Technische Universität Chemnitz-Zwickau

DFG-Forschergruppe "SPC" · Fakultät für Mathematik

Thomas Apel Frank Milde Michael Theß

SPC-PM Po 3D

—

Programmer's Manual

Acknowledgement. The package *SPC-PM Po 3D* has been developed in the research group SPC at the *Fakultät für Mathematik* of the *Technische Universität Chemnitz–Zwickau* under the supervision of A. Meyer and Th. Apel. Other main contributors are G. Globisch, D. Lohse, M. Meyer, F. Milde, M. Pester, and M. Theß.

Chapter 6 of this manual was written together with M. Pester. The authors thank M. Jung and A. Meyer for their proof-reading and helpful comments. U. Reichel typed the manuscript.

The research group SPC is supported by *Deutsche Forschungsgemeinschaft* (German Research Foundation), No. La 767/3.

All this collaboration and support is gratefully acknowledged.

Preprint-Reihe der Chemnitzer DFG-Forschergruppe
"Scientific Parallel Computing"

SPC 95_34 **Dezember 1995**

Authors' addresses:

Thomas Apel
Technische Universität Chemnitz-Zwickau
Fakultät für Mathematik
D-09107 Chemnitz
Germany

email: apel@mathematik.tu-chemnitz.de

Frank Milde
Technische Universität Chemnitz-Zwickau
Fakultät für Naturwissenschaften, Institut für Physik
D-09107 Chemnitz
Germany

email: milde@physik.tu-chemnitz.de

Michael Theß
Technische Universität Chemnitz-Zwickau
Fakultät für Mathematik
D-09107 Chemnitz
Germany

email: thess@mathematik.tu-chemnitz.de

Contents

Chapter 1

Overview

1.1 Introduction

SPC-PMPo 3D is a computer program to solve the Poisson equation or the Lamé system of linear elasticity over a three-dimensional domain on a MIMD parallel computer. It is being developed in the research group SPC (Scientific Parallel Computing) at the *Fakultät für Mathematik* of the *Technische Universität Chemnitz-Zwickau* under the supervision of Prof. A. Meyer, and Dr. Th. Apel. Other main contributors are Dr. M. Meyer, F. Milde, Dr. M. Pester, and M. Theß.

The historical roots of the program are at one hand in several parallel programs for solving problems over twodimensional 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 developed 1987–89 a sequential program for the solution of the Poisson equation over three-dimensional domains which was extended 1993–94 together with F. Milde.

For an introduction of the capabilities of the program, its installation and utilization we refer to the User's Manual [3]. The aim of this Programmer's Manual is to provide a description of the algorithms and their realization. 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 data structure. The main part of this documentation consists of the description of the specific libraries for *SPC-PMPo 3D*: *libNA.a*, *libNT.a*, and *libNQ.a* for hierarchical mesh refinement of tetrahedral and cuboidal (hexahedral) meshes (Chapter 3), *libA.a* for assembly of the equation system and for error assessing (Chapter 4), and *libS.a* for solving this system with a preconditioned CG algorithm (Chapter 5).

Chapter 6 provides a short description of some general libraries that are used also in other program systems developed by the research group SPC and which are described in more detail elsewhere [4, 9, 13]: *libKLZ.a*, *libvbasmod.a*, *libMbasmod.a* and *libDDCMcom.a* for basic matrix and vector operations, *libCubecom.a* for basic communication routines, *libGraf.a* for graphical representation, as well as *libTools.a* with some auxiliary routines.

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 user interface, and `typewriter style` for the names of variables.

1.2 The boundary value problems

Consider the Poisson problem in the notation

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega \subset \mathbb{R}^3, \\ u &= u_0 \quad \text{on} \quad \partial\Omega_1, \\ \frac{\partial u}{\partial n} &= g \quad \text{on} \quad \partial\Omega_2, \\ \frac{\partial u}{\partial n} &= 0 \quad \text{on} \quad \partial\Omega \setminus \partial\Omega_1 \setminus \partial\Omega_2, \end{aligned}$$

or the Lamé problem for $\underline{u} = (u^{(1)}, u^{(2)}, u^{(3)})^T$

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

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

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

$\underline{n}$ is the outward normal, and δ_{ij} is the Kronecker delta. The domain $\Omega \subset \mathbb{R}^3$ must be bounded. In the present version curved boundaries can not be treated by the refinement procedure, thus Ω is restricted to be a polyhedron.

The boundary value problem is solved by a standard finite element method, using either tetrahedral or brick elements with linear or quadratic shape functions of the serendipity class, see Figure 1.1.

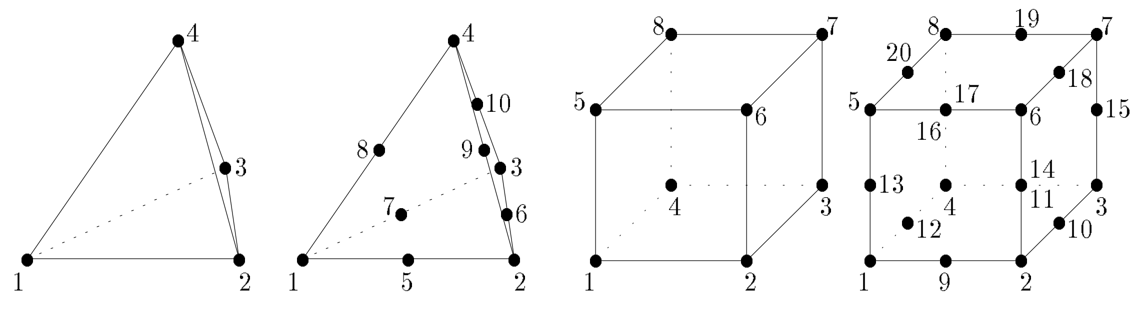

Figure 1.1: Finite elements implemented in *SPC-PM Po 3D*.

Chapter 2

Data structure

2.1 General remarks

The program is working in the SPMD mode, that means single program multiple data. Consequently, all data described are local data, possibly with different length on every processor. The connection between these local data is coded in the arrays `IGLOB`, `KETTE1D`, and `KETTE2D`, see Subsections 2.2.5 and 2.2.7; this information is sufficient for the communication (finite element accumulation).

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 start. We have our own memory management and must take care of calculating these pointers to avoid overlaps.

In our FEM program we have two data structures. First, the mesh is refined with the full data structure (FDS, see 2.2) because of its greater variability. After the change to the more typical reduced data structure (RDS, see 2.3) the assembly and solution of the equation system are performed.

There are a few general variables:

`NDF`	number of degrees of freedom per node,
`NEN2D`	number of nodes per face,
`NEN3D`	number of nodes per volume.

In 2.2 and 2.3 we describe the arrays in the following general form:

1. general description of the array,
2. name and dimension 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 positions of data. Most of the dimensions of the arrays are also variables/parameters which are located in COMMON blocks in the source file *net3ddat.inc*. It is better to use these variables instead of hard numbers because of possible evolution of the data structure.

2.2 Full data structure (FDS)

In the FDS volumes are represented by a number of faces, faces by a number of edges and edges by a number of nodes.

All arrays (except the coordinate array and the kette data) have the same coarse structure representing the hierarchical character of the refinement. The data of all levels of refinement including the coarse mesh are stored in the following way:

| coarse mesh data | level 1 | level 2 | ··· | level N |

For each array there is a variable which points to the first entry of level `N`. Its name is in general of the form `F`+name of the array (for instance: `FVOL` is the first entry of level `N` in `VOL`).

2.2.1 Volumes

1. Each volume is described by its [4|6] faces (4 for a tetrahedron / 6 for a brick).

2. `VOL(DIMVOL,*)` : `DIMVOL`=[4|6]

3. $\overbrace{|\ \text{face\_1}\ |\ \text{face\_2}\ |\ \cdots\ |}^{[4|6]\ \text{faces}}$

   Face_i is a face number.

2.2.2 Faces

1. Each face is described by its [3|4] edges (3 for a triangle / 4 for a quadrilateral).

2. `FACE(DIMFACE,*)` : `DIMFACE` = [3|4] + 2

3. $\overbrace{|\ \text{edge\_1}\ |\ \text{edge\_2}\ |\ \cdots\ |}^{[3|4]\ \text{edges}}$ type | number_of_first_son |

   Edge_i is a edge number

4. Until now there is only one type (1) which means plane face. All sons are numbered consecutively. Up to now the number of sons is always 4.

   The faces of each level are sorted in the following way:

   $$\underbrace{\cdots}_{\text{level } i-1} \quad \underbrace{|\ \text{coupling faces}\ |\ \text{other faces}\ |}_{\text{level } i} \quad \underbrace{\cdots}_{\text{level } i+1}$$

   By our definition, all faces of the coarse mesh and their sons are coupling faces, even if they are not contained in inter-processor boundaries.

   It is recommended to use the following pointer variables for this dataset:

`FZEIG`	position of the type in `FACE`	currently [3|4] + 1
`FCHIELD`	position of the number_of_first_son	currently [3|4] + 2

2.2.3 Edges

1. Each edge is described by its 2 vertices and the middle node (only quadratic case).
2. `KANTE(DIMKANTE,*)` : `DIMKANTE` $= 5$
3. | vertex_1 | vertex_2 | middle node | type | number_of_first_son |
   Vertex_i is a vertex number.
4. Until now there are two possible values of type (1 and -1). The type -1 is used for dummy edges which are necessary to generate the hierarchical list but no face points to it.

   The coupling edges are located at the beginning of each level followed by the other edges of coupling faces.

   $$\underbrace{\cdots}_{\text{level } i-1} \quad \underbrace{\Big|\ \text{coupling edges}\ \Big|\ \text{edges on coupling faces}\ \Big|\ \text{other edges}\ \Big|}_{\text{level } i} \quad \underbrace{\cdots}_{\text{level } i+1}$$

   By our definition, all edges of the coarse mesh and their sons are coupling edges , even if they are not contained in inter-processor boundaries.

   It is recommended to use the following pointer variables for this dataset:

`KZEIG`	position of the type	currently 4
`KCHIELD`	position of the number_of_first_son	currently 5

2.2.4 Coordinates of the nodes

1. Each node is represented by its three Euclidean coordinates.
2. `COOR(3,*)`
3. $|\ X_i\ |\ Y_i\ |\ Z_i\ |$
4. The nodes are placed in `COOR` in the following way:

   $$\Big|\ \text{cross points}\ \underbrace{\Big|\ CE_1\ \Big|\ CE_2\ \Big|\ CE_3\ \Big|\ \cdots\ \Big|}_{\text{1D - kettes}}\underbrace{\ CF_1\ \Big|\ CF_2\ \Big|\ CF_3\ \Big|\ \cdots\ \Big|}_{\text{2D - kettes}}\ \text{inner nodes}\ \Big|$$

   Each 1D kette (CE_i) is a block of nodes which belong to (sons of) a (coupling) edge of the coarse mesh. By analogy, each 2D kette is a block of interior nodes of a (coupling) face of the coarse mesh.

   The structure of these kettes is more complicated. It is shown in all details with an example in 3.3.2.

2.2.5 Global Crosspoint Names `IGLOB`

1. To identify the local crosspoints their global name is stored.
2. `IGLOB(*)`
3. `IGLOB(I)` = global name of the node `I` (which is an crosspoint), where `I` is the local number.

2.2.6 Dirichlet/Neumann data

1. The Dirichlet/Neumann data are associated with faces. They have both the same data structure.

2. `DIR(DVDIR,*)` : `DVDIR` $= 1 + \mathtt{NDF} * (1 + \mathtt{NDIRREAL})$
   `NEUM(DVNEUM,*)` : `DVNEUM` $= 1 + \mathtt{NDF} * (1 + \mathtt{NNEUMREAL})$

3. $\mid \text{number\_of\_face} \mid \overbrace{\text{type, data (DF\_1)} \mid \text{type, data (DF\_2)} \mid \cdots}^{\mathtt{NDF}\text{ data blocks}} \mid$

   DF_k means the k-th degree of freedom.

4. The data are `NDIRREAL`/ `NNEUMREAL` $= 4/5$ real parameters (RP) describing the boundary condition.

   Possible values of the type of the boundary condition data are :

0	none	
1	constant	$f = RP(1)$
2	linear function	$f = RP(1) * X + RP(2) * Y + RP(3) * Z + RP(4)$
100	function call	$f = u(X, Y, Z)$ (from *./Assem/bsp.f*)

   $RP(5)$ has been planned for the coefficient in boundary conditions of 3$^{\text{rd}}$ kind, but this is not implemented yet.

2.2.7 Kette data

1. The purpose of the kette data is the optimization of the communication. Every coupling face/edge of the coarse mesh is referred in the kette data by its global names of vertices. All interior nodes of these faces/edges have consecutive numbers and form a so called kette, see 2.2.4. Thus they can be described by a pointer to the first node and the number of nodes (length) in this block.

   There are two different kette data (`KETTE1D` for edges and `KETTE2D` for faces) which have the same data structure. For more information see [4].

2. `KETTE1D(K1DDIM,*)/KETTE2D(K2DDIM,*)` : `K1DDIM` = `K2DDIM` = 7

3. $\mid \text{pointer} \mid \text{length} \mid \text{pathID} \mid \overbrace{\text{vertex\_1} \mid \text{vertex\_2} \mid \cdots}^{2\text{ or }[3|4]\text{ vertices}} \mid$

   Vertex_i is a vertex number.

4. Note that the vertex numbers here are global (crosspoint) names. For an explanation of pathID see [4].

   It is recommended to use the following pointer variables for this dataset:

`PKZEIG`	position of the pointer	currently 1
`PKLENG`	position of the length	currently 2
`PWEGID`	position of the pathID	currently 3
`PKDAT`	position of the data (first node)	currently 4

2.2.8 CHAIN

1. The array CHAIN is very similar to the KETTE2D data. The difference is that it does not contain pointers to node numbers but to the numbers of subfaces. Note that all subfaces which form a coupling face have consecutive face numbers.

2. CHAIN(K2DDIM,$*$) : K2DDIM $= 7$

3. $| \text{pointer} \mid \text{length} \mid \text{pathID} \overbrace{\mid \text{vertex\_1} \mid \text{vertex\_2} \mid \cdots \mid}^{[3|4]\ \text{vertices}}$
   Vertex_i is a vertex number.

