Technische Universität Chemnitz

Sonderforschungsbereich 393

Numerische Simulation auf massiv parallelen Rechnern

Thomas Apel Frank Milde Uwe Reichel

SPC-PM Po 3D v4.0

—

Programmer's Manual (Part II)

Preprint SFB393/99-37

Acknowledgement. The package *SPC-PM Po 3D* has been developed in the *Sonderforschungsbereich 393* of the *Technische Universität Chemnitz* under the supervision of A. Meyer and Th. Apel. Other main contributors are G. Globisch, G. Kunert, D. Lohse, F. Milde, M. Pester, U. Reichel, and M. Theß.

Preprint-Reihe des Chemnitzer SFB 393

SFB393/99-37 **December 1999**

Authors' addresses:

Thomas Apel
Technische Universität Chemnitz
Fakultät für Mathematik
09107 Chemnitz
Germany

email: na.apel@na-net.ornl.gov

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

email: milde@physik.tu-chemnitz.de

Uwe Reichel
Technische Universität Chemnitz
Fakultät für Mathematik
09107 Chemnitz
Germany

email: reichel@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.

The historical roots of the program are on the one hand in several parallel programs for solving problems over two dimensional domains using domain decomposition techniques. These codes have been developed since about 1988 by A. Meyer, M. Pester, and other collaborators. On the other hand, Th. Apel 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.

The here documented version 4.x of *SPC-PMPo 3D* includes major changes to the already documented versions 2/3 [2, 3]. The new features are *adaptive mesh refinement* (F. Milde), *error estimation* (G. Kunert), and *dynamic load balancing* (U. Reichel). In difference to the previous versions the new adaptive code can only handle tetrahedral meshes. An adaptive version for hexahedral meshes is planned.

For an introduction of the capabilities of the program, its installation and utilization we refer to the User's Manual for the last version [3]. The aim of this new Programmer's Manual for version 4 is to provide a description of the new data structures and to introduce new routines. It is written for those who are interested in a deeper insight into the code, for example for improving and extending. The paper is not intended as a stand alone version, but as an update and extension to [2].

The documentation is organized as follows: In the next section we describe the boundary value problems that can be solved and the finite elements that are used. Chapter 2 is concerned with the changed data structure. In Chapters 3, 4 and 5 we describe the adaptive mesh generation, the assembly and the solving of the system of equations, respectively. Chapter 6 deals with the memory management routines, a library which should be used also in other programs. Chapter 7 is devoted to the communication routines. After the description of auxiliary routines and other tools in Chapter 8 we end this manual with an explanation of a schematic program run. We point out that there is an index at the end where all routines, parameters and variables are included.

In this documentation we use *slanted style* for real existing paths and filenames, *italic 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 are treated only by the refinement procedure.

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

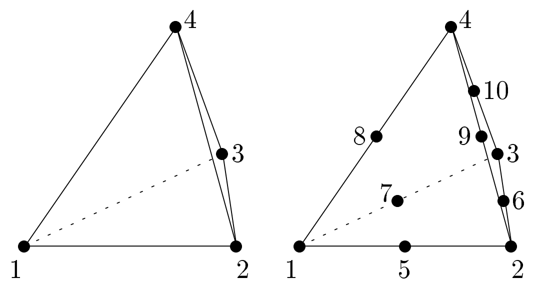


Figure 1.1: Finite elements implemented in *SPC-PMPo 3D* version 4.

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 developed our own memory management and must take care of calculating these pointers to avoid overlaps. For an easier handling the *SPC-PMPo 3D* package provides now a large set of functions and routines for the memory management, see Chapter 6.

Because of the adaptive mesh refinement we use now only the full data structure (FDS, see 2.2) with its greater variability. The reduced data structure (RDS), known from the previous versions, does no longer exist.

Another major change to previous versions is the distinction of the *user mesh*, which is read in from the net file, and the *main mesh*, which could be fixed later. So the notation *coarse mesh* belongs either to the user mesh or the main mesh depending on the program state. For a description in more detail see Subsection 3.2.1 and Chapter 9.

There are a few general variables:

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

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 files *net3ddat.inc* and *adapt.inc*. It is better to use these variables instead of hard numbers because of possible evolution of the data structure.

