{(i_1)} \text{ and } P^{(i_2)} \text{ which are the end points of an edge of a hexahedron from the mesh } \mathcal{T}_{l-1} \\ \frac{1}{4} & \text{(c)} \quad \text{if } j = i_k,\ k = 1,\ldots,4 \text{, where } P^{(i)} \text{ is the midpoint of the face with the four vertices } P^{(i_k)} \\ \frac{1}{8} & \text{(d)} \quad \text{if } j = i_k,\ k = 1,\ldots,8 \text{, where } P^{(i)} \text{ is the midpoint of the volume with the eight vertices } P^{(i_k)} \\ 0 & \quad\quad \text{else.} \end{cases} \tag{6.3}$$

We introduce

$$\tilde{Q}_l = Q_l P_l$$

and

$$Q_l^{l-1} = (Q_{l-1}^l)^t.$$

Then, we can write

$$\tilde{Q}_l = Q_{q-1}^q \cdot \ldots \cdot Q_{l+1}^{l+2} \tilde{Q}_l^{l+1}$$

with

$$(\tilde{Q}_{l-1}^l)_{ij} = \begin{cases} 1 & \text{(a) in (6.3) and node in } \mathcal{N}_l \\ \frac{1}{2} & \text{(b) in (6.3) and nodes in } \mathcal{N}_l \\ \frac{1}{4} & \text{(c) in (6.3) and nodes in } \mathcal{N}_l \\ \frac{1}{8} & \text{(d) in (6.3) and nodes in } \mathcal{N}_l \\ 0 & \text{else.} \end{cases} \tag{6.4}$$

Hence, we can determine $\underline{w}$ using the following Horner-scheme like algorithm.

- Set $y^{(q)} := \underline{r}$.
- For $l = q-1, \ldots, 1$ do
$$y^{(l-1)} = Q_l^{l-1} y^{(l)}$$
endfor.
- Set $z_0 := y_0$

- for $l = 1, \ldots, q$ do

$$z^{(l)} := P_l y^{(l)} + Q_{l-1}^l z^{(l-1)} \tag{6.5}$$

  endfor.

- Set $\underline{w} = z^{(q)}$.

In the case of jumping coefficients in the differential equation a Jacobi modification is helpful. This modification has the form

$$\tilde{P}_l = P_l \hat{D}^{-1} \tag{6.6}$$

where $\hat{D}$ is the extracted main diagonal of the stiffness matrix corresponding to V_l. Its elements are scaled with the mesh size h_i of the zone i of the point it belongs to.
Note that for a node $j \notin \mathcal{N}_l$

$$(Q_{l-1}^l y^{(l-1)})_j = (y^{(l-1)})_j$$

and

$$(P_l y^{(l)})_j = 0$$

are valid, i.e. the application of Q_{l-1}^l does not change the value on the j-th coordinate, and with (6.5) we have

$$(z^{(l-1)})_j = (z^{(l)})_j.$$

Hence, we introduce the following data structure. On level l only nodes of $\mathcal{N}_l$) obtain a memory place in the vector $\tilde{y}^{(l)}$, part of $y^{(l)}$.

For applying the BPX-preconditioner the subroutine *prloes* calls the subroutines *nhstmulbpx* and *nhismulbpx*. Both routines are located in the file *hiebpx.f*.

```
SUBROUTINE NHSTMULBPX(NFG,N,W,NHIER,HIER,M,NN,X)
```

`NN`	I	number of nodes
`N`	I	dimension of linear system
`X`	I	array of node coordinates `X`
`W`	I/O	preconditioned residuum vector $\underline{w}$
`NFG`	I	degrees of freedom per node
`M`	H	additional vector
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`

```
SUBROUTINE NHISMULBPX(NFG,N,W,NHIER,HIER,M,NN,X)
```

`NN`	I	number of nodes
`N`	I	dimension of linear system
`X`	I	array of node coordinates `X`
`W`	I/O	preconditioned residuum vector $\underline{w}$
`NFG`	I	degrees of freedom per node
`M`	H	additional vector
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`

The first routine does the operation $y^{(l)} = Q_l^t \underline{r}$, the second $\sum Q_l P_l y^{(l)}$ (using $\tilde{y}^{(l)}$ only). Note that the number of nodes and index for the nodal shape function associated to this node are different in all levels. In the finest level, the index is stored in last entry of the array `X`, see 2.2.4. The management of the nodes of the coarser levels is described in 6.3.

Before we can use the subroutine *prloes* for the BPX preconditioner some initialization steps are necessary. The first one consists on storing the information of the numbers of the nodes in preceeding levels, the second on obtaining the scaling factors for the main diagonals in the levels l, $l = 1, \ldots, q$.

- For the first step, we have implemented

  ```
  SUBROUTINE NHIE3BPX(N,NHIER,HIER,NN,MFR,MTO,X,NEIN).
  ```

`N`	I	dimension of linear system
`X`	I	array of node coordinates `X`
`NN`	I	number of nodes
`MFR`	H	additional vector
`MTO`	H	additional vector
`NHIER`	I	length of hierarchical list
`HIER`	I/O	array of the hierarchical list `HIER`
`NEIN`	I	number of nodes per elements

  This subroutine is located in *hiemul.f*. For repeating parts of the program, two simple subroutines, *help2bpx* and *help3bpx* are used.

- For the second step, we have

  `SUBROUTINE NDIA2BPX(NFG,N,NHIER,HIER,C,NN,SC,X,NEIN).`

`NN`	I	number of nodes
`N`	I	dimension of linear system
`X`	I	array of node coordinates `X`
`C`	I/O	main diagonal for all levels
`NFG`	I	degrees of freedom per node
`SC`	H	additional vector
`NHIER`	I	length of hierarchical list
`HIER`	I	array of the hierarchical list `HIER`
`NEIN`	I	number of nodes per elements

  This subroutine generates the all level main diagonal. From

$$\frac{a(\phi_j^l, \phi_j^l)}{a(\phi_j^{l-1}, \phi_j^{l-1})} = \frac{1}{2}$$

  for nodal basis functions ϕ_j^l and ϕ_j^{l-1} associated to the same node for linear elements on Levels l and $l-1$, we obtain the scaling factor $\frac{1}{2}$ for the 'jump' between two linear levels. Between the serendipity elements and linear elements on the same level, this factor is $\frac{49}{30}$. Furthermore, this factor is $\frac{28}{75}$ between linear and tri-quadratic elements on the same level. This subroutine is located in *hiedia.f*

As in the case of the Yserentant preconditioner (6.2), a coarse grid solver `ivar=5` can be included. The subroutines `COARS_FOR` and `COARS_BACK` located in *hiecoars.f* are auxiliary routines for the application of the coarse-grid solver in case of the BPX-preconditioner.