4. Note that the vertex numbers here are global (crosspoint) names.

   There are the same pointers PKZEIG, PKLENG, PWEGID, and PKDAT as for the kette data.

2.2.9 REGION

1. The vector REGION contains an integer for each volume. This integer is read from the input file and passed on to the son elements during the refinement process. Currently it is used as a material index.

2. REGION($*$)

3. |number|

2.3 Reduced data structure (RDS)

The RDS is more typically for finite element applications. The basic entities are the nodes and all other objects (volumes, boundary faces ...) are represented by numbers of nodes. The arrays of faces and edges disappear.

2.3.1 Volumes

1. Each volume consists of NEN3D nodes.

2. VOL(NEN3D,$*$) :

3. $\overbrace{\mid \text{node\_1} \mid \text{node\_2} \mid \cdots \mid}^{\texttt{NEN3D}\ \text{nodes}}$

2.3.2 Dirichlet data

1. Dirichlet data are given on faces. A face consists of NEN2D nodes and thus the Dirichlet data consists of the node numbers and the Dirichlet values of every degree of freedom at these nodes. The face number (FDS) is an additional information for the error estimator.

2. DIR(DRDIR,$*$) : DRDIR $= 2 +$ NEN2D $*$ (NDF $+ 1$)

3. $|\text{number\_of\_face}|\overbrace{\text{node\_1}|\text{node\_2}|\cdots}^{\texttt{NEN2D}\text{ nodes}}|\texttt{IFG}|\overbrace{\text{Dir. values (DF\_1)}|\text{Dir. values (DF\_2)}|\cdots}^{\texttt{NDF}\text{ data blocks of length }\texttt{NEN2D}}|$

4. In the case of more than one degree of freedom it may happen that not all degrees of freedom have a Dirichlet condition. The Information about this is coded bitwise in the entry `IFG`.

   It is recommended to use the following pointer variables for this dataset:

`DRNODES`	position of the nodes
`DRIFG`	position of `IFG`
`DRDAT`	position of the data

2.3.3 Neumann data

1. The Neumann data are used in a different way then the Dirichlet data, so the only change from the full to the reduced structure is to add the node numbers which represent the face. The face number (FDS) is again used for error estimation.

2. $\texttt{NEUM(DRNEUM,}*\texttt{)} : \texttt{DRNEUM} = 1 + \texttt{NEN2D} + \texttt{NDF} * (\texttt{NNEUMREAL} + 1)$

3. $|\text{ number\_of\_face }|\overbrace{\text{ node\_1 }|\text{ node\_2 }|\cdots}^{\texttt{NEN2D}\text{ nodes}}|\overbrace{\text{type, data (DF\_1)}|\text{type, data (DF\_2)}|\cdots}^{\texttt{NDF}\text{ data blocks}}|$
   DF_k means the k-th degree of freedom.

4. There are two pointer variables for this dataset:

`NRNODES`	position of the nodes
`NRDAT`	position of the data

2.3.4 Hierarchical List

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

2. `LC`(4,*)

3. | node | father_1 | father_2 | factor |

4. The factor ($0 <$ factor < 1) describes the relative position of the node at the edge:

$$\texttt{COOR}(\text{node}) = \text{factor} * \texttt{COOR}(\text{father\_1}) + (1 - \text{factor}) * \texttt{COOR}(\text{father\_2})$$

   The entries in `LC` are ordered such that the fathers are included before their sons. Note that the entries in `COOR` are ordered in another way (index equals number of the node, for the numbering of the nodes see Section 3.3). Note that father_1 = father_2 = 0 if the node is a crosspoint.

2.3.5 TETF

`TETF` is the old array of volumes `VOL` from the FDS (see 2.2.1). It is used only in the error estimator (Chapter 4).

2.3.6 Node Coordinates/`IGLOB`/Kette Data/`CHAIN`/`REGION`

The data structure is the same as in the FDS, see 2.2.

2.4 INCLUDE-Files/COMMON-Blocks

There is a number of COMMON-Blocks in our program. Most of them are located in INCLUDE-Files. Moreover, some parameters are determined in these files.

2.4.1 *net3ddat.inc*

This INCLUDE-File contains a number of variables/parameters which determines dimensions of data, especially these which depend on the type of the mesh. All variables are in these COMMON-Blocks:

- `/NENXD/`

`NEN2D`	number of nodes per face (see 2.1)
`NEN3D`	number of nodes per volume (see 2.1)

- `/NETDIM/`

`DIMVOL`	dimension of the array of volumes (FDS) (see 2.2.1)
`DIMFACE`	dimension of the array of face (FDS) (see 2.2.2)
`FCHIELD`	pointer to the number of the first subface (see 2.2.2)
`FZEIG`	pointer to the type of the face (see 2.2.2)
`SUB`	name of the subdirectory with the meshes

- `/RB/`

`NDF`	number of degrees of freedom (see 2.1)
`DVDIR`	dimension of the array of Dirichlet data (FDS) (see 2.2.6)
`DRDIR`	dimension of the array of Dirichlet data (RDS) (see 2.3.2)
`DRNODES`	position of the nodes (RDS) (see 2.3.2)
`DRIFG`	position of `IFG` (RDS) (see 2.3.2)
`DRDAT`	position of the data (RDS) (see 2.3.2)
`DVNEUM`	dimension of the array of Neumann data (FDS) (see 2.2.6)
`DRNEUM`	dimension of the array of Neumann data (RDS) (see 2.3.3)
`NRNODES`	position of the nodes (RDS) (see 2.3.3)
`NRDAT`	position of the data (RDS) (see 2.3.3)

  The subroutine `SET_RBCOM` sets all these Variables. (`NDF`, `NEN2D` and `NEN3D` must be correct when calling this routine.)

Moreover, there are the following parameters (via FORTRAN77 parameter statement):

`DIMKANTE`	dimension of the array of edges	currently 5
`KZEIG`	position of the type of the edge	currently 4
`KCHIELD`	position of the child of the edge	currently 5
`K1DDIM`	dimension of the array of 1D kettes	currently 7
`K2DDIM`	dimension of the array of 2D kettes	currently 7
`PKZEIG`	position of the pointer in the kette data	currently 1
`PKLENG`	position of the block length in the kette data	currently 2
`PWEGID`	position of the path identifier in the kette data	currently 3
`PKDAT`	position of the data in the kette data	currently 4
`NDIRREAL`	number of Dirichlet real parameters	currently 4
`NNEUMREAL`	number of Neumann real parameters	currently 5

2.4.2 *com_prob.inc*

There is a number of variables with information concerning the mesh.

- `/PROBLEM/`

`Nk`	number of nodes (local on the processor)
`NCrossG`	number of crosspoints (global)
`NCrossL`	number of crosspoints (local)
`NKettSum`	number of all coupling nodes (local)
`NC`	`NKettSum` + `NCrossL`
`NI`	number of interior nodes (local)
`NanzK1D`	number of 1D kettes
`NanzK2D`	number of 2D kettes
`NanzK`	`NanzK1D` + `NanzK2D`
`LinkLevel`	auxiliary variable for communication

The subroutine `COM_PROB` sets most of these variables.

2.4.3 *filename.inc*

- `/FILENAME/`

`File`	name of the standard file (without *.std*)
`Length`	length of `File`
`Nlevl`	not used
`itri`	not used
`Lunit`	not used
`Fullname`	name of the standard file including subdirectory with the meshes and *.std*

To get the filename and to set these variables the subroutine `SETFILE` is used.

2.4.4 *standard.inc*

This INCLUDE-File contains some program control variables which can be changed with the files *control.quad* or *control.tet* without compiling the program anew, see [3, Section 2.3].

- `/standard/`

`vertvar`	kind of coarse grid partitioning
`femakkvar`	variant of accumulation of distributed data, see [4]
`loesvar`	choice of the preconditioner
`Nint2ass`	number of the quadrature formula used for assembling Neumann boundary data
`Nint3ass`	number of quadrature formula for 3D elements used in the assembling
`Nint2error`	as `nint2ass`, but used in the error estimator for the integration of the jump of the normal derivatives
`Nint3error`	as `nint3ass`, but used for the integration of 3D integrals in the error calculation
`Epsilon`	stop criterion for the CG (relative decrease of the norm of the residual)
`Iter`	maximal number of iterations in the CG algorithm
`NDiag`	upper estimate for the number of nonzero entries in any row of the stiffness matrix
`Verf`	mesh refinement parameter for a certain class of examples, see [3, Subsection 4.1.7]
`lin_quad`	kind of shape functions

2.4.5 *trnet.inc*

There are some variables with information concerning the parallel computer, compare [9, Section 3.1].

- `/TrNet/`

`NCUBE`	dimension of the hypercube
`ICH`	number of the processor in the hypercube topology
`NODENR`	number of the node the processor is located on

- `/TrRing/`

`NPROC`	number of processors
`ICHRING`	number of the processor in ring topology
`Lforw`	number of the link that leads to the successor within the ring
`Lback`	number of the link that leads to the predecessor within the ring

Chapter 3

Hierarchical Mesh Refinement

3.1 Parameters of NETMAKE

The procedure NETMAKE generates the mesh. It reads the coarse mesh from a file, distributes it to the processors, performs the refinement with the full data structure and reduces the data to the reduced data structure.

```
SUBROUTINE NETMAKE(A,JCOOR,NUMNP,JDIR,NDIR,JNEUM,NNEUM,JVOL,
                   JREGION,NUMEL,JIGLOB,JKETTE1D,JKETTE2D,
                   JCHAIN,JFREI,LAENGE,IER,VFS,JTETF,JLC,STEUER)
```

A	I/O	workspace vector
JCOOR	O	pointer to array of node coordinates COOR
NUMNP	O	NUMber of Nodal Points
JDIR	O	pointer to the Dirichlet data DIR
NDIR	O	number of Dirichlet faces
JNEUM	O	pointer to the Neumann data NEUM
NNEUM	O	number of Neumann faces
JVOL	O	pointer to array of volumes VOL
JREGION	O	pointer to the region data REGION
NUMEL	O	NUMber of of ELements (volumes)
JIGLOB	O	pointer to array of global crosspoint names IGLOB
JKETTE1D	O	pointer to array of 1D kette data KETTE1D
JKETTE2D	O	pointer to array of 2D kette data KETTE2D
JCHAIN	O	pointer to array CHAIN
JFREI	I/O	first unused position of A
LAENGE	I	length of A
IER	O	error parameter
VFS	O	number of refinement steps
JTETF	O	pointer to the array of volumes of the full data structure
JLC	O	pointer to the hierarchical list
STEUER	O	array of 10 logical status parameters (used in the output tool, see 3.5)

To optimize the communication the nodes which belong to coupling faces/edges stand together in the array of node coordinates and the order of these points is the same on every processor. They form a so called kette. To realize this it is useful to order the edges and faces too.

3.2 Coarse mesh input and distribution

The coarse mesh will be read from a standardized file, compare [3, Section 3.2]. These files are located in the subdirectories *./mesh3* (tetrahedral meshes) or *./mesh4* (brick meshes). Only processor zero reads the mesh and later it is given to all other processors. There are two methods to determine the owner of the volumes.

The first method is very simple. Each processor should own the same number of volumes and they are distributed simply as they are numbered ("linear distribution"). The next step is to delete all volumes which are not owned by the processor and all other unused data.

The second method is much more complicated and should optimize the later computation. This method is known as Recursive Spectral Bisection (RSB) and is based on results of graph theory. It operates on the dual graph to the coarse mesh.

Spectral bisection calculates the eigenvector corresponding to the second-smallest eigenvalue of the dual graph's Laplacian matrix. To each node in the dual graph naturally corresponds a component of the eigenvector. To partition the dual graph into two nearly equal-sized subgraphs 0 and 1, the algorithm computes the median value of the eigenvector components. Those nodes with eigenvector components smaller than the median value are assigned to subgraph 0, the remaining nodes to subgraph 1.

The routine runs parallel on all processors where the elements corresponding to subgraph 0 remain at the first half of processors and the others at the second half. The not assigned data is deleted on each half. This is recursively applied to the mesh according to the dimension of the hypercube of the processors used, where each half is divided into two new parts until each part contains only one processor. For a detailed description see [17].

Here, we use a program written by Clemens Brand (Montan-Universität Leoben, Austria) with slight modifications by Uwe Reichel.

At last a number of variables and arrays are initialized with its start values. An important fact to notice is that all faces/edges of the coarse mesh are assumed to be coupling faces/edges no matter if they really connect the submeshes of two processors or if they are only within one submesh.

3.3 Refinement

3.3.1 The procedure

In the refinement step every volume is divided in 8 subvolumes, every face in 4 subfaces and every edge in two subedges.
There are in general 5 steps in the refinement procedure FEIN.

1. Preparation of the data array `COOR` for the new nodes. The nodes which belong to coupling faces/edges of the coarse mesh stand together in `COOR` in a well defined way. Namely all nodes on coupling edges (including the edges on coupling faces) stand together in that order they are located on the edge. For instance the nodes on a coupling edge with vertices A and B which was refined three times are located on these edge (and in `COOR`) in that way:

   $$\mathrm{A} — P_n — P_{n+1} — P_{n+2} — P_{n+3} — P_{n+4} — P_{n+5} — P_{n+6} — \mathrm{B}$$

   The middle node P_{n+3} is from the first refinement step. The nodes P_{n+1} and P_{n+5} were generated in the second refinement step, all other P_i in the third one. (In the quadratic case it looks identically but after the second refinement step.) So it is clear that nodes must be moved in the array `COOR` before the refinement.

The kette data were also updated in this step.

2. Refinement of the edges. All subedges which belong to coupling faces/edges stand together at the beginning of the data of the level. There are $X = 2^{\mathtt{NR}}$ subedges of a `NR` times refined edge. These X subedges stand together ordered as they are located on the old father edge. The X (linear case) / $2X$ (quadratic case) new nodes get their correct position in `COOR` as described above. The new subedges are appended to the list of edges in that order they were generated. Because of the special order of the father edges the described order principle is conserved.

3. Refinement of the faces. The subfaces of coupling faces stand together. New faces are appended. In the quadratic case, the middle nodes of the new inner edges are generated and in the case of brick meshes the middle node of the face too.

4. Refinement of the volumes. For tetrahedral meshes the rule after Bey (see [6]) is used to avoid the degeneration of the mesh. By analogy to the refinement of the edges, the middle nodes of the new edges (only quadratic case) and the middle nodes of the brick are generated. Note that also new faces have to be introduced.