To maintain compatibility to previous versions all changes of offset pointers and array dimensions in version 4.x are made in *adapt.inc*, see 2.3.6.

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 structure. To save memory capacity we keep always only the fine mesh in the following way. After an adaptive refinement step the parent volume/face/edge is replaced by its children. So all children of a coarse volume/face/edge are stored in one row. This simplifies the handling of coupling edges and faces.

For an illustration a short example: In the coarse mesh the data of volume i is stored between the data of volume $i-1$ and $i+1$. Now, only volume i is full refined into 8 pieces. Then between the former volumes $i-1$ and $i+1$ all 8 children of volume i are stored.

2.2.1 Volumes

1. Each volume is described by its 4 faces, a type, including the volume type and a coarse element number, and the material name.
2. `VOL(DIMVOL,*)` : `DIMVOL`=4+2
3. $\overbrace{\big|\ \text{face\_1}\ \big|\ \text{face\_2}\ \big|\ \cdots\ \big|}^{\text{4 faces}}\ \text{type}\ \big|\ \text{name\_of\_material}\ \big|$

   Face_i is a face number.
4. The 32 Bit value type is shared by two 16 Bit values, volume_type and number_of_coarse_volume. It is not recommended to read/modify these values directly, but to use the provided routines and functions, see 8.3. The volume_types are used for the refinement, possible values are:

0	red refined or not refined
1–6	green1 refined; number of refined edge
7–9	green2 refined; lowest number of refined edges + 6
10–13	green3 refined; number of refined face + 9

   All green refinements are necessary to avoid hanging nodes. For an illustration see Figure 2.1.

   The value name_of_material is a pointer into the material parameter list or the name of a hard coded material.

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

`VOL_TYP`	position of the type in `VOL`	currently $4+1$
`VOL_REGION`	position of the name_of_material	currently $4+2$

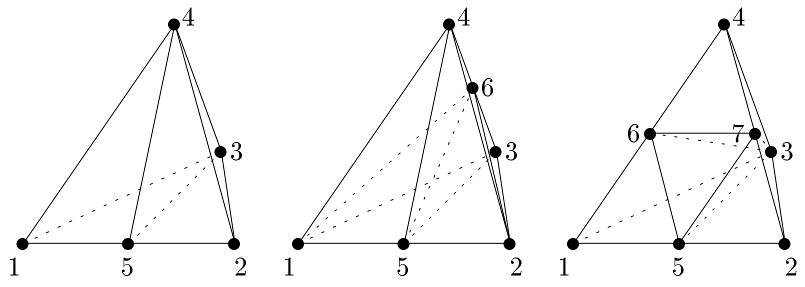


Figure 2.1: Refinement types for tetrahedra. Left: green1, one edge to refine; Middle: green2, two opposite edges to refine; Right: green3, two/three edges of one face to refine.

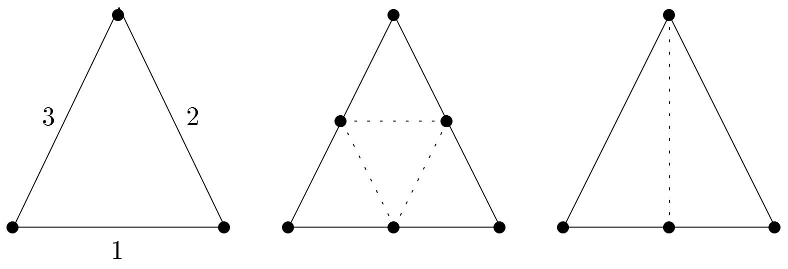


Figure 2.2: Possible face refinements. Left: not refined faces with local edge numbering; Middle: red refined face; Right: green refined face (sample for type 1 or 4).