6.5.5 An example for understanding the preconditioniers

Consider now the simple one-dimensional example of Figure 10. The number above the edges are the node numbers j, the numbers below are the memory places in the global vector, saved in `X(NDIM+NDOF+1,J)`, which we denote in this subsection for simplicity by `MP(J)`. The entries of this array are written

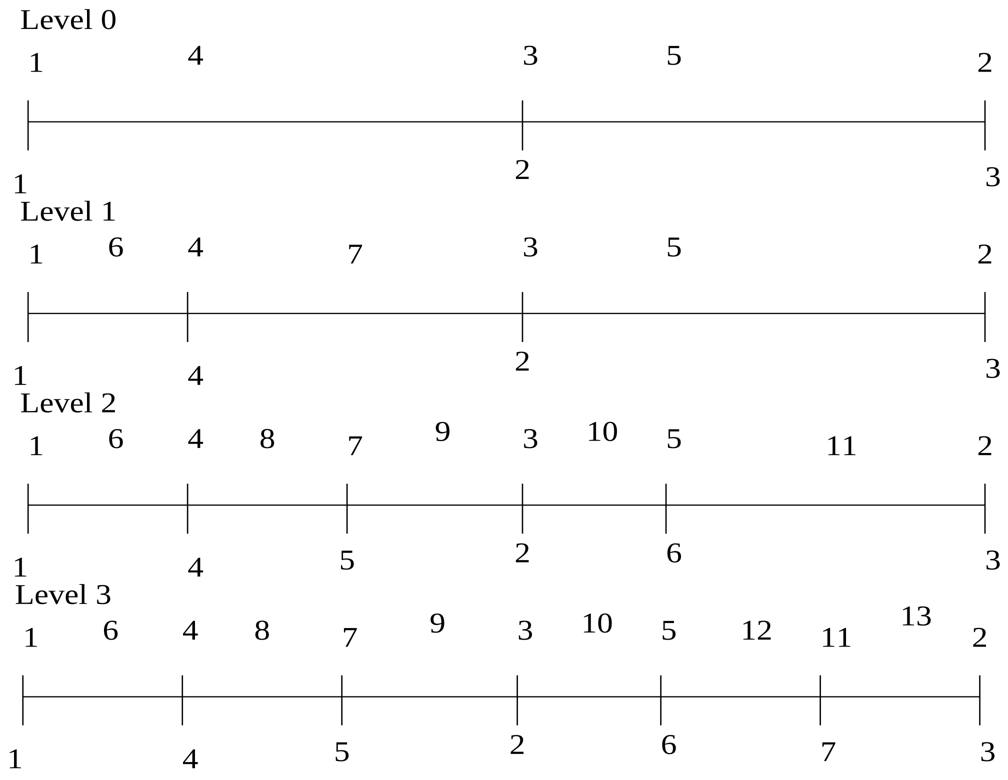


Figure 10: Example for understanding the BPX-preconditioner.

in the following table.

J	1	2	3	4	5	6	7	8	9	10	11	12	13
MP(J)	1	3	2	4	6	0	5	0	0	0	7	0	0

(6.7)

We have for linear elements 3 and for quadratic elements 4 level-hops, the coarsest one is from level 0 to 1. For the Yserentant-preconditioner, the beginning of the hierarchical list looks as in the following table. Informations in brackets are informations of this node.

IH	1	2	3	4	5	6	7	8	9
HIER(IH)	(4	0	-2)	(1	0	0)	(3	0	0)
IH	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
HIER(IH)	(5	0	-2)	(3	0	0)	(2	0	0)

. . .

The node 4 has 2 fathers (`HIER(3)=-2`), the nodes 1 (`HIER(4)=1`) and 3 (`HIER(7)=3`), the node 5 has the two fathers 3 and 2. The remaining entries are only needed for the BPX-preconditioner.
We explain now how we obtain these entries. Only the node 4 is new in level 1 and has the two fathers node 1 and node 2. In level 2, the nodes 5 with the fathers node 2 and node 3, and 7 with the fathers node 2 and node 4 are sons. The node 11, new in level 3, has the fathers node 2 and node 5. The remaining nodes are only introduced for quadratic elements. In matrix representation, the BPX-preconditioner looks as follows. We have

$$Q_3^2 = \begin{pmatrix} & 0 \\ & 0 \\ I_6 & \frac{1}{2} \\ & 0 \\ & 0 \\ & \frac{1}{2} \end{pmatrix}, \quad Q_2^1 = \begin{pmatrix} & 0 & 0 \\ I_4 & \frac{1}{2} & \frac{1}{2} \\ & 0 & \frac{1}{2} \\ & \frac{1}{2} & 0 \end{pmatrix}$$

and

$$Q_1^0 = \begin{pmatrix} & \frac{1}{2} \\ I_3 & \frac{1}{2} \\ & 0 \end{pmatrix}$$

as the basis-transformation matrices between the levels l and $l-1$. The node 2 (memory place 3) has on level 1 no son, the node 1 (1) has no son on level 2, and the nodes 1(1), 3 (2), 4 (4) and 7 (5) have on level 3 no sons. Therefore, for the matrices P_l, we obtain

$$P_0 = I_3 \quad , \quad P_1 = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 0 & \\ & & & 1 \end{pmatrix},$$

$$P_2 = \begin{pmatrix} 0 & \\ & I_5 \end{pmatrix} \quad \text{and} \quad P_3 = \begin{pmatrix} \mathbf{0}_2 & & & \\ & 1 & & \\ & & \mathbf{0}_2 & \\ & & & I_2 \end{pmatrix}.$$

Consider now the certain levels. The numbers in brackets are the corresponding memory places for each node.

- Level 3:
  The nodes 2 (3), 5 (6) and 11 (7) need a memory place. The nodes 2 and 5 exist in a coarser level and get then the memory places 8 and 9.

- Level 2:
  The nodes 2 (8), 3 (2), 4 (4), 5 (9) and 7 (5) need a memory place. They have from `MP(I)` (for the nodes 3, 4 and 7) or from finer levels (nodes 2 and 5) their memory places. The nodes 2, 3 and 4 are fathers and exist in a coarser level. Then, they obtain the memory places 12, 10 and 11.

- Level 1:
  The nodes 1 (1), 3 (10) and 4 (11) have memory places. The nodes 1 and 3 have in a coarser level the memory places 13 and 14.

- Level 0:
  All nodes, 1 (13), 2 (12) and 3 (14), have memory places.

In the hierarchical list, these informations of the nodes on level l and level $l-1$ are stored. We obtain for the array `HIER` the entries

4	17	-2	1	1	19	3	16	20
5	15	-2	3	3	16	2	14	18
7	7	-2	3	3	16	4	4	17
11	11	-2	2	2	14	5	5	15

.

This list is a code of the operation for the BPX-preconditioner. Now, we describe the code in this example. We have `NN=13`, but `N=7`. Hence, we have to subtract $6 = 13 - 7$ from each entry of the hierarchical list greater than 13. For the lower entries j we have to determine `MP(J)` (6.7), which gives the memory places in the global vectors. The steps

$$\begin{aligned} \underline{w}_8 &= \underline{w}_3 + \frac{1}{2}\underline{w}_7, \\ \underline{w}_9 &= \underline{w}_6 + \frac{1}{2}\underline{w}_7 \end{aligned}$$

for level 3,

$$\begin{aligned} \underline{w}_{10} &= \underline{w}_2 + \frac{1}{2}(\underline{w}_9 + \underline{w}_5), \\ \underline{w}_{11} &= \underline{w}_4 + \frac{1}{2}\underline{w}_5, \\ \underline{w}_{12} &= \underline{w}_8 + \frac{1}{2}\underline{w}_9 \end{aligned}$$

for level 2,

$$\begin{aligned} \underline{w}_{13} &= \underline{w}_1 + \frac{1}{2}\underline{w}_{11}, \\ \underline{w}_{14} &= \underline{w}_{10} + \frac{1}{2}\underline{w}_{11} \end{aligned}$$