5. Update of the boundary condition data. (The new subfaces with boundary conditions were added.)

There are some differences in details between the different mesh types (tetrahedral/brick, linear/quadratic):

For tetrahedral meshes it is easy. The difference between the quadratic and the linear case is whether the middle nodes of the edges are generated or not. That causes differences in the preparation and the edge refinement step.

For the brick meshes it is more complicated because of the new middle nodes of the faces/bricks. In the coarse mesh preparation step the middle nodes of the edges where generated. So it is an quadratic mesh of serendipity type. The normal refinement is also quadratic. In the linear case $N-1$ quadratic refinement steps were performed followed by an step to change the quadratic bricks to linear ones. It is the same procedure `FEIN` (there is an switch parameter) with the difference that now no new nodes on edges were generated. There is an third case with 27 node bricks (the middle nodes of the faces and the of brick are included and the order of the nodes of an volume is slightly different). After the normal quadratic refinement these additional nodes were generated in the subroutine `STROEM`.

3.3.2 An example

In general all nodes on coupling faces/edges stand together and form so called kettes. The nodes of 1D-kettes are ordered as they are located on the edges they belong to. The same is true for inner edges of 2D-kettes. But the vertices of these edges are not included in these edge node blocks because they are crosspoints (vertices of coupling edges), nodes on coupling edges or middle points of subfaces (vertices of edges on coupling faces), compare Figure 3.4.

To describe the structure of the array of node coordinates with all details the refinement of a single brick is considered. It is shown what happens with face number 2.

- The coarse mesh, see Figure 3.1:

  There are only the 8 crosspoints in `COOR`:

1 . . . 8	Crosspoints

- After the first refinement step, see Figure 3.2:

  Now there are in `COOR` beside the 8 crosspoints also the middle points of the coupling edges and faces and the first interior point in the center of the brick:

1 . . . 8	Crosspoints
9 . . . 20	middle points of the coupling edges
21 . . . 26	the middle points of coupling faces
27	the middle point of the brick

- After the second refinement step, see Figure 3.3:

  Now the coupling edges are two times refined and there are 3 nodes on it. The middle points (54. . . 57) of the 4 new subfaces and of the 4 edges on the coupling face appear(59. . . 62).

<table>
<tr><td>1. . . 8</td><td colspan="2"></td><td>Crosspoints</td></tr>
<tr><td>9. . . 11</td><td></td><td>CE_1</td><td rowspan="7">nodes on coupling edges</td></tr>
<tr><td>. . .</td><td></td><td>$CE_{2,3,4}$</td></tr>
<tr><td>21. . . 23</td><td></td><td>CE_5</td></tr>
<tr><td>24. . . 26</td><td></td><td>CE_6</td></tr>
<tr><td>. . .</td><td></td><td>$CE_{7,8}$</td></tr>
<tr><td>33. . . 35</td><td></td><td>CE_9</td></tr>
<tr><td>36. . . 44</td><td></td><td>$CE_{10,11,12}$</td></tr>
<tr><td>45. . . 53</td><td></td><td>CF_1</td><td rowspan="5">nodes on coupling faces</td></tr>
<tr><td>54. . . 57</td><td>middle nodes of the 4 subfaces</td><td rowspan="3">CF_2</td></tr>
<tr><td>58</td><td>the middle node of the face</td></tr>
<tr><td>59. . . 62</td><td>4 one time refined edges</td></tr>
<tr><td>63. . . 98</td><td></td><td>$CF_{3...6}$</td></tr>
<tr><td>99. . . 125</td><td colspan="2"></td><td>interior nodes</td></tr>
</table>

- After the third refinement step, see Figure 3.4:

  Now the coupling edges are three times refined and there are 7 nodes on it. Moreover, there were created the middle points of 4^2 subfaces from level 2 and the middle points of the 4^2 edges which where created in the previous step; compare the table below. Note that the middle node and nodes of the four edges of every refined face stand together (for instance nodes 158 . . . 162)

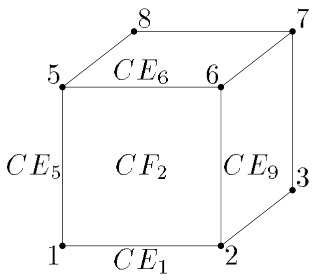

Figure 3.1: The coarse mesh

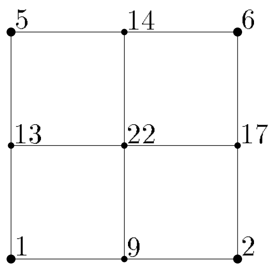

Figure 3.2: One refinement step

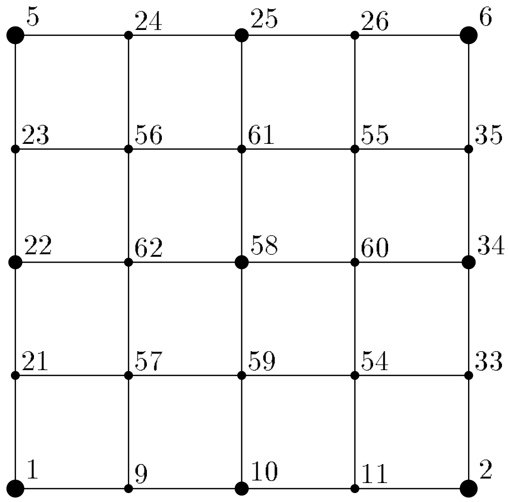

Figure 3.3: Second refinement step

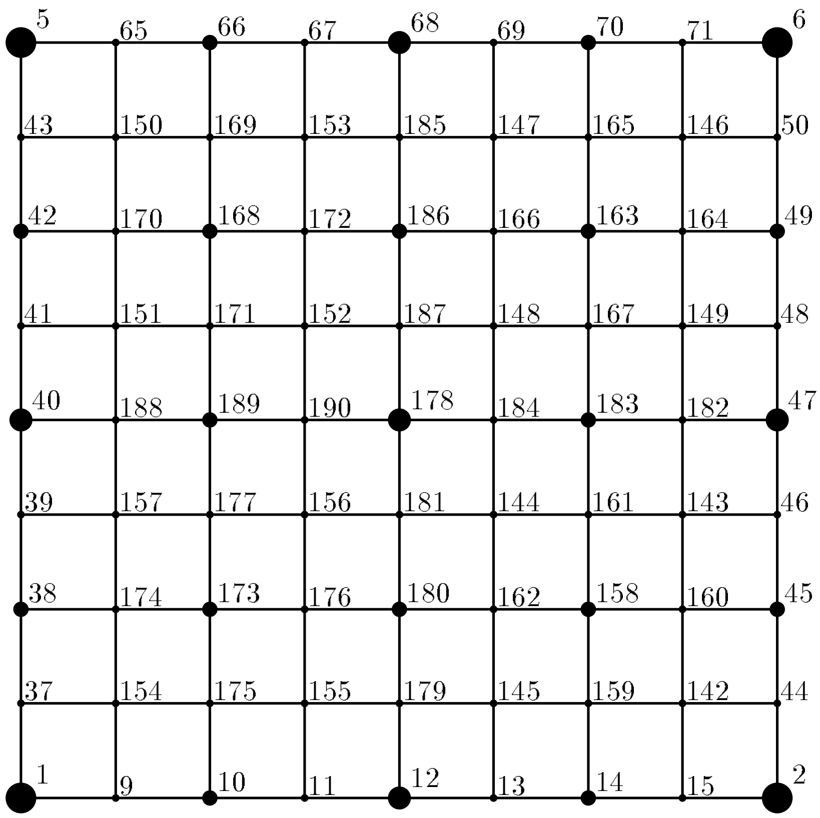

Figure 3.4: Third refinement step

1... 8			Crosspoints
9... 15		CE_1	nodes on coupling edges
...		$CE_{2,3,4}$	
37... 43		CE_5	
44... 50		CE_6	
...		$CE_{7,8}$	
65... 71		CE_9	
72... 92		$CE_{10,11,12}$	
93...141		CF_1	nodes on coupling faces
142...157	middle nodes of the 4^2 subfaces	CF_2	
158	middle node generated in step 2		
159...162	4 one time refined edges		
163	middle node generated in step 2		
164...167	4 one time refined edges		
168	middle node generated in step 2		
169...172	4 one time refined edges		
173	middle node generated in step 2		
174...177	4 one time refined edges		
178	the middle node of the face		
179...190	4 two times refined edges		
191...386		$CF_{3..6}$	
387...729			interior nodes

Note:

- There are no middle nodes of faces in tetrahedral meshes.
- In the quadratic case there is the same structure of `COOR` but all edges have their middle points. So the edges appear one step earlier and they look like one time more refined linear edges.

3.4 Reducing data

The change of the data structure is easy. It is only necessary to determine the nodes a volume/face consists of and to calculate the Dirichlet values on some nodes.

3.5 Parameters of the output tool `AUSGABE`

The subroutine `AUSGABE` is an output tool for several (mesh) data. It works with the FDS as well as with the RDS. There is an array of logical variables `STEUER` which determines the status of the data (FDS/RDS/solved - `STEUER`(1/2/3)).

Features of `AUSGABE`:

- graphical output of mesh data (GRAPE)
- tabular output of mesh data
- tabular output of kette data

- tabular output of the solution/error
- tabular output of error norms
- output of the mesh as standardized file **.std* (works only as one processor version)

```
SUBROUTINE AUSGABE(STEUER,COOR,ZKP,ZFP,ZIP,NUMNP,KANTE,FKANTE,
                   NKANTE,FACE,FFACE,NFACE,VOL,REGION,FVOL,
                   NUMEL,NKKO,NKFO,DIR,FDIR,NDIR,NEUM,FNEUM,
                   NNEUM,KETTE1D,KETTE2D,CHAIN,VFS,X,LC,IER,
                   IGLOB,FREI,LAEFREI)
```

STEUER		mesh status
COOR		array of node coordinates
ZKP	FDS	pointer in COOR, first node on 1D kette
ZFP	FDS	pointer in COOR, first node on 2D kette
ZIP	FDS	pointer in COOR, first inner node
NUMNP		number of nodal points
KANTE	FDS	array of edges
FKANTE	FDS	pointer to the last level in KANTE
NKANTE	FDS	number of edges (only last level)
FACE	FDS	array of faces
FFACE	FDS	pointer to the last level in FACE
NFACE	FDS	number of faces (only last level)
VOL		array of volumes
REGION		array of region data
FVOL	FDS	pointer to the last level in VOL
NUMEL		number of volumes (only last level)
NKKO		number of 1D kettes
NKFO		number of 2D kettes
DIR		array of Dirichlet data
FDIR	FDS	pointer to the last level in DIR
NDIR		number of Dirichlet faces (only last level)
NEUM		array of Neumann data
FNEUM	FDS	pointer to the last level in NEUM
NNEUM		number of Neumann faces (only last level)
KETTE1D		array of 1D kette data
KETTE2D		array of 2D kette data
CHAIN		array of chain data
VFS		number of refinement steps
X	RDS	solution
LC	RDS	hierarchical list
IER		error parameter
IGLOB		array of global crosspoint names
FREI		workspace array
LAEFREI		length of the workspace array

All variables except the error parameter IER are input.

3.6 Tree structure of the routines

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

3.6.1 NETMAKE for tetrahedral meshes

```
NETMAKE
    ↪ GROBNETZ
        ↪ READDATTR
        ↪ SIMP_MARK
        ↪ VERTEILEN
        ↪ VERTEILEN1
    ↪ FEINCALL
        ↪ FEIN
        ↪ COM_PROB
    ↪ MOVE
    ↪ STAUCHEN

VERTEILEN
    ↪ SET_RBCOM
    ↪ DAT_DOWN
    ↪ TKUERZ
    ↪ KUERZEN
    ↪ ZUERST
        ↪ PCORRECT
        ↪ PFACE
            ↪ GEMPKT
    ↪ COM_PROB

VERTEILEN1
    ↪ SET_RBCOM
    ↪ DAT_DOWN
    ↪ RSB
    ↪ TKUERZ
    ↪ KUERZEN
    ↪ ZUERST *
    ↪ COM_PROB

RSB
    ↪ SPEBIS
        ↪ MEDIAN
        ↪ OUTINT1
    ↪ BUILDHV
    ↪ MATRIX
    ↪ EIGVAL
        ↪ INILANC
            ↪ SEED
            ↪ RANDOM
            ↪ MATVEC
        ↪ LCZSTEP
            ↪ MATVEC
        ↪ TRIDEV
            ↪ DELTA
        ↪ BOUNDS
    ↪ FIEDLER
        ↪ INVITER
            ↪ TRIDSOL

FEIN
    ↪ COORPLATZ
        ↪ KPLATZ
    ↪ TEILEKANTEN
        ↪ TKANTEN
            ↪ PCORECT
            ↪ KWRITE
    ↪ TEILEFLAECHEN
        ↪ TFLAECHEN
            ↪ PCORECT
            ↪ KWRITE
            ↪ SCHREIBEDREIECK
    ↪ ECKPUNKTE
        ↪ GEMPKT
    ↪ GEMPKT
    ↪ MITTE
    ↪ KWRITE
    ↪ SCHREIBEDREIECK
    ↪ TETSCHREIBEN
    ↪ RBNEU

STAUCHEN
    ↪ DIRSTAUCHEN
        ↪ PFACE *
        ↪ DIRINTPO
    ↪ NEUMSTAUCHEN
        ↪ PFACE *
    ↪ VOLPUNKTE
        ↪ ECKPUNKTE
            ↪ GEMPKT
        ↪ PBRICK
            ↪ GEMKANTE *
            ↪ GEMPKT
    ↪ MAKE_LC
```

3.6.2 NETMAKE for brick meshes

```
NETMAKE
    ↪ GROBNETZ
        ↪ READDATQ
        ↪ SIMPMARK
        ↪ VERTEILEN
    ↪ FEINCALL
        ↪ FEIN
        ↪ COM_PROB
    ↪ STROEM
    ↪ MOVE
    ↪ STAUCHEN

VERTEILEN
    ↪ SET_RBCOM
    ↪ DAT_DOWN
    ↪ TKUERZ
    ↪ KUERZEN
    ↪ FORIENT
    ↪ VORIENT
    ↪ ZUERST
        ↪ PCORRECT
        ↪ FEBRICK
            ↪ GEMPKT
    ↪ COM_PROB

FEIN
    ↪ COORPLATZ
        ↪ KPLATZ
    ↪ KFTEILEN
        ↪ KTEILEN
            ↪ PCORECT
            ↪ KWRITE
    ↪ FFTEILEN
        ↪ FTEILEN
            ↪ PCORECT
            ↪ KWRITE
            ↪ FAWRITE
    ↪ KWRITE
    ↪ OFDAT
        ↪ GDKANTE
            ↪ GTKANTE
            ↪ KWRITE
    ↪ FAWRITE
    ↪ VWRITE
    ↪ RBNEU

STROEM
    ↪ FMPLATZ
    ↪ FMITTEN
        ↪ FMID
            ↪ KWRITE
    ↪ VMITTEN
        ↪ KWRITE

STAUCHEN
    ↪ DIRSTAUCHEN
        ↪ PFACE *
        ↪ DIRINTPO
    ↪ NEUMSTAUCHEN
        ↪ PFACE *
    ↪ VOLPUNKTE
        ↪ ECKPUNKTE
            ↪ GEMPKT
        ↪ PBRICK
            ↪ PFACE *
            ↪ FEBRICK *
    ↪ MAKE_LC
```

3.6.3 AUSGABE

```
AUSGABE
    ↪ IAUS
    ↪ GEBGRAPE
    ↪ NETZREDOUT
        ↪ FSTRADDI
        ↪ FSTRADDR
        ↪ RDIRPRINT
        ↪ VRBPRINT
        ↪ LCPRINT
    ↪ NETZDRUCK
        ↪ FSTRADDI
        ↪ FSTRADDR
        ↪ VRBPRINT
    ↪ KETPOUT
    ↪ WTABX
        ↪ PWTABX
    ↪ FNTAB
    ↪ STDF_OUT
```

3.7 Short description of the routines in *libNA.a*

The following FORTRAN sources are located in *Netz/Allgemein*.