2.2.2 Faces

1. Each face is described by its 3 edges.
2. `FACE(DIMFACE,*)` : `DIMFACE` $= 3 + 2$
3. $\overbrace{\mid \text{edge\_1} \mid \text{edge\_2} \mid \cdots \mid}^{\text{3 edges}}$ type | data |

   Edge_i is a edge number
4. The value type is a pointer in the `GEOM` array and determines the geometry of the face, see also Subsection 2.2.11. The value data is used twice. Positive values refer to a refinement type, negative values are the number_of_first_son of full (red) refined faces. This is only used during refinement within temporary stored father faces. Possible refinement types are:

< 0	pointer to the first son of this (red) refined face
0	not refined
1–3	green refined outer face; number of refined edge
4–6	green refined inner face; number of refined edge + 3

   For an illustration see Figure 2.1 and Figure 2.2.

   The faces are sorted in the following way:

   | coupling faces | other faces |

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

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

`FZEIG`	position of the type in `FACE`	currently $3+1$
`FCHIELD`	position of the data	currently $3+2$

2.2.3 Edges

1. Each edge is described by its 2 vertices and the middle node (quadratic case only).
2. `KANTE(DIMKANTE,*)` : `DIMKANTE` $= 5$
3. | vertex_1 | vertex_2 | middle node | type | data |
   Vertex_i is a vertex number.
4. The value type is used for the face types of the face(s) containing this edge. The at most 2 face types are stored in the upper and lower 16 bits of the 32 bit value type. This is done by the routine `KA_CODE`, see 8.3.1.

   The value data is shared by the refinement type of the edge and its refinement depth. To set these values the routine `SET_KCHIELD` should be used, see 8.3.1. Possible values for the refinement type are:

< 0	pointer to the first son of this (red) refined edge
0	red/none refined
1	green edge of a green1 volume
2	green inner edge of a green2 volume

   Already red refined edges exist temporary during the refinement at the end of the edge array. Their type value is the name of their first son marked with a negative sign.

   The coupling edges are located at the beginning followed by the edges of coupling faces and the inner edges.

   | coupling edges | edges on coupling faces | inner edges |

   By our definition, all edges of the coarse mesh and their sons are coupling edges even if they do not belong to 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 data	currently 5

2.2.4 Coordinates of the nodes

1. Each node is represented by its three Euclidean coordinates.
2. `COOR(D_COOR,*)` : `D_COOR` $= 4$
3. | X_i | Y_i | Z_i | fatherhood |

4. The 32 bit value fatherhood is used bitwise to mark the fatherhood of a node in the corresponding refinement level. Note that the fatherhood dependencies are reset after defining the main mesh, and from this time on the main mesh is level 1. Although this restricts the maximum count of refinement step to 32 there are some other limitations. The parameter `MAX_ALEV` , currently set to 25 in *adapt.inc*, is the official and only checked restriction. But reaching this limit could already cause problems, because of the only 25 mantis bits of the `REAL*4` node coordinates.

   The nodes are placed in `COOR` in the following way:

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

   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 quite complicated. It is shown in all details with an non-adaptive example in [2], Section 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` $= 1 + \texttt{NDF} * (1 + \texttt{NDIRREAL})$
   `NEUM(DVNEUM,*)` : `DVNEUM` $= 1 + \texttt{NDF} * (1 + \texttt{NNEUMREAL})$

3. $$\mid \text{name\_of\_kette} \mid \overbrace{\text{type, data (DF\_1)} \mid \text{type, data (DF\_2)} \mid \cdots}^{\texttt{NDF}\text{ data blocks}} \mid$$
   DF_k means the k-th degree of freedom.

4. The boundary condition arrays are build twice. First when the user mesh is read in and second when the main mesh is fixed. In difference to the previous versions of *SPC-PMPo 3D* only boundary conditions of the coarse mesh are stored. The value name_of_kette points now in the array `Kette2D`, which holds the information about the refined faces of the coarse face.

   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 *./Bsp/bsp.f*)

$RP(5)$ has been planned for the coefficient in boundary conditions of 3rd 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.

   In version 4.x of *SPC-PMPo 3D* kettes hold also information about edge and faces. To maintain compatibility the first 7 entries in the kette arrays are the same as in previous versions, but now their width is 16!

   There are two different kette data (`KETTE1D` for edges and `KETTE2D` for faces) which have the same data structure. For more information see [1].

2. `KETTE1D(KETDIM,*)/KETTE2D(KETDIM,*)` : `KETDIM` = 16 For compatibility the old parameters `K1DDIM` = `K2DDIM` = 7 in net3ddat.inc exist further, but must not be used!