for level 1 forwards and

$$\begin{aligned} \underline{w}_{11} &= \underline{w}_{11} + \frac{1}{2}(\underline{w}_{13} + \underline{w}_{14}), \\ \underline{w}_1 &= \underline{w}_1 + \underline{w}_{13}, \\ \underline{w}_{10} &= \underline{w}_{10} + \underline{w}_{14} \end{aligned}$$

for level 1,

$$\begin{aligned} \underline{w}_9 &= \underline{w}_9 + \frac{1}{2}(\underline{w}_{10} + \underline{w}_{12}), \\ \underline{w}_5 &= \underline{w}_5 + \frac{1}{2}(\underline{w}_{11} + \underline{w}_{10}), \\ \underline{w}_4 &= \underline{w}_4 + \underline{w}_{11}, \\ \underline{w}_8 &= \underline{w}_8 + \underline{w}_{12}, \\ \underline{w}_2 &= \underline{w}_2 + \underline{w}_{10} \end{aligned}$$

for level 2,

$$\begin{aligned}\underline{w}_7 &= \underline{w}_7 + \frac{1}{2}(\underline{w}_8 + \underline{w}_9), \\ \underline{w}_3 &= \underline{w}_3 + \underline{w}_8, \\ \underline{w}_6 &= \underline{w}_6 + \underline{w}_9\end{aligned}$$

level 3 backwards are done by this hierarchical list.

6.6 Further subroutines

6.6.1 Parameters of *stave*

```
SUBROUTINE STAVE(U,NEIN,EDGE,NE,FACE,NF,X,NN,DIR,
                                     NDIR,H,KETTE,IGLOB)
```

`U`	O	solution vector u
`NEIN`	I	number of unknowns per element
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NN`	I	number of nodes
`X`	I	array of node coordinates `X`
`NDIR`	I	number of Dirichlet faces
`DIR`	I	the Dirichlet data `DIR`
`H`	H	additional vector

This subroutine copies the function values from the array `X` of node coordinates and values, see 2.2.4, to the vector `U` as a good initial vector for the PCG-method. Furthermore, the Dirichlet boundary conditions are build in. This procedure is located in *stave.f*.

6.6.2 Parameters of *ppcgm*

```
SUBROUTINE PPCGM(NK,KOOR,NEIN,N,NEDG,EDGE,NF
               ,FACE,A,LA,C,B,X,EPS,IT,R,W,S,V,NCC,P,
                CC,LCC,KETTE,IGLOB,NHIER,HIER)
```

`U`	O	solution vector u
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NEDG`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NK`	I	number of nodes
`KOOR`	I	array of node coordinates `KOOR`
`NEIN`	I	number of unknowns per element
`N`	I	dimension of the linear system
`A`	I	stiffness matrix, element-wise stored
`B`	I	right hand side
`C`	I	main diagonal of stiffness matrix
`R`	H	additional vector
`S`	H	additional vector
`V`	H	additional vector
`W`	H	additional vector
`P`	H	additional vector
`NCC`	I	number of coarse-grid nodes
`CC`	I	array of the coarse-grid matrix `CC`
`LCC`	I	array of the index-vector for the coarse-grid matrix
`EPS`	I	relative accuracy `epsilon`
`IT`	I/0	maximal number of iterations
`LA`	I	number of volumes and `NEIN`
`NHIER`	I/O	length of hierarchical list
`HIER`	I/O	array of the hierarchical list `HIER`

This subroutine solves the linear system

$$A\underline{u} = \underline{b},$$

where the matrix A is stored element-wise using the pre-conditioned conjugate gradient method. This procedure is located in *ppcgm.f.*

6.6.3 Parameters of *prevor*

```
SUBROUTINE PREVOR(NK,N,A,LA,C,EDGE,NCC,P,CC,LCC,
                  KETTE,IGLOB,V,IPOD,NHIER,HIER,NEIN,X)
```

`EDGE`	I	array of edges `EDGE`
`NK`	I	number of nodes
`X`	I	array of node coordinates `X`
`NEIN`	I	number of unknowns per element
`N`	I	dimension of the linear system
`A`	I	stiffness matrix, element-wise stored
`C`	I	main diagonal of stiffness matrix
`V`	H	additional vector
`P`	O	permutation vector for the coarse-grid nodes
`NCC`	I	number of coarse-grid nodes
`CC`	I	array of the coarse-grid matrix `CC`
`LCC`	I	array of the index-vector for the coarse-grid matrix
`IPOD`	O	marking for errors
`LA`	I	number of volumes and `NEIN`
`NHIER`	I/O	length of hierarchical list
`HIER`	I/O	array of the hierarchical list `HIER`

This subroutine does the described initialization steps for the BPX- and Yserentant preconditioner. This procedure is located in *prevor.f*.

6.6.4 Parameters of *axmebe*

```
SUBROUTINE AXMEBE(ADD,NEL,NEO,EL,N,U,Y,X)
```

`ADD`	I	character variable for kind of multiplication
`NEL`	I	number of elements
`X`	I	array of node coordinates `X`
`NEO`	I	number of unknowns per element
`N`	I	dimension of A and the vectors
`U`	I	input vector $\underline{u}$
`Y`	O	output vector $\underline{y}$
`EL`	I	stiffness matrix A, element-wise stored

This subroutine does the multiplication

$$\underline{y} = A\underline{u}$$

in the case of `ADD='0'` and

$$\underline{y} = \underline{y} + A\underline{u}$$

in the case of `ADD='+'`, if the matrix is stored by the element stiffness matrices A_{el}. This subroutine is located in *axmebe.f*. Note, that

$$A = \sum_{Elements} L_{el}^t A_{el} L_{el}. \tag{6.8}$$

This multiplication is done by the 3 operations of (6.8).

- `SUBROUTINE VDPERM(N,X,Y,P)` and
  `SUBROUTINE VDOUTELVAL(N,NDOF,X,Y,EL)` do the multiplication

$$\underline{x} = L_{el}\underline{u}.$$

- `SUBROUTINE SMAMVEKD(N,Y,A,X)` does the matrix vector multiplication

$$\underline{y}_{el} = A_{el}\underline{x}.$$

- The subroutine *fakku*, see 5.2.3, does

$$\underline{y} = \sum L_{el}^t \underline{y}_{el}.$$

The subroutines *vdperm* and *vdoutelval* are located in *vdperm*.

6.6.5 Subroutines located in *value.f*

The subroutine *backutox* saves the solution from the solution vector `U` back to the array of node coordinates `X`. Note, that for linear and serendipity elements not each existing node has a shape function associated to this node. The

`SUBROUTINE VALUE(NVOL,SOLID,NF,FACE,NE,EDGE,NK,X,NEIN,PHI)`

`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NK`	I	number of nodes
`X`	I/O	array of node coordinates `X`
`NEIN`	I	number of unknowns per element
`NVOL`	I	number of volumes
`SOLID`	I	array of solids `SOLID`
`PHI`	H	additional vector

determines the solution on these nodes.

6.7 Tree structure

Tree substructures of subroutines marked with the symbol ∗ are described before in the list.