AUSGABE	*netzdruck.f*	frame for the output of several data (mesh, solution, error estimates)
BOUNDS	*rsb.f*	a bound on the Raleigh-Ritz approximation θ_j to the maximum eigenvalue λ is computed
BUILDHV	*rsb.f*	accumulation of an auxiliary array for the spectral bisection
COM_PROB	*com_prob.f*	sets the variables of the common block in *com_prob.inc*
DAT_DOWN	*dat_down.f*	distributes mesh data from processor 0 to all other processors
DATRED	*kuerzen.f*	deletes faces/edges/nodes which are not referred in the volumes/faces/edges; generates IGLOB and deletes unused boundary condition data
DELTA	*rsb.f*	function to calculate a special determinant during RSB
DIRINTPO	*stauchen.f*	computes the Dirichlet value at a node
DIRSTAUCHEN	*stauchen.f*	changes the Dirichlet data to the reduced data structure
DREGIO	*stdwrite.f*	determines the names of regions and the number of volumes per region
ECKPUNKTE	*AUpfein.f*	determines the vertices of a tetrahedron
EIGVAL	*rsb.f*	computes the second smallest eigenvalue of the Laplacian matrix
FEBRICK	*AUpfein.f*	determines the vertices of a quadrilateral
FIEDLER	*rsb.f*	computes the fiedler vector, i.e. the eigenvector corresponding to the second largest eigenvalue of the Laplacian matrix
FSTRADDI	*netzdruck.f*	generates a format string
FSTRADDR	*netzdruck.f*	generates a format string
GEMKANTE	*AUpfein.f*	determines the common edge of two faces
GEMPUNKT	*AUpfein.f*	determines the common node of two edges
GETDOFS	*getdofs.f*	description of the degrees of freedom for the visualization tool GRAPE
IAUS	*netzdruck.f*	displays a menu for the different output routines
INILANC	*rsb.f*	initialization of the Lanczos-method
INVITER	*rsb.f*	inverse iteration for calculation of the eigenvector to a given eigenvalue
KETPOUT	*netzdruck.f*	output of kette data (for kette see [4])
KPLATZ	*AUpfein.f*	copies the nodes of an edge to the new places in COOR (before the refinement step)
KUERZEN	*kuerzen.f*	deletes unused mesh data and performs the necessary renumbering, generates the array of global crosspoint names IGLOB (Note that DAT_DOWN distributes the whole coarse mesh, then some elements are marked, and KUERZEN deletes all elements not marked.)
KWRITE	*AUpfein.f*	writes an edge into the array of edges
LCPRINT	*netzdruck.f*	output of one row of the hierarchical list

LCZSTEP	*rsb.f*	iteration step of the Lanczos-method
LIES	*standard.f*	reads and analyzes a row of the file of program control variables (*control.quad* or *control.tet*, respectively)
MAKE_LC	*stauchen.f*	generates the hierarchical list
MAKREGIO	*stdwrite.f*	prepares the REGION data for STDWRITE
MATRIX	*rsb.f*	generates the Laplacian matrix of element connectivity
MATVEC	*rsb.f*	matrix vector multiplication
MEDIAN	*rsb.f*	calculates the median of a set (array) of numbers
MOVE	*cnetz.f*	realization of a coordinate transformation for special applications
NEUMSTAUCHEN	*stauchen.f*	changes the Neumann data to the reduced data structure
NETZDRUCK	*netzdruck.f*	output of the full data structure
NETZREDOUT	*netzdruck.f*	output of the reduced data structure
OUT	*netzdruck.f*	displays the part of the solution vector of one processor
OUTINT1	*rsb.f*	auxiliary routine for integer output
OUTKETTE	*netzdruck.f*	displays the part of the kette data (for kette see [4]) of one processor
OUTSTANDARD	*standard.f*	displays program control variables
PBRICK	*AUpfein.f*	determines the 8/20 nodes of an linear/quadratic brick
PCORECT	*pcorect.f*	determines the middle point of an edge
PFACE	*AUpfein.f*	determines the nodes of a face
PWTABX	*netzdruck.f*	displays one row of the table of the solution/error
RANDOM	*rsb.f*	calculates a random value
RBNEU	*AUpfein.f*	copies the boundary condition data from father faces to the new subfaces after a mesh refinement step
RDIRPRINT	*netzdruck.f*	output of Dirichlet data (reduced data structure)
RNDPRINT	*stdwrite.f*	output of boundary condition data
RSB	*rsb.f*	computes the spectral bisection for a given mesh
SEED	*rsb.f*	a very basic random number generator; uses RANDOM
SETFILE	*setfile.f*	input of the filename
SET_RBCOM	*set_rbcom.f*	sets the values of the variables in the common block RB in the file *net3ddat.inc*
SETSTANDARD	*standard.f*	sets the program control variables using file *control.tet*/ *control.quad*
SIMP_MARK	*simp_mark.f*	marks the volumes with the number of the processor which should own them (very simple method)
SPEBIS	*rsb.f*	the median value of the fiedler vector is computed; all elements of the fiedler lower then the median are marked
STAUCHEN	*stauchen.f*	reduction of the full data structure to the reduced data structure
STDF_OUT	*stdwrite.f*	frame for the output of the full data structure as a standard file **.std*
STDWRITE	*stdwrite.f*	output of the full data structure as a standard file
TKUERZ	*kuerzen.f*	deletes all volumes which are not marked with the own processor number (array MARK)
TRIDEV	*rsb.f*	computes largest eigenvalue of the tridiagonal matrix

TRIDSOL	*rsb.f*	solver for a tridiagonal, symmetric, positive definite matrix
VERSION	*version.f*	displays the title of the program
VOLPUNKTE	*stauchen.f*	determines the vertices of a volume
VRBPRINT	*netzdruck.f*	output of Dirichlet data (full data structure)
VREGIO	*stdwrite.f*	determines the names of the volumes in the regions
WTABX	*netzdruck.f*	table of the solution
YSFAKTOR	*cnetz.f*	determines the relative length of the subedges
ZWEIIWERTE	*standard.f*	reads two integer values from an string variable

3.8 Short description of the routines in *libNT.a*

The FORTRAN sources are located in *Netz/Tetraeder*.

COORPLATZ	*upfein.f*	prepares COOR for the following mesh refinement step (The nodes in COOR are ordered. A renumbering is necessary because some of the new points will be placed between old points.)
GROBNETZ	*grobnetz.f*	provides the coarse mesh
FEIN	*upfein.f*	hierarchical mesh refinement of tetrahedral meshes
FEINCALL	*upfein.f*	frame for the hierarchical mesh refinement
NETMAKE	*netmake.f*	frame for the mesh generation
MITTE	*upfein.f*	determines the middle point of an edge of an face
SCHREIBEDREIECK	*upfein.f*	writes one triangle into the array of faces
SET_NETDIM	*control.f*	sets constants (especially array dimensions) for tetrahedral meshes
STWERTE	*control.f*	presets the program control variables with standard values and opens the file *control.tet*
TEILEFLAECHEN	*upfein.f*	divides the faces
TEILEKANTEN	*upfein.f*	divides the edges
TETSCHREIBEN	*upfein.f*	writes one tetrahedron into the array of volumes
TFLAECHEN	*upfein.f*	divides one face
TKANTEN	*upfein.f*	divides one edge
VERTEILEN	*grobnetz.f*	sends the coarse mesh data to all processors, deletes all unused data and prepares the mesh for refinement (simple spreading of the volumes)
VERTEILEN1	*grobnetz.f*	sends the coarse mesh data to all processors, deletes all unused data and prepares the mesh for refinement (spreading of the volumes with recursive spectral bisection)
ZUERST	*grobnetz.f*	prepares the coarse mesh for the refinement (sets initial values of variables etc.)

3.9 Short description of the routines in *libNQ.a*

The FORTRAN sources are located in *Netz/Quader*.

`COORPLATZ`	*upfein.f*	prepares `COOR` for the following mesh refinement step (The nodes in `COOR` are ordered. A renumbering is necessary because some of the new points will be placed between old points.)
`F6`	*upfein.f*	determines face number 6 (used in `VORIENT`)
`FAWRIT`	*upfein.f*	writes one quadrilateral into the array of faces
`FEIN`	*upfein.f*	hierarchical mesh refinement of brick meshes
`FEINCALL`	*upfein.f*	frame for the hierarchical mesh refinement
`FFTEILEN`	*upfein.f*	divides the faces
`FORIENT`	*upfein.f*	orders the four edges of an quadrilateral (the first is connected with the second which is connected with the third which is connected with the fourth which is connected with the first)
`FTEILEN`	*upfein.f*	divides one face
`FMID`	*stroem.f*	determines the middle nodes of one face
`FMITTEN`	*stroem.f*	determines the middle nodes of the faces
`FMPLATZ`	*stroem.f*	prepares the data array `COOR` for the new middle nodes of the faces and volumes (for 27 node bricks)
`GDKANTE`	*upfein.f*	determines the relative position of subfaces of two father brick faces
`GROBNETZ`	*grobnetz.f*	provides the coarse mesh
`GTKANTE`	*upfein.f*	determines the common edge of two subfaces (used in subroutine `GDKANTE`)
`KTEILEN`	*upfein.f*	divides one edge
`KFTEILEN`	*upfein.f*	divides the edges
`NETMAKE`	*netmake.f*	frame for the mesh generation
`OFDAT`	*upfein.f*	determines the subfaces and inner subedges of an father brick face
`SET_NETDIM`	*control.f*	sets constants for brick meshes
`STROEM`	*stroem.f*	determines the middle nodes of the faces and volumes (for 27 node bricks)
`STWERTE`	*control.f*	presets the program control variables with standard values and opens the file *control.quad*
`TAUSCHE`	*upfein.f*	exchange of the values of two integer variables
`VERTEILEN`	*grobnetz.f*	sends the coarse mesh data to all processors, deletes all unused data and prepares the mesh for refinement (simple spreading of the volumes)
`VMITTEN`	*stroem.f*	determines the middle nodes of the volumes
`VORIENT`	*upfein.f*	orders the six faces of a brick (the first is not connected with the 6th and the other four faces are connected as described for the edges of a quadrilateral above)
`VWRITE`	*upfein.f*	writes a brick into the array of volumes
`ZUERST`	*grobnetz.f*	prepares the coarse mesh for the refinement (sets initial values of variables etc.)

Chapter 4

Assembly of the equation system and error estimation

4.1 General remarks

The assembly of the equation system and the error calculation have the common feature of the computation of integrals over volumes (elements) and faces. Moreover, values of shape functions are needed only in these two parts of the program. That is why we put the sources of the related subroutines together in the directory *Assem* and the object files form the library *libA*`$archi`*.a*. The aim of this chapter is to describe the realization of these routines. The hope is that many of them can be used in further extensions of the program system *SPC-PMPo 3D* or in other related programs.

Many routines have been taken from the sequential code *FEMPS3D*. A brief description of this code is contained in [5]. But there are also some significant differences:

1. In *FEMPS3D*, only the Poisson equation can be solved, while in *SPC-PMPo 3D* also the Lamé system is included.

2. In *FEMPS3D*, the error computation is restricted to linear tetrahedral elements, here this has been extended to the full scale of elements treatable, see below. Moreover, some communication is necessary.

3. The storage of the system matrix is slightly different, the storage and the treatment of boundary conditions is fully different due to the influence of the code *SPC-PMPo 2D*.

Because of these changes the parameter lists of many of the subroutines have been changed, thus it is not suggested to mix the subroutines from the serial and the parallel program.

Essentially, there are two subroutines, which are called from programs outside the library *libA.a*: `ASSLOES` and `FNTAB`. In both parts, numerical integration is used; we describe the corresponding routines in Section 4.2.

`ASSLOES` realizes the frame for the assembly and solution of the equation system: Given the finite element mesh, some control parameters and auxiliary memory, the routine returns the finite element solution. The matrix and the right hand side are local in `ASSLOES`. The main ingredients are: initialization, the call of `ASSEM` and `COARSMAT3`, the call of several routines for the solution of the equation system (see Chapter 5) and time measurement. The stiffness matrix and the right hand side are assembled in `ASSEM`; the main steps are described in Sections 4.2–4.3. Moreover, a coarse grid matrix is assembled in `COARSMAT3`, see 4.3.5.

The calculation of the error norms and their output (for each subdomain (processor) and global) is done by `FNTAB`. The different error measures are the discrete maximum norm (maximal difference of u and u_h in nodal points), and the H^1-and L_2-seminorms of $u - u_h$ (calculated via numerical integration).