3. $$\Big|\ \text{pointer}\ \Big|\ \text{length}\ \Big|\ \text{pathID}\ \overbrace{\Big|\ \text{vertex\_1}\ \Big|\ \text{vertex\_2}\ \Big|\ \cdots\ \Big|}^{\text{2 or [3|4] vertices}}\ \text{extended info}\ \cdots\ \Big|$$
   Vertex_i is a vertex number.

4. Note that the vertex numbers here are global (crosspoint) names. For an explanation of pathID see [1].

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

`K_COFF` = `PKZEIG`	position of the pointer to `COOR`	currently 1
`K_CLEN` = `PKLENG`	position of the length in `COOR`	currently 2
`K_WID` = `PWEGID`	position of the pathID	currently 3
`K_NOD1` = `PKDAT`	position of the data (first node)	currently 4
`K_NOD2`	position of the data (second node)	currently 5
`K_NOD3`	position of the data (third node)	currently 6
`K_NOD4`	position of the data (fourth node)	currently 7
`K_FOFF`	position of pointer to `FACE` (first son)	currently 8
`K_FLEN`	position of length in `FACE`	currently 9
`K_KOFF`	position of pointer to `KANTE` (first son)	currently 10
`K_KLEN`	position of length in `KANTE`	currently 11
`K_NIK`	number of new edges (internal use only)	currently 12
`K_NEWC`	number of new nodes (internal use only)	currently 13
`K_O_BPX`	position of pointer to BPX list `LC`	currently 14
`K_L_BPX`	position of length in BPX list `LC`	currently 15
`K_T_BPX`	not used yet but reserved	currently 16

In the KETTE1D array the space of column K_FOFF and K_FLEN remains unused.

The kette offset parameters K_* and the parameter KETDIM are defined in *adapt.inc*. The parameters K1DDIM, K2DDIM, PKZEIG, PKLENG, PWEGID, and PKDAT, defined in *net3ddat.inc* exist only for compatibility.

The number of kettes is NanzK which is the sum of NanzK1D and NanzK2D which are the number of 1D and 2D kettes, respectively. An important fact is that the two kette arrays are always stored in a continuous way, such that KETTE2D follows immediately KETTE1D. Therefore it is possible to refer to both kettes as KETTE. At the end of KETTE2D exists an additional line with the following data:

Position	Value
KETTE[K_COFF,NanzK+1]	KETTE[K_COFF,NanzK]+ KETTE[K_CLEN,NanzK]
KETTE[K_FOFF,NanzK+1]	KETTE[K_FOFF,NanzK]+ KETTE[K_FLEN,NanzK]
KETTE[K_KOFF,NanzK+1]	KETTE[K_KOFF,NanzK]+ KETTE[K_KLEN,NanzK]
KETTE[K_O_BPX,NanzK+1]	KETTE[K_O_BPX,NanzK]+ KETTE[K_O_BPX,NanzK]

This gives the name of the first inner node, face, and edge and is only for internal use.

2.2.8 CHAIN

The CHAIN array, known from previous versions, does not longer exist. It data is now integrated in the KETTE2D data.

2.2.9 REGION

The REGION array, known from previous versions, is still supported by the mesh reading routines but does not exist in this version of *SPC-PMPo 3D*. The material index for each volume, formerly stored in this array, is now integrated in the VOL array as column VOL_REGION.

2.2.10 Hierarchical List

1. The hierarchical list connects all nodes with its father nodes.
2. LC(LC_LEN,*) : LC_LEN = 3 + LC_DAT = 9; LC_DAT = 6
3. $| \text{node} | \text{father\_1} | \text{father\_2} \overbrace{| \text{data} | \cdots |}^{\texttt{LC\_DAT}}$
4. In case of the Yserentant preconditioner only the first position of data (fourth of LC) is used for a factor. The factor ($0 < \text{factor} < 1$) describes the relative position of the node at the edge:

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

In case of the BPX preconditioner 6 entries are made in the data space. The use of the 6 data values during the BPX is described in 5.4.