6.7.1 CHGHIER

```
CHGHIER
```

6.7.2 HIERANF

```
HIERANF
    ↪ HIERARCH
    ↪ GETNODEFA∗
    ↪ GETNODE∗
```

6.7.3 HANGNOD

```
HANGNOD
    ↪ EDORTOFA∗
```

6.7.4 KORREDG

```
KORREDG
```

6.7.5 STAVE

```
STAVE
    ↪ GETNODEFA∗
    ↪ DVALUE
    ↪ NODEGLOBAL∗
    ↪ PROJHANGNODE∗
```

6.7.6 PPCGM

```
PPCGM
    ↪ PREVOR
        ↪ NHIE3BPX
            ↪ HELP2BPX
            ↪ HELP3BPX
        ↪ NDIA2BPX
        ↪ DIA3YS
            ↪ NODEGLOBAL∗
    ↪ AXMEBE
        ↪ VDPERM
        ↪ VDOUTELVAL
        ↪ SMAMVEKD
        ↪ FAKKU∗
        ↪ NODEGLOBAL∗
    ↪ PRLOES
        ↪ PROJHANGNODBPX1
            ↪ NODEGLOBAL∗
        ↪ PROJHANGNODET
            ↪ NODEGLOBAL∗
            ↪ GETNODEFA∗
        ↪ NHSTMUL
            ↪ NODEGLOBAL∗
        ↪ NHSTMULBPX
        ↪ NHISMUL
            ↪ NODEGLOBAL∗
        ↪ NHISMULBPX
        ↪ PROJHANGNODE
            ↪ NODEGLOBAL∗
            ↪ QUADINT
        ↪ GETNODEFA*
    ↪ COARS_FOR
    ↪ COARS_BACK
↪ ZWISCH
```

6.7.7 VALUE

```
VALUE
    ↪ DOMASTERVAL*
```

6.7.8 BACKUTOX

```
BACKUTOX
COARS
    ↪ QUAD_TO_LIN
    ↪ NUMBER3D
    ↪ NODEGLOBAL*
```

7 Error estimation

7.1 Error estimator

For a given numerical solution u_h, an estimate about the error $u - u_h$ is needed. We have implemented a Zienciewicz-Zhu error estimator, [5]. For this estimator, we need two tools, the gradient jump along faces, e.g. for (1.1)

$$jump(Face, u) = \int_{Face} \left[(\nabla u)^t A(x) n\right]^2 \, \mathrm{d}s \tag{7.1}$$

where n denotes the normal vector and $[v]$ the jump of v along this face, and residuum in the element, e.g. for (1.1)

$$res(Element, u) = \int_{Element} (Lu - f)^2 \, \mathrm{d}x. \tag{7.2}$$

Both integrals are estimated by a simple midpoint rule

$$\int_E f(x) \mathrm{d}x \approx \mathrm{meas}(E) f(x_m), \tag{7.3}$$

where x_m denotes the midpoint of E. These integrals and the needed derivatives are calculated by *efromu*. Here, second derivatives on the elements are required. In the appendix A, we describe the algorithm in order to compute the second derivatives using the Jacobians and the first and second derivatives of the form functions. The subroutine *errorest* does the calculation of the error estimator and marks all elements. We will describe both routines in the next subsections.

7.2 Description of *efromu*

```
SUBROUTINE EFROMU(NFG,N2DIM,NN,U,X,NVOL,NEIN,SOLID,NE,EDGE,NF,
                  FACE,NDIR,DIR,NNEUM,NEUMW,NMAX,NMATE,RMATE,
                  EST,E,XU,DN,PHI,T,D2,T2,H)
```

`NN`	I	number of nodes
`X`	I	pointer to array of node coordinates `X`
`NVOL`	I	number of volumes
`NEIN`	I	number of unknowns per element

`SOLID`	I	array of solids `SOLID`
`NE`	I	number of edges
`EDGE`	I	array of edges `EDGE`
`NF`	I	number of faces
`FACE`	I	array of faces `FACE`
`NDIR`	I	number of Dirichlet b.c.
`DIR`	I	array of Dirichlet b.c.`DIR`
`NNEUM`	I	number of Neumann b.c.
`NEUMW`	I	array of Neumann b.c.`NEU`
`NMATE`	I	number of material
`NMAX`	I	number of material informations
`RMATE`	I	array of materials `MAT`
`E`	O	integrals (7.1) and measures for each face
`EST`	O	integrals (7.2) for each volume
`NFG`	I	degrees of freedom
`H`	H	additional vector
`N2DIM`	I	number of unknowns per face
`DN`	H	additional vector
`PHI`	H	additional vector
`T`	H	additional vector
`D2`	H	additional vector
`T2`	H	additional vector

This subroutine determines the integrals (7.1) and (7.2) by the midpoint rule (7.3). In the array `EST` only the values of the integral (7.2) are saved. For `E`, we have the data structure `E(NFG+1,*)`, because for (1.2) relation (7.1) is a integral for each component of the vector $\underline{u}$. This information is saved on positions $2, \ldots, 1+$`NFG`. On the first position of `E`, we save for each face F

$$E(1, F) = \sum_{\text{Volumes with} F} \text{meas}(Volume).$$

This subroutine is located in *efromu.f*.

7.3 Description of *errorest*

```
SUBROUTINE ERROREST(NFG,NVOL,NEIN,SOLID,NF,FACE,
                 E,EREST,ERROUT,X)
```

`X`	I	pointer to array of node coordinates `X`
`NVOL`	I	number of volumes
`NEIN`	I	number of unknowns per element
`SOLID`	I/O	array of solids `SOLID`
`NF`	I	number of faces
`FACE`	I/O	array of faces `FACE`
`E`	I	integrals (7.1) and measures for each face
`EREST`	I	integrals (7.2) for each volume
`NFG`	I	degrees of freedom
`ERROUT`	O	total error

This subroutine calculates the error estimator. It is possible to choose between a face oriented error estimator without estimating the residuum and an element oriented error estimator which estimates residuum and gradient jumps over all faces. In the first case the marking for subdividing a face i is done by `FACE(10,I)=1`, in the second case for subdividing a volume j by `SOLID(1,j)<0`. The subroutine is located in *efromu.f*.

7.4 Tree structures

Tree substructures of subroutines marked with the symbol $*$ are described before in the list.

```
EFROMU
  ↪ GETMATERIAL∗
  ↪ NODEGLOBAL∗
  ↪ MATTMUL∗
  ↪ GETNODEU∗
  ↪ SOLV33T∗
  ↪ DOMASTERVAL∗
  ↪ XMASTER∗
  ↪ SOLV33
      ↪ DET3∗
  ↪ CROSSPRO
      ↪ DETER22
  ↪ GETNODEFA∗
  ↪ RANDKOOR∗
```

```
ERROREST
```

Nodes	Elements	Faces	Time for Assem	Solver Iter.	Solver Time	Error	Error·N
27	1	6	0.000	3	0.000	1.3E+04	1.E+05
125	8	42	0.008	8	0.000	1.8E+03	3.E+04
413	29	150	0.023	11	0.008	5.7E+02	2.E+04
673	57	258	0.027	12	0.012	4.9E+01	3.E+03
1053	85	402	0.027	11	0.020	2.9E+01	2.E+03
2025	169	786	0.086	13	0.059	1.5E+01	2.E+03
2897	260	1146	0.094	13	0.094	5.8E+00	8.E+02
3163	288	1254	0.031	15	0.117	2.3E+00	3.E+02
4115	386	1638	0.098	15	0.156	2.4E+00	4.E+02
4577	428	1818	0.047	15	0.176	7.2E-01	1.E+02
5873	533	2322	0.109	14	0.227	5.1E-01	1.E+02

Table 30: Refinement history for layer3.

8 Numerical examples

We will give some numerical test examples. All examples are discretized with serendipity elements, e.g. `NEIN=20`. The linear system is solved with the BPX-preconditioner. The relative accuracy in the preconditioned energy norm is 10^{-2}, and the initial vector in the PCG-method is the solution of the preceeding levels. Hence, this accuracy is enough.