Of course, these norms can only be calculated if the exact solution is programmed. Note that the module *bsp.f* supplies function routines for u, its derivatives, and the right hand sides f and g. They have to be programmed by the user. The realization of the error norms is described in Section 4.4.

For the tree structure of `ASSLOES/ASSEM` and `FNTAB` see Section 4.5, all routines are described shortly in Section 4.6.

4.2 Numerical integration and shape functions

4.2.1 Mathematical Background

The assembly of the stiffness matrix and the right hand side as well as the calculation/estimation of error norms require the numerical integration of integrals over volume elements or faces. This integration is realized by a transformation to reference elements, that means a coordinate transformation: Let Ω be a volume element (tetrahedron or hexahedron) with nodes $x_i \in \mathbb{R}^3$ ($i = 1, \dots,$ `NEN3D`). Moreover, let $\hat{\Omega}$ be the reference element with nodes $\hat{x}_i \in \mathbb{R}^2$ ($i = 1, \dots,$ `NEN3D`) and shape functions $\hat{\varphi}_i(\hat{x})$, with $\hat{\varphi}_i(\hat{x}_j) = \delta_{ij}$, then the corresponding coordinate transformation is

$$x = \sum_{i=1}^{\texttt{NEN3D}} x_i \hat{\varphi}_i(\hat{x}),$$

and the volume integral is transformed and approximated by

$$\int_\Omega f(x)\, dx = \int_{\hat{\Omega}} \hat{f}(\hat{x}) \cdot |\det(J(\hat{x}))|\, d\hat{x} \approx \sum_{j=1}^{\texttt{NQP3}} \hat{f}(\hat{y}_j) \cdot |\det(J(\hat{y}_j))| \omega_j$$

where $\hat{y}_j$ are the integration points, ω_j are the weights, and J is the Jacobian functional matrix.

The two-dimensional case is treated by analogy: Let G be a face (triangle or quadrilateral) in $\mathbb{R}^2$ with nodes $x_i \in \mathbb{R}^3$ ($i = 1, \dots,$ `NEN2D`), and let $\hat{G}$ be the reference element with nodes $\hat{x}_i \in \mathbb{R}^2$ ($i = 1, \dots,$ `NEN2D`) and shape functions $\hat{\varphi}_i(\hat{x})$, then the relations are

$$x = \sum_{i=1}^{\texttt{NEN2D}} x_i \hat{\varphi}_i(\hat{x}),$$

$$\int_G f(x)\, dx = \int_{\hat{G}} \hat{f}(\hat{x}) \sqrt{EC - F^2}\, d\hat{x} \approx \sum_{j=1}^{\texttt{NQP2}} \hat{f}(\hat{y}_j) \sqrt{E(\hat{y}_j)C(\hat{y}_j) - F^2(\hat{y}_j)}\, \omega_j$$

$$\text{where } E(\hat{x}) = \sum_{i=1}^{3} \left(\frac{\partial x^{(i)}}{\partial \hat{x}^{(1)}}\right)^2, \quad C(\hat{x}) = \sum_{i=1}^{3} \left(\frac{\partial x^{(i)}}{\partial \hat{x}^{(2)}}\right)^2, \quad F(\hat{x}) = \sum_{i=1}^{3} \frac{\partial x^{(i)}}{\partial \hat{x}^{(1)}} \cdot \frac{\partial x^{(i)}}{\partial \hat{x}^{(2)}},$$

and $x = (x^{(1)}, x^{(2)}, x^{(3)})$, $\hat{x} = (\hat{x}^{(1)}, \hat{x}^{(2)})$.

Thus it is useful to supply arrays QGST(3,NQP2) and QGST3(4,NQP3) with the quadrature points and the corresponding weights, as well as arrays SHP2(3,NEN2D,NQP2) and SHP3(4,NEN3D,NQP3) with the values of the shape functions and their derivatives in the quadrature points.

4.2.2 The arrays QGST2, QGST3, SHP2, and SHP3

Actually, these arrays depend on the element type (triangle / quadrilateral, tetrahedron / pentahedron / hexahedron). In the sequential code *FEMPS3D*, see [2, 5], there has been no restriction to use one type of elements exclusively. Thus all arrays could have been necessary in one run of the program. So it was decided to supply the arrays mentioned above for all element types. This means:

- NQP2 and NQP3 are vectors of length 2 and 3, respectively:

  NQP2(1)... number of quadrature points for quadrilaterals,
  NQP2(2)... number of quadrature points for triangles,
  NQP3(1)... number of quadrature points for hexahedra,
  NQP3(2)... number of quadrature points for pentahedra,
  NQP3(3)... number of quadrature points for tetrahedra.

  These arrays are assigned in E3LEHF, because the user supplies only the number of the formula, see below, and not the corresponding number of quadrature points.

- The actual length of the arrays are:

array	length
QGST2	$3*(\texttt{NQP2(1)}+\texttt{NQP2(2)})$
QGST3	$4*(\texttt{NQP3(1)}+\texttt{NQP3(2)}+\texttt{NQP3(3)})$
SHP2	$3*(3*\texttt{NQP2(1)}+4*\texttt{NQP2(2)})$ for linear elements $3*(6*\texttt{NQP2(1)}+8*\texttt{NQP2(2)})$ for quadratic elements
SHP3	$4*(8*\texttt{NQP3(1)}+6*\texttt{NQP3(2)}+4*\texttt{NQP3(3)})$ for linear elements $4*(20*\texttt{NQP3(1)}+15*\texttt{NQP3(2)}+10*\texttt{NQP3(3)})$ for quadratic elements

  Of course, one can call subroutines with a part of the array, and use the array as introduced in 4.2.1. These arrays are allocated in E3LEHF.

The quadrature formulae programmed are given in Tables 4.1 to 4.5. They are realized in the subroutines E2INTG and E3INTG on information from the books [1, 15, 20], for a description of the routines see Section 4.6.

The arrays SHP2 and SHP3 are assigned in E2SHAP and E3SHAP using the arrays QGST2 and QGST3 as well as subroutines where the shape functions and their derivatives for the different cases are programmed (P2L, P2Q for triangles, PHI2L, PHI2Q for quadrilateral, PTL, PTQ for tetrahedra, P3L, P3Q for pentahedra, and PHI3L, PHI3Q for hexahedra). The 15 subroutines mentioned in this paragraph are contained in the file *upe2e3.f*.

4.2.3 A modification of the arrays for the use in the error estimator

An additional case must be considered for the error estimator, compare Section 4.4, Here, the values of the 3D shape functions and their derivatives are used in the quadrature points

Formula number	Number of points	Description	exact for x^iy^j with
1	1	midpoint (center of gravity)	$i,j \leq 1$
2	4	2x2 Gaussian points	$i,j \leq 3$
3	9	3x3 Gaussian points	$i,j \leq 5$

Table 4.1: Quadrature formulas for quadrilaterals.

Formula number	Number of points	Description	exact for x^iy^j with
1	1	center of gravity	$i,j \leq 1$
2	3	midpoints of the edges	$i,j \leq 2$
3	4	Gaussian points	$i,j \leq 3$
4	7	Gaussian points	$i,j \leq 5$

Table 4.2: Quadrature formulas for triangles.

Formula number	Number of points	Description	exact for $x^iy^jz^k$ with
1	1	center of gravity	$i+j+k \leq 1$
2	4	Gaussian points	$i+j+k \leq 2$
3	5	Gaussian points	$i+j+k \leq 3$
4	11	Gaussian points	$i+j+k \leq 4$
5	14	Gaussian points	$i+j+k \leq 5$

Table 4.3: Quadrature formulas for tetrahedra.

Formula number	Number of points	the formula is a cross product of the formulas for triangle	for interval (z-direction)	exact for $x^iy^jz^k$ with
1	$1 = 1 \cdot 1$	center of gravity	midpoint	$i+j \leq 1, k \leq 1$
2	$3 = 3 \cdot 1$	midpoints of edges	midpoint	$i+j \leq 2, k \leq 1$
3	$4 = 4 \cdot 1$	4 Gaussian points	midpoint	$i+j \leq 3, k \leq 1$
4	$6 = 3 \cdot 2$	midpoints of edges	2 Gaussian points	$i+j \leq 2, k \leq 3$
5	$8 = 4 \cdot 2$	4 Gaussian points	2 Gaussian points	$i+j \leq 3, k \leq 3$
6	$12 = 4 \cdot 3$	4 Gaussian points	3 Gaussian points	$i+j \leq 3, k \leq 5$
7	$14 = 7 \cdot 2$	7 Gaussian points	2 Gaussian points	$i+j \leq 5, k \leq 3$
8	$21 = 7 \cdot 3$	7 Gaussian points	3 Gaussian points	$i+j \leq 5, k \leq 5$

Table 4.4: Quardature formulas for pentahedra.

Formula number	Number of points	Description	exact for $x^iy^jz^k$ with
1	1	midpoint (center of gravity)	$i,j,k \leq 1$
2	8	2x2x2 Gaussian points	$i,j,k \leq 3$
3	27	3x3x3 Gaussian points	$i,j,k \leq 5$
4	6	midpoints of the faces	$i+j+k \leq 3$
5	14	Irons formula	$i+j+k \leq 5$

Table 4.5: Quadrature formulas for hexahedra.

of the faces of the elements. This is still under development.

4.3 Assembly of the stiffness matrix and the right hand side

4.3.1 Main steps

The stiffness matrix A and the right hand side F are assembled in the subroutine `ASSEM`. Only the non-zero elements of A are stored, see [13]. The main steps are the following:

1. Allocation of memory for the matrix `A`, its index vector `LA(N)`, the right hand side `F(N)`, and the solution `X(N)`, where `N` = `NUMNP`$*$`NDF` is the number of unknowns, `NUMNP` is the number of nodal points, and `NDF` is the number of degrees of freedom per node. Our approach is not to determine the structure of A a priori, but to use a rectangular matrix `A(NDIAG,N)` preliminary. The maximal number `NDIAG` of non-zero entries in a row of the matrix must be estimated before, see Section 2.3 of the User's manual [3]. If it proves to be too small in step 4, then it is increased iteratively.

2. Initialisation of `A`, `LA`, `F`, and `X`. The subroutine `MAKEKZU` initializes the index vector `LA` as if the matrix has exactly `NDIAG` non-zero elements per row.

3. Allocation and calculation of the arrays `QGST2`, `QGST3`, `SHP2`, and `SHP3`, see Section 4.2.

4. Loop over all elements: The element stiffness matrix S and the element right hand side P are computed in subroutine `ELEMENT`, then they are accumulated using `AKKU/AKKUEL` and `FAKKU/FAKKUEL`. Note that the first parameter in `ELEMENT` is the equation number: If it is equal to 1, then the Poisson equation is taken as the basis, and internally the subroutine `ELS` is called, see 4.3.2. In the case of linear elasticity, the parameter is 2 and the routine `ELAST` is used, see 4.3.3. In `ELEMENT`, there are also the element types (tetrahedron,...) distinguished via the auxiliary routine `IHPT`.

5. Treatment of Dirichlet data: Dirichlet data are stored in an array `DIRF`, see Section 2.3. Depending on the indicator in `DIRF(NEN2D+1,*)`, the values are copied to the appropriate position in the vector `X` and the corresponding diagonal element of `A` is set to 1.D+40. The realization of the CG-algorithm ensures the correct handling of these equations.

6. Treatment of inhomogeneous Neumann data: They are stored in the array `NEUMF`, see Section 2.3. In a loop over all Neumann faces the integrals for the element right hand side are calculated via subroutines `NEUMANN/E3RSOB` and then accumulated via `FAKKU/FAKKUEL`. Note that some faces may have Dirichlet and Neumann data for different degrees of freedom. Then they are included both in `DIRF` and `NEUMF`.

7. In a last step the matrix A is compressed by deleting all zero entries (subroutine `PACKKLZ`). Moreover, the elements in each row are now sorted by its index of the column (subroutine `SORTKZU`).

4.3.2 The element routine ELS in the case of the Poisson equation

We are going now to describe the element routine `ELS` which calculates the element stiffness matrix $S = (s_{k\ell})_{k,\ell=1}^{\mathtt{NEN3D}}$ and the right hand side $P = (p_k)_{k=1}^{\mathtt{NEN3D}}$ in the case of the Poisson equation. With the notation from 4.2.1 there holds:

$$\begin{aligned} s_{k\ell} &= \int\limits_{\Omega} (\nabla\varphi_k)^T \cdot \nabla\varphi_\ell \, dx = \int\limits_{\hat\Omega} (J^{-T}\hat\nabla\hat\varphi_k)^T \cdot (J^{-T}\hat\nabla\hat\varphi_\ell) \cdot |\det J| \, d\hat x \\ &\approx \sum_{j=1}^{\mathtt{NQP3}} \left(J(\hat y_j)^{-T}\hat\nabla\hat\varphi_k(\hat y_j)\right)^T \left(J(\hat y_j)^{-T}\hat\nabla\hat\varphi_\ell(\hat y_j)\right) |\det J(\hat y_j)| \cdot \omega_j \\ p_k &= \int\limits_{\Omega} f(x)\varphi_k(x)\, dx = \int\limits_{\hat\Omega} \hat f(\hat x)\hat\varphi_k(\hat x) |\det J(\hat x)| \, d\hat x \\ &\approx \sum_{j=1}^{\mathtt{NQP3}} \hat f(\hat y_j)\hat\varphi_k(\hat y_j) |\det J(\hat y_j)| \omega_j \end{aligned}$$

Thus, after initialization of `S` and `P` with zeros, a loop over all quadrature points y_j is performed, where the following steps are carried out:

1. Calculation of $J = J(\hat y_j) = (J_{mn})_{m,n=1}^3$, J^{-1}, and $|\det J|$, with

   $$J_{mn} = \frac{\partial x^{(m)}}{\partial \hat x^{(n)}} = \frac{\partial}{\partial \hat x^{(n)}} \sum_{i=1}^{\mathtt{NEN3D}} x_i^{(m)} \hat\varphi(\hat x) = \sum_{i=1}^{\mathtt{NEN3D}} x_i^{(m)} \frac{\partial\hat\varphi_i}{\partial \hat x^{(n)}}.$$

   The values of $\frac{\partial\hat\varphi_i(\hat y_j)}{\partial\hat x^{(n)}}$ are contained in the array `SHP3`, see 4.2.1.