The entries in LC are ordered such that the fathers are included before their sons. Furthermore LC is ordered level-wise, see also 2.2.13. Note that the entries in COOR are ordered in another way, see [2] Section 3.3. Note that father_1 = father_2 = 0 if the node is a crosspoint.

2.2.11 Geometry data

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

2. `GEOM(DIMGEOM,*)` : `DIMGEOM` = 9

3. | kind_of_face | data_1 | $\cdots$ | data_8 |

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

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

   For more details see [8]. the required parameters for the geometric correction are stored in data_1 to data_8.

2.2.12 X

1. The array `X` stores the previous solution. It is used to compute a good start vector for the CG in the adaptive mesh refinement.

2. `X(*)`

3. | solution for node |

2.2.13 AZONE

1. `AZONE` stores the name of the first and the last node of a refinement level in `LC`.

2. `AZONE(D_AZONE,MAX_ALEV)` : D_AZONE = 2; MAX_ALEV = 25

3. | name_of_first_node | name_of_last_node |

2.2.14 DGRAPH

1. `DGRAPH` stores all data needed for the dynamic load balancing. The data structure corresponds with the widely used CSR format for storing sparse graphs. The stored dual graph belongs to the present coarse mesh.

2. `DGRAPH` is a dynamic structure organized as follows:

part	length	description
index vector	`NVOL+1`	pointer to the first data entry for the corresponding volume; the last entry points to the start of the partitioning data
adjacency data	`DGRAPH(NVOL+1)-NVOL-1`	names of neighbored volumes
partition	`NVOL`	number of processor the volume belongs to
volume weights	`NVOL`	partitioning weight for of each volume; the weight is the number of fine volumes belonging to the coarse volume

   Note: Here, the value `NVOL` is the number of Volumes of the main mesh.

3. | index | adjacency data | partition data | volume weights |

4. The structure `DGRAPH` is build by the subroutine `make_dgraph` and can be viewed with the subroutine `print_dgraph`. To get the partition and weight offsets the functions `Get_Part_Off` and `Get_Wgt_Off` (see 8.3.3) should be used. For storing a new partitioning the subroutine `store_partition` exists. All this subroutines and functions are defined in *dgraph.f* from the library *libaNetzA.a*.

2.3 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.

2.3.1 *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 (see 2.2.1)
`DIMFACE`	dimension of the array of faces (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
`CH_DUMMY`	
`LC_LEN`	dimension of the hierarchical list (see 2.2.10)
`LC_DAT`	dimension of the data in hierarchical list (see 2.2.10)

- `/PROT/`

`NProt1`	dimension 1 of the protocol array (see 7.2)
`NProt2`	dimension 2 of the protocol array (see 7.2)
`Protinfo`	info variable used in *SPC-PM CFD*

- `/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) (obsolete)
`DRNODES`	position of the nodes (RDS) (obsolete)
`DRIFG`	position of `IFG` (RDS) (obsolete)
`DRDAT`	position of the data (RDS) (obsolete)
`DVNEUM`	dimension of the array of Neumann data (FDS) (see 2.2.6)
`DRNEUM`	dimension of the array of Neumann data (RDS) (obsolete)
`NRNODES`	position of the nodes (RDS) (obsolete)
`NRDAT`	position of the data (RDS) (obsolete)

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
`DIMGEOM`	dimension of the array of geometric data	currently 9
`K1DDIM`	dimension of the array of 1D kettes (obsolete)	currently 7
`K2DDIM`	dimension of the array of 2D kettes (obsolete)	currently 7
`PKZEIG`	position of the pointer in the kette data (obsolete)	currently 1
`PKLENG`	position of the block length in the kette data (obsolete)	currently 2
`PWEGID`	position of the path identifier in the kette data (obsolete)	currently 3
`PKDAT`	position of the data in the kette data (obsolete)	currently 4
`NDIRREAL`	number of Dirichlet real parameters	currently 4
`NNEUMREAL`	number of Neumann real parameters	currently 5

2.3.2 *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)
`NanzK`	`NanzK1D` + `NanzK2D`
`NanzK1D`	local number of 1D kettes
`NanzK2D`	local number of 2D kettes
`LinkLevel`	auxiliary variable for communication
`NanzK1DG`	global number of 1D kettes
`NanzK2DG`	global number of 2D kettes

The subroutine `COM_PROB` sets most of these variables.