8.1 Layer3

The first example considers a boundary layer for the convection- diffusion equation. More precisely, we solve

$$\begin{aligned} -\epsilon\triangle u + u &= 1 \text{ in } \Omega = (0,1)^3, \\ u &= 0 \text{ on } x=0, y=0, z=0, \\ \frac{\partial u}{\partial n} &= 0 \text{ on } x=1, y=1, z=1. \end{aligned}$$

After 13 steps of refinement we get the mesh of Figure 11. The coarse mesh consists of the cube $(0,1)^3$. The vertex $(0,0,0)$ is not visible. The refinement history is displayed in Table 30.

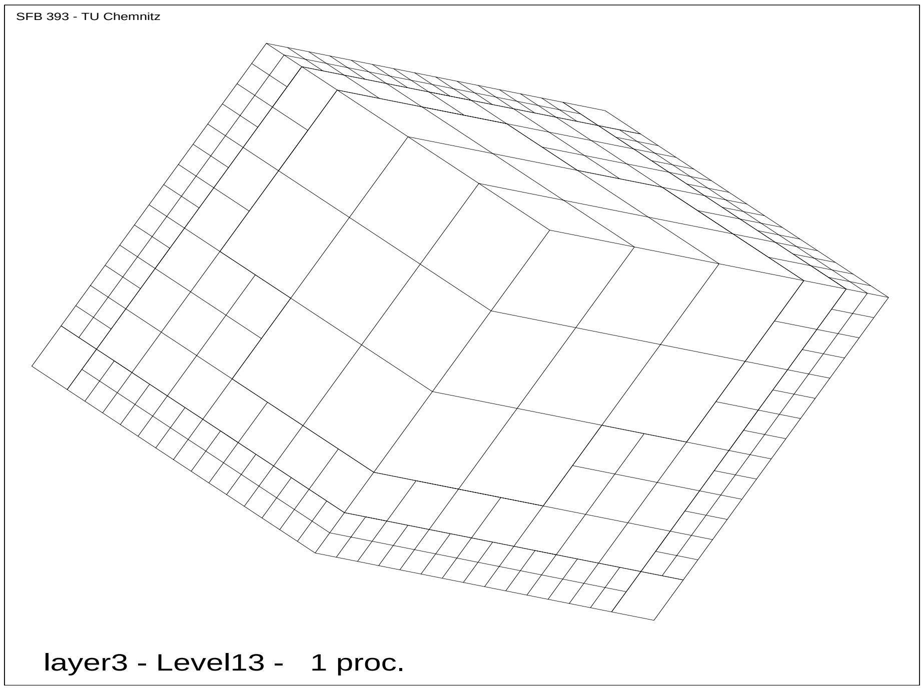


Figure 11: Layer3 after 13 refinements.

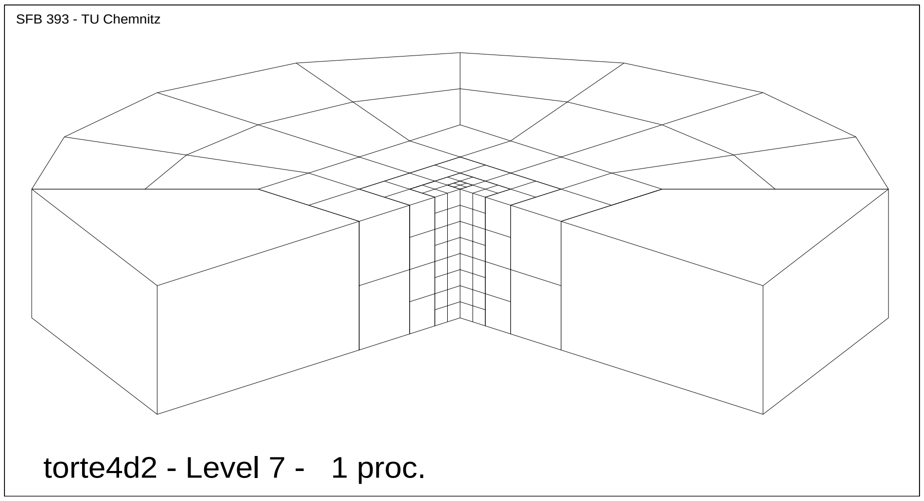


Figure 12: Torte4d2 after 13 refinements.

8.2 Torte4d2

The second example considers a layer near an non-convex edge. More precisely, we solve

$$\begin{aligned} -\triangle u &= 0 \,\text{in}\, \Omega, \\ u &= 100 \,\text{on}\, \Gamma_1. \end{aligned}$$

The domain Ω is in cylinder coordinates (r, ϕ, z) the domain $(0,3) \times (0, \frac{3}{2}\pi) \times (0,1)$, $\Gamma_1 = \partial\Omega$.

After 13 steps of refinement we get the mesh of Figure 12. The refinement history is displayed in Table 31.

8.3 Ct01

The last example considers the Lamé-equation (1.2) in the domain $\Omega = (0,2) \times (0,1) \times (0,4)$. and the boundary conditions $\partial\Omega_1^i = (0,1) \times (0,1) \times \{0\}$ for $i = 1, \ldots, 3$ and $\partial\Omega_2^{(3)} = (0,2) \times (0,1) \times \{4\}$ with $g^{(3)} = 1000$. The coarse mesh consists of 8 cubes of size 1. The point $(2,1,4)$ is not visible. After 5 steps of refinement we get the mesh of Figure 13. The refinement history is displayed in Table 32.

Nodes	Elements	Faces	Time for Assem	Solver Iter.	Solver Time	Error	Error·N
141	9	41	0.008	3	0.000	1.3E+02	3.E+03
239	16	77	0.008	5	0.000	2.2E+01	6.E+02
501	37	177	0.023	7	0.004	1.4E+01	6.E+02
583	44	209	0.008	7	0.004	2.9E+00	1.E+02
1079	86	401	0.043	8	0.020	1.8E+00	1.E+02
1227	100	461	0.016	8	0.020	5.9E-01	5.E+01
2037	170	777	0.070	10	0.047	3.8E-01	4.E+01
2527	219	977	0.051	10	0.066	3.6E-01	5.E+01
2757	240	1069	0.023	8	0.059	1.2E-01	2.E+01
3613	310	1397	0.070	11	0.105	7.6E-02	1.E+01
4647	415	1821	0.105	12	0.164	7.4E-02	1.E+01
5533	492	2169	0.078	9	0.148	2.3E-02	5.E+00
6057	534	2369	0.047	9	0.164	1.5E-02	4.E+00

Table 31: Refinement history for Torte4d2.

Nodes	Elements	Faces	Time for Assem	Solver Iter.	Solver Time	Error	Error·N
135	8	38	0.023	40	0.020	4.9E+06	4.E+08
233	15	74	0.023	44	0.055	2.3E+06	3.E+08
495	36	174	0.086	51	0.195	7.9E+05	2.E+08
989	78	366	0.160	57	0.477	2.3E+05	7.E+07
1947	162	742	0.293	60	1.082	7.0E+04	3.E+07
4283	372	1666	0.742	65	2.801	2.1E+04	2.E+07
8205	722	3214	1.246	34	3.059	1.3E+04	2.E+07

Table 32: Refinement history for Ct01.