   Note that in the case of a linear tetrahedron the matrix J is independent of $\hat x$ and this step can be executed outside the loop. More than one integration point can be useful for linear tetrahedra in the case of non-constant right hand side f.

2. Preparation of an array $D = (d_{mi})_{m=1,i=1}^{3,\mathtt{NEN3D}} = J^{-T}\left(\hat\nabla\hat\varphi_k\right)_{k=1}^{\mathtt{NEN3D}}$.

3. $S := S + DD^T \cdot \omega_j \cdot |\det J|$, $\quad \omega_j = \mathtt{QGST3(4,j)}$

4. Transformation of the quadrature point:

   $$y_j = \sum_{i=1}^{\mathtt{NEN3D}} x_i \hat\varphi_i(\hat y_j)$$

5. $p_k := p_k + f(y_j)\hat\varphi_k(\hat y_j)|\det J| \cdot \omega_j$

   Note that $f(y_j) = \hat f(\hat y_j)$, but only $f(.)$ is given as a function.

4.3.3 The element routine ELAST for the Lamé system

The element routine `ELAST` for linear elasticity is programmed with similar steps as in 4.3.2. Denote by $S = (S_{k\ell})_{k,\ell=1}^{\mathtt{NEN3D}}$ the element stiffness matrix with $S_{k\ell} \in \mathbb{R}^{3\times 3}$ and by $P = (P_k)_{k=1}^{\mathtt{NEN3D}}$,

$P_k \in \mathbb{R}^3$, the element right hand side. Then one finds that

$$\begin{aligned} S_{k\ell} &:= \lambda T_{k\ell} + \mu T_{k\ell}^T + \mu \cdot \operatorname{tr}(T_{k\ell}) \cdot I, \quad T_{k\ell} := \int_\Omega \nabla\varphi_k (\nabla\varphi_\ell)^T \, dx, \\ P_k &:= \int_\Omega f(x)\varphi_k(x)\, dx \quad (f(x) \in \mathbb{R}^3), \end{aligned}$$

which can be calculated similary to 4.3.2. Thus the steps in the loop are:

1. Calculation of J, J^{-1}, and $|\det J|$.
2. Calculation of $D = J^{-T}(\hat\nabla\hat\varphi_k)_{k=1}^{\mathtt{NEN3D}} \in \mathbb{R}^{3\times \mathtt{NEN3D}}$.
3. `For` k`=1,NEN3D,` ℓ`=1,NEN3D do:`

   $T_{k\ell} := D_k D_\ell^T$, where $D_k = \nabla\varphi_k(y_j) \in \mathbb{R}^3$ is the k-th column of D,
   $S_{k\ell} := S_{k\ell} + \big(\lambda T_{k\ell} + \mu T_{k\ell}^T + \mu \cdot \operatorname{tr}(T_{k\ell}) I\big)\, \omega_j \cdot |\det J|$

4. Calculation of $y_j = \sum_{i=1}^{\mathtt{NEN3D}} x_i \hat\varphi_i(\hat y_j)$.
5. `For` k`=1,NEN3D do:`

   $$P_k := P_k + f(y_j)\hat\varphi_k(\hat y_j)\omega_j \cdot |\det J|,$$

   where $f(y_j) \in \mathbb{R}^3$ is calculated in the function routine.

4.3.4 The surface integrals for inhomogeneous Neumann boundary conditions

The subroutine `E3RSOB` calculates the element right hand side $P = (P_k)_{k=1}^{\mathtt{NEN2D}}$, $P_k \in \mathbb{R}^{\mathtt{NDF}}$, for inhomogeneous Neumann boundary conditions, that means:

$$\begin{aligned} p_k &= \int_G g(x)\varphi_k(x)\, dx = \int_{\hat G} \hat g(\hat x)\hat\varphi_k(\hat x)\sqrt{E(\hat x)C(\hat x) - F^2(\hat x)}\, d\hat x \\ &\approx \sum_{i=1}^{\mathtt{NQP2}} \hat g(\hat y_j)\hat\varphi_k(\hat y_j)\sqrt{E(\hat y_j)C(\hat y_j) - F^2(\hat y_j)}\,\omega_j \end{aligned}$$

where

$$\begin{aligned} E(\hat x) &= \sum_{m=1}^{3}\left(\frac{\partial x^{(m)}}{\partial \hat x^{(1)}}\right)^2 = \sum_{m=1}^{3}\left(\sum_{i=1}^{\mathtt{NEN2D}} x_i^{(m)}\frac{\partial\hat\varphi_i(\hat x)}{\partial\hat x^{(1)}}\right)^2, \\ C(\hat x) &= \sum_{m=1}^{3}\left(\frac{\partial x^{(m)}}{\partial \hat x^{(2)}}\right)^2 = \sum_{m=1}^{3}\left(\sum_{i=1}^{\mathtt{NEN2D}} x_i^{(m)}\frac{\partial\hat\varphi_i(\hat x)}{\partial\hat x^{(2)}}\right)^2, \\ F(\hat x) &= \sum_{m=1}^{3}\frac{\partial x^{(m)}}{\partial \hat x^{(1)}}\frac{\partial x^{(m)}}{\partial \hat x^{(2)}} = \sum_{m=1}^{3}\left(\sum_{i=1}^{\mathtt{NEN2D}} x_i^{(m)}\frac{\partial\hat\varphi_i(\hat x)}{\partial\hat x^{(1)}}\right)\left(\sum_{i=1}^{\mathtt{NEN2D}} x_i^{(m)}\frac{\partial\hat\varphi_i(\hat x)}{\partial\hat x^{(2)}}\right). \end{aligned}$$

Thus a loop over all quadrature points $\hat y_j$ is performed with the following steps:

1. Calculation of $J = J(\hat{y}_j) = (J_{mn})_{m=1,n=1}^{3,2}$, with $J_{mn} = \sum_{i=1}^{\mathtt{NEN2D}} x_i^{(m)} \frac{\partial \hat{\varphi}_i}{\partial \hat{x}^{(n)}}(\hat{y}_j)$.

2. Determining $E = \sum_{m=1}^{3} J_{m1}^2$, $C = \sum_{m=1}^{3} J_{m2}^2$, and $F = \sum_{m=1}^{3} J_{m1}J_{m2}$.

3. Calculation of $y_j = \sum_{i=1}^{\mathtt{NEN2D}} x_i \hat{\varphi}_i(\hat{y}_j)$.

4. $P_k = P_k + g(y_j)\hat{\varphi}(\hat{y}_j)\sqrt{E(y_j)C(y_j) - F^2(y_j)}\,\omega_j$, where $g(y_j) \in \mathbb{R}^{\mathtt{NDF}}$. `NDF` is the number of the degrees of freedom.

4.3.5 The coarse grid matrix

For the coarse grid solver there is a coarse grid matrix necessary. Theoretically [7], this matrix shall be obtained by accumulating

$$M = \begin{pmatrix} I_{\mathtt{NDF}} & -I_{\mathtt{NDF}} \\ -I_{\mathtt{NDF}} & I_{\mathtt{NDF}} \end{pmatrix}$$

over all edges of the coarse grid. $I_{\mathtt{NDF}} \in \mathbb{R}^{\mathtt{NDF}\times\mathtt{NDF}}$ is the identity matrix. Moreover, this matrix is used only by processor 0.

Thus we send all 1D-Kettes (for *Kette* see [4]) to processor 0, which assembles then the coarse grid matrix by going through this list of Kettes. This procedure shall be improved in the future because all Kettes which belong to several processors are treated more than once. Note that the coarse grid matrix is stored in its profile structure.

4.4 Calculation and estimation of error norms

4.4.1 The computed values

Error norms are calculated and / or estimated element by element, using quadrature rules as in the assembly of the stiffness matrix / right hand side. That's why these routines are included in the same library *libA.a*.

We will understand under error calculation the elementwise computation of the integrals

$$\begin{aligned} \|u - u_h; H^1(\Omega)\|^2 &:= \int_\Omega |\nabla(u - u_h)|^2\, dx \approx \sum_{j=1}^{\mathtt{NQP3}} \left(\nabla u(x_j) - \sum_{i=1}^{\mathtt{NEN3}} u_i \nabla\varphi_i(x_j)\right)^2 \omega_j, \\ \|u - u_h; L_2(\Omega)\|^2 &:= \int_\Omega (u - u_h)^2\, dx \approx \sum_{j=1}^{\mathtt{NQP3}} \left(u(x_j) - \sum_{i=1}^{\mathtt{NEN3}} u_i \varphi_i(x_j)\right)^2 \omega_j, \end{aligned}$$

where we assume that u and ∇u are programmed functions.

For error estimation a residual type error estimator is under development.

4.4.2 Exact error norms

The exact error norms are computed similarly to 4.3.1 using the subroutine `FEHLER`. The main steps are the allocation and calculation of the arrays `QGST3` and `SHP3`, see Section 4.2, and a loop over the elements. In this loop, the local error contributions are computed, using the subroutines `ELNORM` and `ELN`, and added to a global norm. In `ELNORM` the different element types are distinguished and `ELN` does the actual computations.

By analogy to 4.3.2 / 4.3.3, the local terms are calculated by a loop over the quadrature points, doing:

1. Calculation of J, J^{-1}, and $|\det J|$.
2. Calculation of $D = J^{-T}(\hat{\nabla}\hat{\varphi}_k)_{k=1}^{\mathtt{NEN3D}} = (d_k)_{k=1}^{\mathtt{NEN3D}}, \quad d_k = \nabla\varphi_k(x_j)$.
3. Calculation of $y_j = \sum_{i=1}^{\mathtt{NEN3D}} x_i \hat{\varphi}_i(\hat{y}_j)$.
4. $\mathtt{H1LOC}^2 := \mathtt{H1LOC}^2 + \left(\nabla u(x_j) - \sum_{i=1}^{\mathtt{NEN3D}} u_h(x_i) d_i\right)^2 \cdot |\det J| \cdot \omega_j,$

   $\mathtt{L2LOC}^2 := \mathtt{L2LOC}^2 + \left(u(x_j) - \sum_{i=1}^{\mathtt{NEN3D}} u_h(x_i)\varphi_k(x_j)\right)^2 \cdot |\det J| \cdot \omega_j.$

4.5 Tree structures

```
ASSLOES
    ↪ KLZ_INIT[1]
    ↪ ASSEM
        ↪ VDCOPY[2]
        ↪ MAKEKZU[1]
        ↪ E3LEHF
        ↪ E2INTG
        ↪ E2SHAP
            ↪ PHI2L, PHI2Q, P2L, P2Q
        ↪ E3INTG
        ↪ E3SHAP
            ↪ PHI3L, PHI3Q, P3L, P3Q, PTL, PTQ
        ↪ ELEMENT
            ↪ IHPT
            ↪ ELS, ELAST
                ↪ VDCOPY[2]
                ↪ JACOBIAN
                ↪ F[3]
        ↪ AKKUS, AKKUEL
            ↪ AKKUIJ
        ↪ FAKKU, FAKKUEL
        ↪ AKKUIJ
```

[1] in *libKLZ.a*, see Section 6.1
[2] in *libvbasmod.a*, see Section 6.3
[3] to be supplied in *bsp.f*

```
        ↪ NEUMANN
            ↪ GET_FACE, GET_NEUM
            ↪ E3RSOB
            ↪ G³
        ↪ PACKKLZ¹, SORTKZU¹
    ↪ KETT1DAKK_VOR⁴, KETT2DAKK_VOR⁴, KETT3DAKK_VOR ⁴
    ↪ COARSMAT3
        ↪ TREE_UP
        ↪ ASSCOARS3
            ↪ HDIAGKZU
            ↪ AKKUS, AKKUEL
            ↪ PACKKLZ, SORTKZU
            ↪ CVBKLZ
        ↪ VDCOPY
    ↪ STARTWR3D ⁵, STAVE⁵
    ↪ HB2BPX⁵, HSTH⁵
    ↪ PPCGM⁵
    ↪ GET_TIMES⁶, TIME_GRAF⁶

FNTAB
    ↪ FEHLER
        ↪ E3LEHF
        ↪ E3INTG
        ↪ E3SHAP
            ↪ PHI3L, PHI3Q, P3L, P3Q, PTL, PTQ
        ↪ ELNORM
            ↪ IHPT
            ↪ ELN
                ↪ JACOBIAN
                ↪ U³, UX³, UY³, UZ³
    ↪ CUBE_DOD⁶
```

4.6 Short description of the subroutines

AKKUEL	*akku.f*	accumulates the element stiffness matrix to the global matrix (elasticity)
AKKUS	*akku.f*	accumulates the element stiffness matrix to the global matrix (Poisson)
ASSCOARS3	*coarse.f*	assembles the coarse grid matrix
ASSEM	*assem.f*	assembles the equation system
ASSLOES	*assloes.f*	frame for the assembly and solving the equation system
COARSMAT3	*coarse.f*	frame for ASSCOARS3
E2INTG	*upe2e3.f*	determines integration points and weights, 2D

[4] in *libDDCMcom.a*, see Section 6.5
[5] in *libS.a*, see Section 5
[6] in *libCubecom.a*, see Section 6.2

E2SHAP	*upe2e3.f*	determines the shape functions/derivatives in the integration points, 2D
E3INTG	*upe2e3.f*	determines integration points and weights, 3D
E3LEHF	*upe2e3.f*	allocates memory for the arrays QGST2, QGST3, SHP2, SHP3, S, and P
E3RSOB	*neumann.f*	calculates surface integrals (Neumann boundary conditions)
E3SHAP	*upe2e3.f*	determines the shape functions/derivatives in the integration points, 3D
ELAST	*elast.f*	computes the element stiffness matrix and the right hand side (elasticity)
ELEMENT	*element.f*	frame for ELAST / ELS
ELN	*fehler.f*	computes element errors
ELNORM	*fehler.f*	frame for ELN
ELS	*element.f*	computes element stiffness matrix and the right hand side (Poisson)
F	*fehler.f*	computes the right hand side (equation) in a given point (user supplied)
FAKKU	*akku.f*	accumulates element right hand side to the global vector (Poisson)
FAKKUEL	*akku.f*	accumulates element right hand side to the global vector (elasticity)
FEHLER	*fehler.f*	frame for calculating the error norms
FNTAB	*fehler.f*	calls FEHLER and prints the errors
G	*neumann.f*	computes right hand side (boundary condition) in a given point (user supplied)
GET_FACE	*neumann.f*	extracts the node numbers and coordinates out of NEUMF and COOR
GET_NEUM	*neumann.f*	extracts Neumann data out of NEUMF
IHPT	*ihpt.f*	integer function, determines whether an element is a hexahedron (1), a pentahedron (2), or a tetrahedron (3)
IVD	*ihpt.f*	integer function, determines whether an face is a quadrilateral (1), or a triangle (2)
JACOBIAN	*element.f*	determines the Jacobian functional matrix J, its inverse J^{-1}, and its determinant for one integration point in an element
NEUMANN	*neumann.f*	frame for E3RSOB
P2L	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (linear triangle)
P2Q	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (quadratic triangle)
P3L	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (linear pentahedron)
P3Q	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (quadratic pentahedron)
PHI2L	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (linear quadrilateral)
PHI2Q	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (quadratic quadrilateral)