2.3.3 *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 path and *.std*

To input the filename from keyboard and to set these variables the subroutine `SETFILE` is used.

2.3.4 *standard.inc*

This INCLUDE-File contains some program control variables which can be changed with the file *control.adapt* (equivalent to the previously used *control.tet*) without recompiling the program, see [3, Section 2.4].

- `/standard/`

`vertvar`	kind of coarse grid partitioning
`femakkvar`	variant of accumulation of distributed data, see [1]
`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

2.3.5 *trnet.inc*

There are some variables with information concerning the parallel computer, compare [5, Section 3.1].

- `/TrNet/`

`NCUBE`	dimension of the hypercube
`ICH`	number of the processor in the hypercube topology
`NODENR`	internal info when PARIX is used

- `/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

2.3.6 *adapt.inc*

All major changes of data structure parameters and additional definitions are include in the new file *adapt.inc*. Several special COMMON-blocks are defined:

- `/A_NETDIM/`

`VOL_TYP`	position of type of the volume
`VOL_REGION`	position of material index of the volume

  `/A_NETDIM/` provides additional information to `/NETDIM/` from *net3ddat.inc*.

- `/A_STD/`

`MARK_VAR`	kind of marking for adaptive refinement (see 3.3.1)
`MARK_LOG`	choice if marks should be written to a logfile
`TET_ORD`	choice if tetrahedron should be sorted (see 3.7)
`scale`	factor determining the relation between the maximum estimated error per volume and the used bound for marking, only used with `MARK_VAR`=3
`min_verf`	rate of total number of volumes that must be marked ($0 <$`min_verf`≤ 1), only used with `MARK_VAR`=3

  `/A_STD/` provides additional information to `/standard/` from *standard.inc*.

- `/A_BPX/`

`AZONE`	`AZONE` array (see 2.2.13)
`A_NLEV`	deepest reached refinement level, starts with 1 at the main mesh
`NBPX`	length of the (extended) array `LC` within BPX

- `/LOESER/`

`JACOBI`	choice if Jacobi preconditioning should be used
`YSER`	choice if Yserentant preconditioning should be used
`BPX`	choice if BPX preconditioning should be used
`Ivar`	variant of solver
`Delta`	factor for the simplified coarse grid matrix

- `/SEL_COM/`

`LOC_CUBE`	hypercube size actually used
`N_GROB_T`	number of main mesh volumes on processor
`PROCS_FULL`	logical value, true if all processors are used
`SPLIT_WERT`	percentage of memory usage before data split between processors
`N_JE_PROC`	minimum amount of volumes per processor required for fixing the main mesh
`ORG_LOESVAR`	stores the value `loesvar` until the main mesh is fixed
`ORG_MAXADR`	temporary used during the fixing of the main mesh
`N_LC`	length of index vector of the coarse grid matrix
`N_CC`	length of data vector of the coarse grid matrix

Chapter 3

Adaptive mesh generation

3.1 General mesh handling

Unlike previous versions of *SPC-PMPo 3D*, version 4.x constructs the mesh in several steps driven by the solution until the estimated local error of each volume is below a certain bound. Therefore the mesh generation consists also of several steps.

- Read the user mesh data and generate the user mesh,
- adaptive mesh refinement,
- main mesh fixing,
- mesh distribution,
- further adaptive refinement with load balancing.

The mesh generation and the main mesh fixing is done only once, but the mesh distribution and of course the adaptive refinement could happen several times.

3.2 User mesh input, main mesh fixing, and mesh distribution

3.2.1 The procedure

The user mesh is read from a standardized file, compare [3, Section 3.2]. These files are located in the subdirectory *./mesh3* (tetrahedral meshes). Only processor zero reads the mesh and generates the data structure. A number of variables and arrays are initialized with its start values. The user mesh is taken as coarse mesh. 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 sub-meshes of two processors or if they are only within one sub-mesh. That's why the kette arrays are defined despite the fact that all further computation is done only on processor 0. Now the computation starts with solving the problem on the user mesh. After finding the solution there are two possibilities for the program to proceed, compare also Figure 9.1 on page 60.