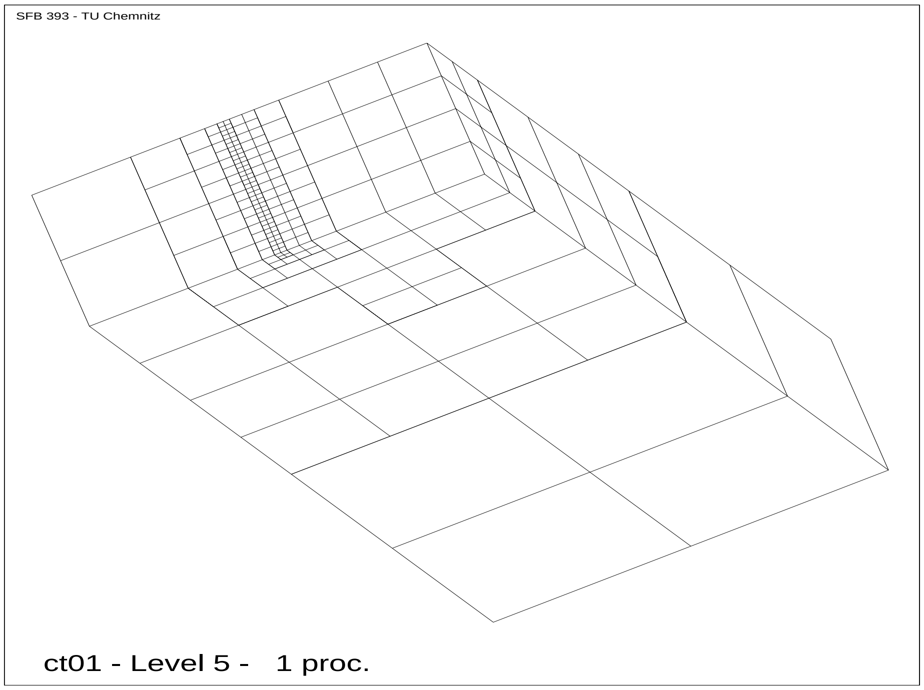


Figure 13: Ct01 after 5 refinements.

A Computing Second Derivatives $\nabla\nabla^\top u$

A.1 Notations

Coordinates in the reference element: $\hat{x} \in \hat{\square}, \quad \hat{x} = (\hat{x}_1, \hat{x}_2, \hat{x}_3)^\top$
Coordinates in the world element: $x \in \square, \quad x = (x_1, x_2, x_3)^\top$
Hexahedral elements with $n_e \in \{8, 20, 27\}$ nodes x^k $(k = 1, \ldots, n_e)$
Mappings:

$$\varphi : \hat{\square} \to \square, \quad x = \varphi(\hat{x}) = \sum_{k=1}^{n_e} \varphi^k(\hat{x})\, x^k, \quad T = \left(\frac{\partial x_i}{\partial \hat{x}_j}\right)_{i,j} \tag{A.1}$$

$$\hat{\varphi} : \square \to \hat{\square}, \quad \hat{x} = \hat{\varphi}(x) = \varphi^{-1}(x), \quad \hat{T} = T^{-1} = \left(\frac{\partial \hat{x}_i}{\partial x_j}\right)_{i,j} \tag{A.2}$$

$$\frac{\partial}{\partial \hat{x}_j} x = \sum_{k=1}^{n_e} \frac{\partial \varphi^k(\hat{x})}{\partial \hat{x}_j} x^k, \qquad \text{(column } j \text{ of } T)$$

Function(s): $u(x)$ or $[u_1(x), u_2(x), u_3(x)]^\top$,
evaluated for node points: $u_s^k = u_s(x^k),\ k = 1, \ldots, n_e$.

$$u(x) = \sum_{k=1}^{n_e} \varphi^k(\hat{x}) u^k, \text{ with } \hat{x} = \varphi^{-1}(x) \tag{A.3}$$

$$\hat{\nabla} u = \left(\frac{\partial u}{\partial \hat{x}_i}\right)_i = \left(\sum_{k=1}^{n_e} \frac{\partial \varphi^k(\hat{x})}{\partial \hat{x}_i} u^k\right)_i \tag{A.4}$$

$$\nabla u = \left(\frac{\partial u}{\partial x_j}\right)_j = \left(\sum_{i=1}^{3} \frac{\partial u}{\partial \hat{x}_i} \frac{\partial \hat{x}_i}{\partial x_j}\right)_j = T^{-\top} \hat{\nabla} u \tag{A.5}$$

$$\hat{H}(u) = \hat{\nabla}\hat{\nabla}^\top u = \left(\frac{\partial^2 u}{\partial \hat{x}_i \partial \hat{x}_j}\right)_{i,j} = \left(\sum_{k=1}^{n_e} \frac{\partial^2 \varphi^k(\hat{x})}{\partial \hat{x}_i \partial \hat{x}_j} u^k\right)_{i,j} \tag{A.6}$$

$$H(u) = \nabla\nabla^\top u = \left(\frac{\partial^2 u}{\partial x_i \partial x_j}\right)_{i,j} \qquad \ldots \qquad \text{is needed.} \tag{A.7}$$

A.2 Quantities Evaluable by Form Functions

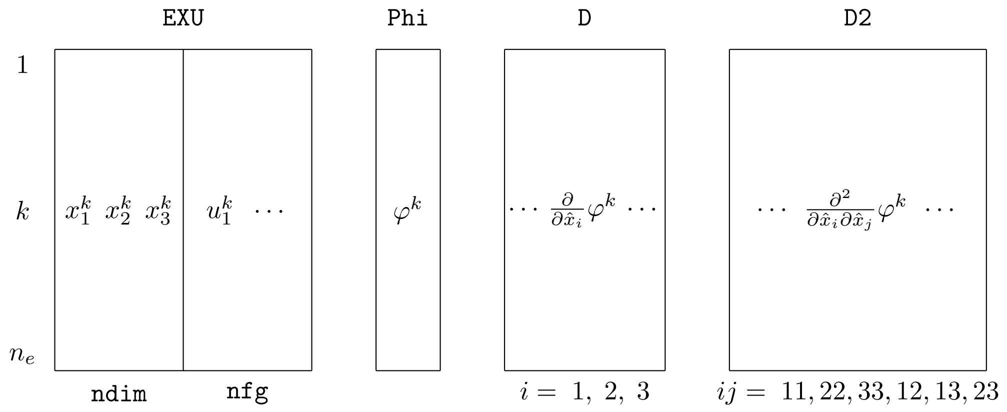
The (world) coordinates x^k of the n_e nodes and their computed function values u^k are stored as an array `EXU(ne,ndim+nfg)` (`ndim`=3, `nfg`=1 or 3).

For any point $\hat{x}$ of the reference element the subroutine `domasterval2` returns the values

$$\varphi^k(\hat{x}), \quad \frac{\partial}{\partial \hat{x}_i}\varphi^k(\hat{x}), \quad \frac{\partial^2}{\partial \hat{x}_i \partial \hat{x}_j}\varphi^k(\hat{x})$$

of the n_e ansatz functions φ^k for $k = 1, \ldots, n_e,\ \ i = 1, 2, 3,$ $ij = 11, 22, 33, 12, 13, 23.$

These values are stored in arrays `Phi(ne)`,`D(ne,3)`,`D2(ne,6)`, correspondingly.