PHI3L	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (linear hexahedron)
PHI3Q	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (quadratic hexahedron)
PTL	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (linear tetrahedron)
PTQ	*upe2e3.f*	computes the values of all shape functions/derivatives in a point (quadratic tetrahedron)
U	*bsp.f*	computes the solution u in a point (user supplied)
UX	*bsp.f*	computes the derivative $\frac{\partial u}{\partial x}$ in a point (user supplied)
UY	*bsp.f*	computes the derivative $\frac{\partial u}{\partial y}$ in a point (user supplied)
UZ	*bsp.f*	computes the derivative $\frac{\partial u}{\partial z}$ in a point (user supplied)

Chapter 5

Parallel Preconditioned Conjugate Gradient Method (PPCG)

5.1 The Parallel CG-Method

The PPCG method is realized in the subroutine PPCGM. The parallelization of the PCG-method is based on the non-overlapping domain decomposition in connection with the following two types of data storage:

type I: the solution vector $\underline{u}$ is represented locally on each processor i by the vector $\underline{u}_i = A_i \underline{u}$,

type II: the right hand side vector $\underline{f}$ is represented locally on each processor i by $\underline{f}_i$ with

$$\underline{f} = \sum_{i=1}^{\mathtt{p}} A_i^T \underline{f}_i,$$

where p is the number of processors.

A_i is the super element connectivity matrix of the subdomain Ω_i (located on the i-th processor) with the dimension $\mathtt{N}_i \times \mathtt{N}$ ($\mathtt{N}$ ... number of unknowns of the global problem, $\mathtt{N}_i$ ... number of unknowns on the subdomain $\bar{\Omega}_i$) which maps a global vector $g \in \mathbb{R}^{\mathtt{N}}$ on a local vector $g_i \in \mathbb{R}^{\mathtt{N}_i}$. These data types arise from the assembly of the equation system. The local assembly of the right hand side and the stiffness matrix leads to the additivity of the right hand side (type II) while the solution (type I) is assumed to contain the true values on each subdomain. Starting from this domain decomposition we easily get the parallelization of the CG method which is described in the literature, for example in [11].

Currently, there are three types of preconditioners used in the PCG algorithm: the Jacobi-, the Yserentant-, and the BPX-preconditioner, which will be described in Sections 5.2 – 5.4.

Let us consider some technical details: The solution vector is initialized by the subroutine STAVE. At first, the subroutine assigns the value zero to all initial vector entries. Then the points of the Dirichlet boundary conditions of this vector get their correct boundary values. In order to guarantee a vector of type I, this requires some data exchange.

The subroutine STARTWR3D serves as an interactive input routine for the control parameters of the CG algorithm. The user can choose the options given in Table 5.1, compare also [3, Section 2.4].

Option	Description		
`v`	variant of preconditioning:	`v`=1	Jacobi
		`v`=2	Yserentant without coarse grid solver
		`v`=3	Yserentant with coarse grid solver
		`v`=4	BPX without coarse grid solver
		`v`=5	BPX with coarse grid solver
`i`	`iter`, maximal number of iterations		
`e`	`epsilon`, termination criterion for the relative error norm in the CG algorithm		
`d`	`delta`, scaling factor for the coarse grid matrix		
`z`	control of the amount of screen output, see `ion` in [3, Table 2.1]		

Table 5.1: Control parameters for the solver.

Some specific initializations for the CG method and the preconditioners are realized in the subroutine `PREVOR`. First the subroutine `D_OUT_KLZ` (see [13]) extracts the main diagonal D of the stiffness matrix locally on each subdomain. If the coarse grid solver is used in the preconditioner (Section 5.3 and 5.4) the crosspoint values of the main diagonals of each processors stiffness matrix are sent to processor 0 which modifies the global coarse grid stiffness matrix with respect to the Dirichlet boundary conditions and computes the Cholesky factorization of this matrix.

At the end the subroutine `PREVOR` makes some special initializations depending on the kind of the chosen preconditioner. In particular, the inverse entries of D are stored, because only D^{-1} is used subsequently. Here, the information on Dirichlet boundary conditions is introduced, by setting the inverse of 1.D+40 to zero (see 4.3.1, Step 5).

After finishing the subroutine `PREVOR` the PCG iteration starts.

5.2 The Jacobi preconditioner

The Jacobi preconditioner is the simplest preconditioner of all. It only consists of a multiplication of the residual vector r with the inverse D^{-1} of the main diagonal of the stiffness matrix. This preconditioning is realized by the subroutine `JACOBI`.

After this vector multiplication the subroutine transfers the resulting vector $w = D^{-1}r$ from data type II to data type I using the subroutine `FEMAKK`, see [4]. This necessity follows from the data type structure of the PCG method. Therefore the communication cost of the Jacobi preconditioned CG is the same as that of a unpreconditioned CG, and only $\mathtt{N}_i$ essential arithmetical operations per step are needed on processor i.

The condition number of $C^{-1}K = D^{-1}K$ equals $\mathcal{O}(h^{-2})$ where K is the stiffness matrix of our global problem and h is the discretization parameter, but the performance is better than without preconditioning because the sums of the elements in the rows of the matrix are now nearly equilibrated.

5.3 The Yserentant preconditioner

The Yserentant preconditioner [19] is based on a hierarchy of the finite element meshes. It can be written in the following form:

$$C^{-1} = SS^T.$$

Here, S is the basis transformation matrix which transfers the usual nodal basis to the h-hierarchical basis. For the q-th level we can write $S = S_q = S^q_{q-1} \dots S^2_1$ with

$$\left(S^k_{k-1}\right)_{ij} = \begin{cases} 1 & \text{if } i = j, \quad i,j = 1,2,\dots \mathtt{N}_q \\ \frac{1}{2} & \text{if } j = i_1 \text{ and } j = i_2, \text{ where } P^{(i)} \text{ is the middle point} \\ & \text{between } P^{(i_1)} \text{ and } P^{(i_2)} \text{ which are the end points of an} \\ & \text{edge of a tetrahedron from the mesh } \mathcal{T}_{k-1} \\ 0 & \text{else} \end{cases} \tag{5.1}$$

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

$$C^{-1} = SA_0^{-1}S^T, \quad \text{with}$$

$$A_0 = \begin{cases} \delta LL^T & \text{on the coarse grid,} \\ I & \text{else.} \end{cases}$$

LL^T is the Cholesky decomposition of the matrix C_0, and C_0 is the finite element assembly of $\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$ over all pairs of crosspoints having a common edge in the coarse grid (δ has been found empirically, a good value is 0.1).

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

$$C^{-1} = SD^{-\frac{1}{2}}A_0^{-1}D^{-\frac{1}{2}}S^T \tag{5.2}$$

is helpful. D is the diagonal matrix extracted from the stiffness matrix whose elements are scaled with the mesh size h_i of the level i of the point it belongs to.

While the communication cost of the Yserentant preconditioner is nearly as low as without it, the condition number $C^{-1}K$ 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.

The Yserentant preconditioning is realized by the subroutine `YSERENT`. The transformation with the matrices S and S^T is carried out in the subroutines `HiSmulYser` and `HSTmulYser`, respectively:

Routine	Description
`HiSmulYser(Nfg,Nk,X,Hielis)`	$X = SX$
`HSTmulYser(Nfg,Nk,X,Hielis)`	$X = S^TX$

Here, `Nfg` denotes the number of degrees of freedom on each node, `Nk` is the number of nodes on the subdomain `k`, `X` is the vector of the length `N` $=$ `Nk` $*$ `Nfg`, and `Hielis` is the hierarchical list on the subdomain, which is generated by the mesh refinement procedures, see Chapter 3. `Hielis` is a two-dimensional array of the following form:

array	Description
`Hielis[4,Nk]`	Hielis[1,*] - node number Hielis[2,*] - left father Hielis[3,*] - right father Hielis[4,*] - coefficient

The last coefficient defines the basis transformation matrix S. In our definition (5.1) (and in the most cases) it is $\frac{1}{2}$.

The routine `YSERENT` copies first the residual vector to a working vector w setting the Dirichlet values to zero. Then the multiplication $w = S^T w$ is carried out. Now the resulting vector w is multiplied with $D^{-\frac{1}{2}}$ (therefore in the subroutine `PREVOR` the square root of D is computed).

In the next step we have to transform w from data type II to type I. Here communication is necessary which becomes somewhat complicated if we include a coarse grid solver. Because our coarse grid solver is based on a Cholesky factorization computed by processor 0 in a first communication step all processors have to send their crosspoint values to processor 0. While this processor computes the coarse grid solution the other processors start the communication with respect to their edges and faces. In the last communication step all processors receive their parts of the coarse grid solution. Nevertheless, at the end the amount of communication is only slightly higher than that without any coarse grid solution.

In coincidence with equation (5.2) we compute $w = D^{-\frac{1}{2}} w$ once again and after this $w = Sw$. We set the values at the Dirichlet points in the resulting vector to zero and finish the Yserentant preconditioning step.

5.4 The BPX preconditioner

The BPX preconditioner [8] is also a hierarchical preconditioner. It can be written in the following form:

$$C^{-1} = \hat{S}\hat{S}^T.$$

Here $\hat{S}$ is a transformation matrix which transforms the normal nodal basis of the space V_q into the generating system of the Cartesian product space $V_q^E = V_1 \times V_2 \times \ldots \times V_q$ (with the nodal basis spaces V_i, $V_i \subset V_{i+1}$).

For the q-th level we can write $\hat{S} = \hat{S}_q = \left[\mathcal{I}_1^q \,|\, \mathcal{I}_2^q \,|\cdots|\, \mathcal{I}_{q-1}^q \,|\, I_q\right]$, $\mathcal{I}_j^q = \mathcal{I}_{q-1}^q \mathcal{I}_{q-2}^{q-1} \cdot \ldots \cdot \mathcal{I}_j^{j+1}$ with

$$\left(\mathcal{I}_{k-1}^k\right)_{ij} = \begin{cases} 1 & \text{if } i = j, \quad i,j = 1,2,\ldots \mathtt{N}_{k-1} \\ \frac{1}{2} & \text{if } j = i_1 \text{ and } j = i_2, \text{ where } P^{(i)} \text{ is the middle point} \\ & \text{between } P^{(i_1)} \text{ and } P^{(i_2)} \text{ which are the end points of an} \\ & \text{edge of a tetrahedron from the mesh } \mathcal{T}_{k-1} \\ 0 & \text{else} \end{cases} \tag{5.3}$$

If we include a coarse grid solver we get the following for

$$C^{-1} = \hat{S}\hat{A}_0^{-1}\hat{S}^T, \quad \text{with}$$

$$\hat{A}_0 = \begin{cases} \delta LL^T & \text{on the grid of } V_1, \\ I & \text{else.} \end{cases}$$

For δLL^T see Section 5.3.

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

$$C^{-1} = \hat{S}\hat{D}^{-\frac{1}{2}}\hat{A}_0^{-1}\hat{D}^{-\frac{1}{2}}\hat{S}^T \tag{5.4}$$

where $\hat{D}$ is the extracted main diagonal of the stiffness matrix corresponding to V_q^E. Its elements are scaled with the mesh size h_i of the zone i of the point it belongs to.

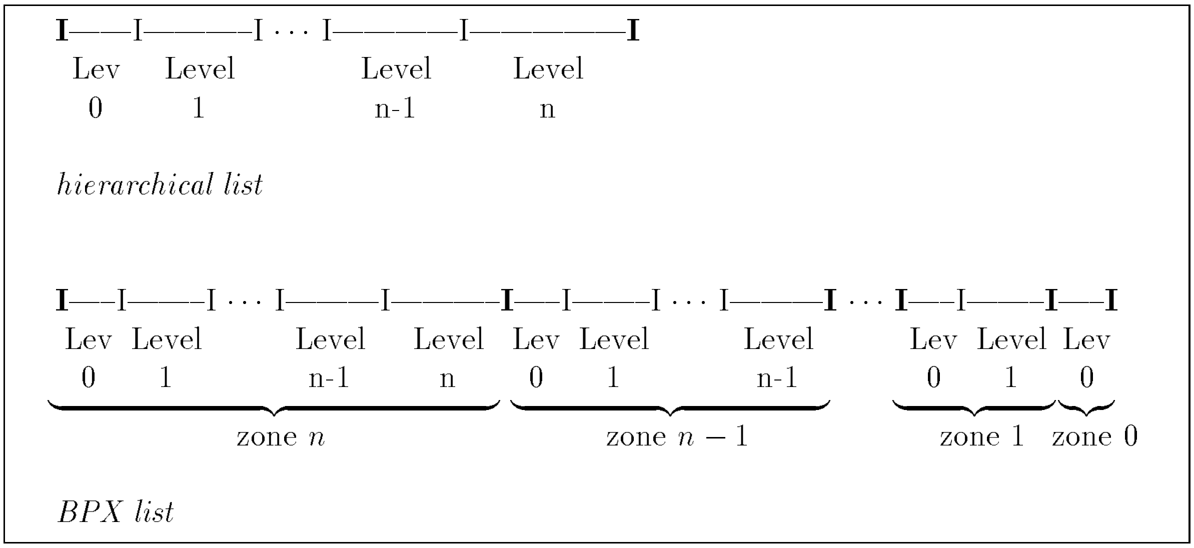

Figure 5.1: List extension for the BPX preconditioner

Due to the fact that we must communicate in the space corresponding to V_q^E the amount of communication data of the BPX preconditioner is higher than that of the preconditioners mentioned before. But at the other hand the condition number of $C^{-1}K$ is $\mathcal{O}(1)$ for the BPX preconditioner.