The way depends on the parameter `N_JE_PROC` read from *control.adapt*. It determines the minimal number of volumes per processor which the main mesh should consist of. For an explanation a short example: If `N_JE_PROC` is 20 and the program runs on 8 processors the mesh on processor 0 must consist of more that 160 volumes before it is fixed as main mesh. If the number of volumes is too small the program proceeds with an adaptive refinement step and computes a new solution.

If the desired mesh size is reached the main mesh is fixed, by making the present mesh the new coarse mesh, and redefining the boundary condition data and kettes. The hierarchy level of the mesh is set to 1 and the present stiffness matrix is stored as coarse grid matrix.

Now the program checks whether the amount of data on processor 0 exceeds the limit set by the parameter `SPLIT_WERT`. This parameter determines the minimal percentage of used memory on a processor before the problem is split and distributed on 2 processors. This splitting is going on until the percentage use of memory on each processor is smaller then `SPLIT_WERT` or all processor have data.

Finally the program asks how to proceed, offering the possibilities to compute a new solution, adaptively refine the mesh, read in a new mesh, quit, or modify program parameters.

3.2.2 Parameters of `NET_0`

The procedure `NET_0` generates the initial mesh. It reads the user mesh from a file and generates the full data structure.

```
SUBROUTINE NET_0(A,JCOOR,NUMNP,JDIR,NDIR,JNEUM,NNEUM,JVOL,
                 NVOL,JKANTE,NKANTE,JFACE,NFACE,JIGLOB,JKETTE1D,
                 JKETTE2D,JLC,JGEOM,NGEOM,JX,JDGRAPH,IER)
```

`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`
`NVOL`	O	Number of VOLumes
`JKANTE`	O	pointer to array of edges `KANTE`
`NKANTE`	O	Number of KANTEs (edges)
`JFACE`	O	pointer to array of faces `FACE`
`NFACE`	O	Number of FACEs
`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`
`JLC`	O	pointer to the hierarchical list
`JGEOM`	O	pointer to array of geometry data `GEOM`
`NGEOM`	O	Number of GEOMetry data sets
`JX`	O	pointer to the solution vector `X`
`JDGRAPH`	O	pointer to the dual graph of the mesh `DGRAPH`
`IER`	O	error indicator of the subroutine

To optimize the communication, the nodes which belong to coupling faces/edges have consecutive numbers 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 arrange the edges and faces in a certain way.

3.2.3 Parameters of SET_GROBNETZ

The procedure SET_GROBNETZ sets the the main mesh by defining a new coarse mesh. It stores the real coarse grid matrix and modifies all relevant arrays, including the boundary conditions, the kettes, and the hierarchical list.

```
SUBROUTINE SET_GROBNETZ(A,JLA,JA,J_GLC,J_GCC,JCOOR,NUMNP,JDIR,NDIR,
                        JNEUM,NNEUM,JVOL,NVOL,JKANTE,NKANTE,JFACE,
                        NFACE,JIGLOB,JKETTE1D,JKETTE2D,JLC,JGEOM,NGEOM,
                        JX,JDGRAPH,JF,IER)
```

A	I/O	workspace vector
JLA	I	index vector of the present stiffness matrix
JA	I	data array of the present stiffness matrix
J_GLC	I/O	index vector of the coarse grid matrix
J_GCC	I/O	data array of the coarse grid matrix
JCOOR	I/O	pointer to array of node coordinates COOR
NUMNP	O	NUMber of Nodal Points
JDIR	I/O	pointer to the Dirichlet data DIR
NDIR	I/O	number of Dirichlet faces
JNEUM	I/O	pointer to the Neumann data NEUM
NNEUM	I/O	number of Neumann faces
JVOL	I/O	pointer to array of volumes VOL
NVOL	I	Number of VOLumes
JKANTE	I/O	pointer to array of edges KANTE
NKANTE	I	Number of KANTEs (edges)
JFACE	I/O	pointer to array of faces FACE
NFACE	I	Number of FACEs
JIGLOB	I/O	pointer to array of global crosspoint names IGLOB
JKETTE1D	I/O	pointer to array of 1D kette data KETTE1D
JKETTE2D	I/O	pointer to array of 2D kette data KETTE2D
JLC	I/O	pointer to the hierarchical list
JGEOM	I/O	pointer to array of geometry data GEOM
NGEOM	I	Number of GEOMetry data sets
JX	I/O	pointer to the solution array
JDGRAPH	I/O	pointer to the dual graph of the mesh
IER	O	error indicator of the subroutine