EXU
Phi
D
D2
1
k
n_e
x_1^k x_2^k x_3^k
u_1^k $\cdots$
φ^k
$\cdots \frac{\partial}{\partial \hat{x}_i}\varphi^k \cdots$
$\cdots \frac{\partial^2}{\partial \hat{x}_i \partial \hat{x}_j}\varphi^k \cdots$
ndim
nfg
$i = 1, 2, 3$
$ij = 11, 22, 33, 12, 13, 23$

For a given point $x = \varphi(\hat{x})$ the array `xu` is determined by (A.1),(A.3):

$$\texttt{xu} = (x_1, x_2, x_3, u_1\, [, u_2, u_3]) = (\texttt{Phi})^\top(\texttt{EXU}).$$

If $\hat{x}$ is one of the node points of the reference element then `xu` is a "copy" of the corresponding row of `EXU`, since $\varphi^k(\hat{x}^j) = \delta_{kj}$.

The other values are stored analogically in arrays:

- The Jacobian matrix T and the gradient(s) $\hat{\nabla} u$ are concatenated to one array `TTU (ndim,ndim+nfg)`:

$$\texttt{TTU} = \left(T^\top \mid \hat{\nabla} u_1 \cdots \right) = (\texttt{D})^\top(\texttt{EXU})$$

- The gradient ∇u is determined by (A.5) as solution of

$$T^\top \nabla u = \hat{\nabla} u$$

using the system solver `solv33` with the system matrix `TTU(:,1:3)`$=$ $T^\top$ and the right-hand side `TTU(:,4:3+nfg)`; the solution is stored as an array `GradU(3,nfg)`.

- The second derivatives (A.6) are stored in a matrix `T2U(6,ndim+nfg)`, one column for each component of x and of u. One column contains the upper triangle of the Hessian in a sequence corresponding to that of `D2` in the figure above:

$$\mathtt{T2U} = \left(\cdots \frac{\partial^2 x_k}{\partial \hat{x}_i \partial \hat{x}_j} \cdots \left| \frac{\partial^2 u_1}{\partial \hat{x}_i \partial \hat{x}_j} \cdots \right. \right) = (\mathtt{D2})^\top (\mathtt{EXU})$$

A.3 Computing $\nabla\nabla^\top u$

First consider this matrix by elements: $(\nabla\nabla^\top u)_{ij} = \dfrac{\partial^2 u}{\partial x_i \partial x_j}$.

$$\begin{aligned} \frac{\partial}{\partial x_i}\left(\frac{\partial u}{\partial x_j}\right) &\overset{(A.5)}{=} \sum_p \frac{\partial}{\partial x_i}\left(\frac{\partial u}{\partial \hat{x}_p}\right) \cdot \frac{\partial \hat{x}_p}{\partial x_j} + \sum_p \frac{\partial u}{\partial \hat{x}_p} \cdot \frac{\partial}{\partial x_i}\left(\frac{\partial \hat{x}_p}{\partial x_j}\right) \\ &= \sum_{p.q} \frac{\partial \hat{x}_p}{\partial x_j} \frac{\partial^2 u}{\partial \hat{x}_p \partial \hat{x}_q} \frac{\partial \hat{x}_q}{\partial x_i} + \sum_p \frac{\partial u}{\partial \hat{x}_p} \cdot \frac{\partial^2 \hat{x}_p}{\partial x_i \partial x_j} \end{aligned} \qquad (A.8)$$

and now in matrix notation:

$$H(u) = \nabla\nabla^\top u = T^{-\top} \hat{H}(u) T^{-1} + \sum_p (\hat{\nabla} u)_p H(\hat{x}_p) \qquad (A.9)$$

$$\text{with } H(\cdot) = \left(\frac{\partial^2 \cdot}{\partial x_i \partial x_j}\right)_{ij}, \qquad \hat{H}(\cdot) = \left(\frac{\partial^2 \cdot}{\partial \hat{x}_i \partial \hat{x}_j}\right)_{ij}$$

Computing $H(\hat{x}_p)$

$$\left(\frac{\partial^2 \hat{x}_p}{\partial x_i \partial x_j}\right)_{i,j,p} = \left(\frac{\partial}{\partial x_i}\left(\frac{\partial \hat{x}_p}{\partial x_j}\right)_{p,j}\right)_i = \left(\frac{\partial}{\partial x_i}\hat{T}\right)_i = \left(\frac{\partial}{\partial x_i} T^{-1}\right)_i \qquad (A.10)$$

Thus, consider

$$0 = \frac{\partial}{\partial x_i}\left(T \cdot T^{-1}\right) = \left(\frac{\partial}{\partial x_i} T\right) \cdot T^{-1} + T \cdot \left(\frac{\partial}{\partial x_i} T^{-1}\right)$$

$$\frac{\partial}{\partial x_i} T^{-1} = -T^{-1}\left(\frac{\partial}{\partial x_i} T\right) T^{-1} \tag{A.11}$$

$$\begin{aligned}\frac{\partial}{\partial x_i} T = \left(\frac{\partial}{\partial x_i}\frac{\partial x_j}{\partial \hat{x}_k}\right)_{j,k} &= \left(\sum_r \frac{\partial^2 x_j}{\partial \hat{x}_r \partial \hat{x}_k}\frac{\partial \hat{x}_r}{\partial x_i}\right)_{j,k} \\ &= \left(\sum_r \left(T^{-1}\right)_{ri}\left(\hat{H}(x_j)\right)_{r,*}\right)_{j,k}\end{aligned} \tag{A.12}$$

Now consider (A.10) as multiple sum using (A.11) and (A.12), where T^{-1} and $\hat{H}(x_j)$ are used according to (A.2) and (A.6) respectively:

$$\begin{aligned}H(\hat{x}_p) &= \left(\frac{\partial^2 \hat{x}_p}{\partial x_i \partial x_j}\right)_{i,j} = \left(-\sum_{r,k,m}\frac{\partial \hat{x}_r}{\partial x_i}\frac{\partial \hat{x}_k}{\partial x_j}\frac{\partial \hat{x}_p}{\partial x_m}\frac{\partial^2 x_m}{\partial \hat{x}_r \partial \hat{x}_k}\right)_{i,j} \\ &= -\sum_m \frac{\partial \hat{x}_p}{\partial x_m} \cdot T^{-\top}\hat{H}(x_m)T^{-1} \\ &= -T^{-\top}\left(\sum_m \frac{\partial \hat{x}_p}{\partial x_m}\hat{H}(x_m)\right)T^{-1} \\ \sum_p (\hat{\nabla} u)_p H(\hat{x}_p) &= -T^{-\top}\left(\sum_{m,p}\frac{\partial u}{\partial \hat{x}_p}\frac{\partial \hat{x}_p}{\partial x_m}\hat{H}(x_m)\right)T^{-1} \\ &= -T^{-\top}\left(\sum_m \frac{\partial u}{\partial x_m}\hat{H}(x_m)\right)T^{-1}\end{aligned}$$

The sum over m can be implemented in a quiet simple way, if the matrices of second derivatives $\hat{H}(x_m)$ are represented as columns m ($m = 1, 2, 3$) of a matrix `HX(6,3)`. The column length 6 is caused by the symmetry of the

3×3-Hessian $\hat{H}(x_m)$. Hence, the sum over m is computed as $\texttt{HX} \cdot \nabla u$.