Before we can use the subroutine `BPXLOES` for the BPX preconditioning some additional initialization steps are necessary. At first we have to extend the hierarchical list in the way shown by Figure 5.1. In the new BPX list zone 0 corresponds with V_1, zone 1 with V_2, ..., zone n with V_q ($q = n + 1$). The length of the BPX list is denoted by `nbpx`. Caused by the communication over all zones we also have to extend the arrays `KETTE1D` and `KETTE2D` with respect to V_i^E, $i = 0, \ldots, q$. These arrays contain some parameters for the communication related to the edges and faces respectively. All this is done by the subroutine `HB2BPX`:

`HB2BPX(N,Lev,nkett,Kett,nLev,zone,help)`		
	input	output
`N`	`Nk`	`nbpx`
`Lev`	hierarchical list	bpx list
`nkett`	length of the `Kett` list	new length of the extended `Kett` list
`Kett`	`Kett` list	extended `Kett` list
`nLev`	—	number of levels
`zone`	—	array of pointers to zones

Keep in mind that before using `HB2BPX` there must be provided enough memory for the extended BPX and `Kett` list. The same applies to the auxiliary vector w in the BPX preconditioner and the vector of the main diagonal of the stiffness matrix (better: its square root $D^{\frac{1}{2}}$) which also have to be extended according to V_q^E.

The application of the transformation matrices $\hat{S}$ and $\hat{S}^T$ is done by the subroutines `HiSmulBPX` and `HSTmulBPX`:

Routine	Describtion
`HiSmulBPX(Nfg,Nk,X,BPXlis)`	$X = \hat{S}X$
`HSTmulBPX(Nfg,Nk,X,BPXlis)`	$X = \hat{S}^T X$

Like the subroutine `YSERENT` the subroutine `BPXLOES` copies first the residual vector to the working vector w setting the Dirichlet values to zero. Then the multiplication $w = \hat{S}^T w$ takes place. As the result we get the extended vector w which is multiplied with $\hat{D}^{-\frac{1}{2}}$ ($\hat{D}$ is the extended main diagonal of the stiffness matrix).

Now we have to transfer w from data type II to type I. This communication concerns all zones. We start with the crosspoint communication where we communicate from the highest down to the lowest zone. If a coarse grid solver is included then after arriving at zone 0 all processors send their crosspoint values of zone 0 to processor 0. While this processor is computing the coarse grid solution the other processors start communication over their edges and faces from zone 1 up to the highest zone. In the last communication step all processors receive their part of the coarse grid solution from processor 0.

Finally we compute $w = \hat{D}^{-\frac{1}{2}} w$ once more and $w = \hat{S} w$ reduces our vector w to the length `Nk`. After inserting the Dirichlet boundary conditions the BPX preconditioning step ends.

5.5 Tree structure of the routines

In the case of a BPX preconditioning the initialization subroutine `HB2BPX` is called in the subroutine `ASSLOES`, see Section 4.5. The PCG method is realized by the subroutine `PPCGM`:

```
PPCGM
  ↪ PREVOR
      ↪ D_OUT_KLZ [1]
      ↪ TREEUP_DOD[2]
      ↪ OXCOPYVBZ
      ↪ CHOVBZ[3]
      ↪ FEM_AKK [4]
      ↪ TREE_DOWN[2]
      ↪ HISCALE3D
      ↪ HSTCOP
  ↪ AXMKLZ[1]
  ↪ VDMINUS[5]
  ↪ JACOBI
      ↪ VDMULT[5]
      ↪ FEMAKK[4]
  ↪ YSERENT
      ↪ VDOMUL[5]
      ↪ HSTMULYSER
      ↪ VDMULT[5]
      ↪ TREEUP_DOD[2]
      ↪ RUEVBZ[3]
      ↪ VORVBZ[3]
      ↪ FEMAKK[4]
      ↪ TREE_DOWN[2]
      ↪ VDMULT[5]
      ↪ HISMULYSER
      ↪ VDOMUL[5]
  ↪ BPX
      ↪ VDOMUL[5]
      ↪ HSTMULBPX
      ↪ VDMULT[5]
      ↪ TREE_DOD[2]
      ↪ TREEUP_DOD[2]
      ↪ RUEVBZ[3]
      ↪ VORVBZ[3]
      ↪ KETTAKK[4]
      ↪ TREE_DOWN[2]
      ↪ VDMULT[5]
      ↪ HISMULBPX
      ↪ VDOMUL[5]
  ↪ DSCAPR[5]
  ↪ TREE_DOD[2]
  ↪ VDAXPY[5]
  ↪ AXMKLZ[1]
  ↪ DSCAPR[5]
  ↪ TREE_DOD[2]
  ↪ VDAXPY[5]
  ↪ ZWISCH
```

[1] in *libKLZ.a*, see Section 6.1
[2] in *libCubecom.a*, see Section 6.2
[3] in *libMbasmod.a*, see Section 6.4
[4] in *libDDCMcom.a*, see Section 6.5
[5] in *libvbasmod.a*, see Section 6.3

5.6 Description of the routines

The following FORTRAN sources are located in the subdirectory *./solve*.

`BPX`	*bpx.f*	BPX preconditioning
`HB2BPX`	*hb2bpx.f*	extends the hierarchical and the `KETT` lists with respect to the BPX data structure
`HISCALE3D`	*hiemul.f*	scaling of the main diagonal elements with the mesh size of the corresponding zone
`HISMULBPX`	*hiemul.f*	multiplication with the transformation matrix $\hat{S}$
`HISMULYSER`	*hiemul.f*	multiplication with the transformation matrix S
`HSTCOP`	*hiemul.f*	extends the main diagonal with respect to the BPX data structure
`HSTMULBPX`	*hiemul.f*	multiplication with the transformation matrix $\hat{S}^T$
`HSTMULYSER`	*hiemul.f*	multiplication with the transformation matrix S^T
`JACOBI`	*jacobi.f*	Jacobi preconditioning
`PPCGM`	*ppcgm.f*	parallel preconditioned conjugate gradient method
`PREVOR`	*prevor.f*	initializations depending on the kind of the chosen preconditioner
`YSERENT`	*yserent.f*	Yserentant preconditioning
`ZWISCH`	*zwisch.f*	displays the values of the CG parameters

Chapter 6

General libraries

6.1 *libKLZ.a* for matrix operations with compactly stored matrices

Discretization methods as the finite element method lead to systems of equations with sparse matrices. For an economic use of the memory of the computer and for a decrease of the number of arithmetic operations, it is necessary to use special storage techniques to exploit the sparsity of the matrix. We use the most economic way, namely to store only the non-zero elements of the matrix.

This method has been implemented and used in different applications for many years at the *Technische Universität Chemnitz-Zwickau*, and the most important routines are available in the library *libKLZ.a*. A description of the storage and of the routines in *libKLZ.a* is given in [13].

6.2 *libCubecom.a* with basic communication routines

MIMD parallel computers consists of a number of processors with local memory which exchange data via inter-processor communication. Unfortunately, the system routines for the communication are dependent on the hardware and the operating system. Thus it is very comfortable from the point of view of the programmer to use standardized communication routines which are independent of the system.

The library *libCubecom.a* consists of a number of routines for the typical communications within a MIMD parallel computer. The consequent use of these routines leads to portability of parallel programs. Only some basic routines have to be adapted. This portability has been tested for transputer, nCube, KSR-1 (TCGMSG), Paramid i860, Linux, and workstation cluster of different producers. A detailed description of the library is given by the developers in [9].

6.3 *libvbasmod.a* with basic vector operations

The library *libvbasmod.a* contains routines for vector operations. The use of these routines instead of `DO`-loops has three advantages:

- The program becomes clearer and better readable.

- The realization of the operation is optimized (unrolled loops, usage of BLAS1-routines).
- Optimizations can be done once, but eventually also system dependent.

A description of the routines and examples for useful applications are given in [9].

6.4 *libMbasmod.a* for matrix routines

This library provides a couple of matrix-oriented subroutines for local operations on each processor or on single processor machines.

There are matrix-vector operations for a special kind of matrix storing schemes (called "VBZ", storing a profile with row-wire variable bandwidth), a set of subroutines for Fast Fourier Transform and some routines for Schur complement preconditioners, see [10]. A more detailed description should be published later.

6.5 *libDDCMcom.a* for DD and coarse matrix routines

The Domain Decomposition Coarse Mesh COMmunication library does what its name says. The kernel of this library is a set of subroutines for FEM accumulation via an effective communication across the coupling edges (and coupling faces in the 3D case).

6.6 The libraries *libGraf.a* and *libNoGraf.a*

The library *libGraf.a* contains routines for simple graphical output which are described in [9]. Based on them, some specific routines are included for graphical output of two- [16] and three-dimensional [14] finite element applications (including meshes and isolines).

The only subroutine which is directly related to *SPC-PMPo 3D* is `GEBGRAPE`, which is the interface to the package GRAPE [18]. The call of this routine, the idea of the interface and an example are described in [14]. Note, that the user has to supply a subroutine `GETDOFS` for a correct labeling of the buttons which are related to the degrees of freedom. For *SPC-PMPo 3D* this is realized in *Netz/Allgemein/getdofs.f*.

Occasionally, the user might wish to gain memory by linking without graphics. For this case the library *libNoGraf.a* may replace the library *libGraf.a* in the link step. It supplies a set of dummy routines, which are called instead of the graphic routines. Set the variable `$GRAF` in *Makedir/makefile.*`$archi` to `Graf` or `NoGraf` to link the desired library, see [3, Section 2.2].

6.7 *libTools.a* for auxiliary routines

The library *libTools.a* contains some auxiliary routines of general interest, most of them for the program-user-dialog and for file and string manipulations, see [9, Section 5]. Examples to get a impression of the routines:

`BEEP(`*n*`)`	Output of an acoustic signal, the integer parameter n determines its length.
`ENTER`	Processor 0 is waiting for an $\boxed{\text{ENTER}}$ and the program continues (after a `tree_down`).
`FLUSH(`*log_unit*`)`	Forces the output of the buffer of *log_unit*. This is essential alter an input request, when the program is started via *rsh* (remote shell).
`LEN_TRIM(`*string*`)`	Returns the length of *string*.
`UPCASE(`*string*`)`	Change of all lower case letters in *string* to upper case.
`YES(`*string*`)`	Output of *string* and input request "`(J/N)`". The return value is `.TRUE.`, when "`J`", "`j`", "`Y`", or "`y`" is entered, and `.FALSE.` in all other cases.

Bibliography

[1] J. Altenbach and A. S. Sacharov. *Die Methode der finiten Elemente in der Festkörpermechanik.* Fachbuchverlag, Leipzig, 1982.

[2] Th. Apel. Programmbeschreibung zum 3D-Finite-Elemente-Programmsystem FEMPS3D. Technical report, TU Chemnitz, 1990. Unpublished.

[3] Th. Apel. *SPC-PMPo 3D* — User's Manual. Preprint SPC95_33, TU Chemnitz-Zwickau, 1995.

[4] Th. Apel, G. Haase, A. Meyer, and M. Pester. Parallel solution of finite element equation systems: efficient inter-processor communication. Preprint SPC95_5, TU Chemnitz-Zwickau, 1995.

[5] Th. Apel and F. Milde. Realization and comparison of various mesh refinement strategies near edges. Preprint SPC94_15, TU Chemnitz-Zwickau, 1994.

[6] J. Bey. Der BPX-Vorkonditionierer in 3 Dimensionen: Gitter-Verfeinerung, Parallelisierung und Simulation. Preprint 3, Universität Heidelberg, IWR, 1992.

[7] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring I–IV. *Math. Comp.*, 1986, 1987, 1988, 1989. 47, 103–134, 49, 1–16, 51, 415–430, 53, 1–24.

[8] J. H. Bramble, J. E. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.

[9] G. Haase, Th. Hommel, A. Meyer, and M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Preprint SPC 95_20, TU Chemnitz–Zwickau, 1995. Updated version of SPC 94_4 and SPC 93_1.

[10] G. Haase, U. Langer, and A. Meyer. The approximate Dirichlet domain decomposition method. Part I: An algebraic approach. Part II: Applications to 2nd-order elliptic boundary value problems. *Computing*, 47:137–151 (Part I), 153–167 (Part II), 1991.

[11] G. Haase, U. Langer, and A. Meyer. Parallelisierung und Vorkonditionierung des CG-Verfahrens durch Gebietszerlegung. In G. Bader, R. Rannacher, and G. Wittum, editors, *Numerische Algorithmen auf Transputer–Systemen*, pages 80–116, Stuttgart, 1993. Teubner. Proceedings of the GAMM–Seminar Heidelberg, 1991.

[12] G. Kunert. Ein Residuenfehlerschätzer für anisotrope Tetraedernetze und Dreiecksnetze in der Finite-Elemente-Methode. Preprint SPC 95_10, TU Chemnitz–Zwickau, 1995.

[13] A. Meyer and M. Pester. Verarbeitung von Sparse-Matrizen in Kompaktspeicherform (KLZ/KZU). Preprint SPC 94_12, TU Chemnitz–Zwickau, 1994.

[14] M. Meyer. Grafik-Ausgabe vom Parallelrechner für 3D-Gebiete. Preprint SPC 95_4, TU Chemnitz–Zwickau, 1995.

[15] I. P. Mysovskikh. *Interpolating cubature formulae.* Nauka, Moskow, 1981. (Russian).

[16] M. Pester. Grafik-Ausgabe vom Parallelrechner für 2D-Gebiete. Preprint SPC 94_24, TU Chemnitz-Zwickau, 1994.

[17] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[18] A. Wierse and M. Rumpf. GRAPE – Eine objektorientierte Visualisierungs- und Numerikplattform. *Informatik Forsch. Entw.*, 7:145–151, 1992.

[19] H. Yserentant. Über die Aufspaltung von Finite-Elemente-Räumen in Teilräume verschiedener Verfeinerungsstufen. Habilitationsschrift, RWTH Aachen, 1984.

[20] O. C. Zienkiewicz. *Methode der finiten Elemente.* Fachbuchverlag, Leipzig, 1983.

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.