$$\begin{array}{cc|c|c|c} \multicolumn{2}{c|}{\texttt{HX}} & \partial^2 x_1 & \partial^2 x_2 & \partial^2 x_3 \\ \hline \partial\hat{x}_1 & \partial\hat{x}_1 & & & \\ \partial\hat{x}_2 & \partial\hat{x}_2 & & & \\ \partial\hat{x}_3 & \partial\hat{x}_3 & & & \\ \hline \partial\hat{x}_1 & \partial\hat{x}_2 & & & \\ \partial\hat{x}_1 & \partial\hat{x}_3 & & & \\ \partial\hat{x}_2 & \partial\hat{x}_3 & & & \\ \hline \end{array} \cdot \begin{bmatrix} \dfrac{\partial u}{\partial x_1} \\ \dfrac{\partial u}{\partial x_2} \\ \dfrac{\partial u}{\partial x_3} \end{bmatrix} = \begin{bmatrix} \\ \\ \sum\limits_m \\ \\ \\ \end{bmatrix}$$

Thus, each Hessian $\hat{H}(x_m)$ is "vectorized" as column

$$\underline{\texttt{svec}}\left(\hat{H}(x_m)\right) = (h_{11}, h_{22}, h_{33}, h_{12}, h_{13}, h_{23})^\top$$

of a 6×3-matrix $\hat{\mathcal{H}}(x)$:

$$\hat{\mathcal{H}}(x) = \left[\ \underline{\texttt{svec}}\left(\hat{H}(x_1)\right) \middle| \ \underline{\texttt{svec}}\left(\hat{H}(x_2)\right) \middle| \ \underline{\texttt{svec}}\left(\hat{H}(x_3)\right)\ \right],$$

and similar for $\mathcal{H}(\hat{x})$. Correspondingly, $\hat{\mathcal{H}}(u) = \underline{\texttt{svec}}\left(\hat{H}(u)\right)$ is stored as one such column yielding the following representation:

$$\sum_p (\hat{\nabla} u)_p \, \underline{\texttt{svec}}\left(H(\hat{x}_p)\right) = \mathcal{H}(\hat{x})\hat{\nabla} u = \underline{\texttt{svec}}\left(-T^{-\top}\underline{\texttt{mat}}\left(\hat{\mathcal{H}}(x)\nabla u\right) T^{-1}\right),$$

and finally:

$$\boxed{H(u) = T^{-\top} \cdot \underline{\texttt{mat}}\left(\hat{\mathcal{H}}(u) - \hat{\mathcal{H}}(x)\nabla u\right) \cdot T^{-1}} \quad \text{where} \quad \underline{\texttt{mat}}\left(\underline{\texttt{svec}}\, A\right) = A\,.$$

Instead of this relative complex operation we should better find an operation that is directly applied to vectorized storage scheme $\hat{\mathcal{H}}$ of the symmetric matrices providing $H(u)$ in vectorized form again:

$$\mathcal{H}(u) = F \cdot \left(\hat{\mathcal{H}}(u) - \hat{\mathcal{H}}(x)\nabla u\right) = \underline{\texttt{svec}}\left(H(u)\right), \qquad \text{with } F \in \mathbb{R}^{6,6}$$

Hence, the action $\underline{\texttt{svec}}\left(T^{-\top} \cdot (\ A\ ) \cdot T^{-1}\right)$ should be performed by $F \cdot \underline{\texttt{svec}}(A)$. The matrix F realizing this action consists of four 3×3 blocks, which are computed in the following way:

```
C variables:
C    T - holds the inverse of the Jacobian matrix
C    F - stores the transpose of the matrix F mentioned above
      DATA i1 /1,3,2/, i2 /2,1,3/
      DO i=1,3
        DO j=1,3      ! entries to blocks of F:
C                     upper left
          F(j,i)    = T(j,i)**2
C                     upper right
          F(j+3,i)  = 2d0*T(i1(j),i)*T(i2(j),i)
C                     lower left
          F(j,i+3)  = T(j,i1(i))*T(j,i2(i))
C                     lower right
          F(j+3,i+3)= T(i1(j),i2(i))*T(i2(j),i1(i))
     +              + T(i2(j),i2(i))*T(i1(j),i1(i))
        ENDDO
      ENDDO
```

References

[1] Th. Apel, F. Milde, and U. Reichel. Spc-pm po 3d v4.0 - programmer's manual (part ii). Technical Report SFB393 99-37, Technische Universität Chemnitz, December 1999.

[2] Th. Apel, F. Milde, and M. Theß. Spc-pm po 3d - programmer's manual. Technical Report SPC95-34, Technische Universität Chemnitz-Zwickau, December 1995.

[3] Th. Apel and U. Reichel. Spc-pm po 3d v3.3 - user's manual. Technical Report SFB393 99-06, Technische Universität Chemnitz, February 1999.

[4] J. Bramble, J. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55(191):1–22, 1991.

[5] G. Kunert. A posteriori error estimation for anisotropic tetrahedral and triangular finite element meshes. Phd thesis, Technische Universität Chemnitz, 1999.

[6] D. Lohse. Ein standard-file für $3d$-gebietsbeschreibungen. - definition des fileformats v 2.1. Technical Report SFB393 98-17, Technische Universität Chemnitz, April 1998.

[7] M. Pester. Behandlung gekrümmter oberflächen in einem 3d-fem-programm für parallelrechner. Technical Report SFB393 97-10, Technische Universität Chemnitz, April 1997.

[8] H. Yserentant. On the multi-level-splitting of the finite element spaces. *Numer. Math.*, 49:379–412, 1986.

00-14 C. Bourgeois, R. Schneider. Biorthogonal wavelets for the direct integral formulation of the heat equation. March 2000.

00-15 A. Rathsfeld, R. Schneider. On a quadrature algorithm for the piecewise linear collocation applied to boundary integral equations. March 2000.

00-16 S. Meinel. Untersuchungen zu Druckiterationsverfahren für dichteveränderliche Strömungen mit niedriger Machzahl. März 2000.

00-17 M. Konstantinov, V. Mehrmann, P. Petkov. On Fractional Exponents in Perturbed Matrix Spectra of Defective Matrices. April 2000.

00-18 J. Xue. On the blockwise perturbation of nearly uncoupled Markov chains. April 2000.

00-19 N. Arada, J.-P. Raymond, F. Tröltzsch. On an Augmented Lagrangian SQP Method for a Class of Optimal Control Problems in Banach Spaces. April 2000.

00-20 H. Harbrecht, R. Schneider. Wavelet Galerkin Schemes for 2D-BEM. April 2000.

00-21 V. Uski, B. Mehlig, R. A. Römer, M. Schreiber. An exact-diagonalization study of rare events in disordered conductors. April 2000.

00-22 V. Uski, B. Mehlig, R. A. Römer, M. Schreiber. Numerical study of eigenvector statistics for random banded matrices. May 2000.

00-23 R. A. Römer, M. Raikh. Aharonov-Bohm oscillations in the exciton luminescence from a semiconductor nanoring. May 2000.

00-24 R. A. Römer, P. Ziesche. Hellmann-Feynman theorem and fluctuation-correlation analysis of i the Calogero-Sutherland model. May 2000.

00-25 S. Beuchler. A preconditioner for solving the inner problem of the p-version of the FEM. May 2000.

00-26 C. Villagonzalo, R.A. Römer, M. Schreiber, A. MacKinnon. Behavior of the thermopower in amorphous materials at the metal-insulator transition. June 2000.

00-27 C. Mehl, V. Mehrmann, H. Xu. Canonical forms for doubly structured matrices and pencils. June 2000. S. I. Solov'ev. Preconditioned gradient iterative methods for nonlinear eigenvalue problems. June 2000.