3.2.4 Parameters of N_SPLIT

The subroutine N_SPLIT manages the mesh distribution to the processors. If the amount of data on a processor is higher then the given bound the routine determines a 'split partner' and tries to divide the data equally and sends one half of the data to the second processor. The present version uses only a linear distribution scheme this will be replaced by a partitioner from the ParMetis library in the next version. Note that only (the children belonging to) coarse mesh tetrahedra are moved between processors.

The data splitting stops, if the data on all processors is below the given bound or if all processors have data.

```
SUBROUTINE N_SPLIT(A,JCOOR,NUMNP,JDIR,NDIR,JNEUM,NNEUM,JVOL,NVOL,
                   JKANTE,NKANTE,JFACE,NFACE,JIGLOB,JKETTE1D,JKETTE2D,
                   JLC,JGEOM,NGEOM,JX,J_GLC,J_GCC,JDGRAPH,SCHWELLE,IER)
```

`A`	I/O	workspace vector
`JCOOR`	I/O	pointer to array of node coordinates `COOR`
`NUMNP`	I/O	NUMber of Nodal Points
`JDIR`	I/O	pointer to the Dirichlet data `DIR`
`NDIR`	I/O	number of Dirichlet faces
`JNEUM`	I/O	pointer to the Neumann data `NEUM`
`NNEUM`	I/O	number of Neumann faces
`JVOL`	I/O	pointer to array of volumes `VOL`
`NVOL`	I/O	Number of VOLumes
`JKANTE`	I/O	pointer to array of edges `KANTE`
`NKANTE`	I/O	Number of KANTEs (edges)
`JFACE`	I/O	pointer to array of faces `FACE`
`NFACE`	I/O	Number of FACEs
`JIGLOB`	I/O	pointer to array of global crosspoint names `IGLOB`
`JKETTE1D`	I/O	pointer to array of 1D kette data `KETTE1D`
`JKETTE2D`	I/O	pointer to array of 2D kette data `KETTE2D`
`JLC`	I/O	pointer to the hierarchical list
`JGEOM`	I/O	pointer to array of geometry data `GEOM`
`NGEOM`	I/O	Number of GEOMetry data sets
`JX`	I/O	pointer to the solution array
`J_GLC`	I	index vector of the coarse grid matrix
`J_GCC`	I	data array of the coarse grid matrix
`JDGRAPH`	I/O	pointer to the dual graph of the mesh
`SCHWELLE`	I	bound for data splitting; defined as `SPLIT_WERT`
`IER`	O	error indicator of the subroutine

3.3 Adaptive refinement

3.3.1 The procedure

In general there are 6 major steps in the adaptive refinement procedure `A_REFINE`:

1. marking of volumes, faces, or edges to refine by several criteria,
2. prediction of all changes resulting from the initial marking, extend the marking of volumes, faces, and edges accordingly (red, green1/2/3, ...),
3. prediction of the expected load imbalance after the refinement and repartitioning if necessary,
4. calculation of the array lengths needed during the refinement and allocation of memory,
5. refinement including green closure of the mesh,
6. memory usage optimization,
7. restart with step 2 if necessary.

Now we explain the steps in more detail.

Marking Adaptive refinement means that only selected volumes get refined. To do so the desired ones have to be marked. The present version of the program offers 3 ways to set marks for the refinement. They are distinguished by the parameter `MARK_VAR`.

The first possibility (`mark_var=0`) asks the user for the names of the volumes to refine. This is especially helpful for development and test reasons.

The second possibility (`mark_var=1`) sets marks on geometrical criteria programmed in *geo_mark.f*. `GEO_MARK` provides the possibility to mark volumes, faces, edges, and any mixture.

The third, and in practice most common way (`mark_var=2`), is the marking based on an error estimator. We use the Zienkiewicz-Zhu error estimator from a library written by G. Kunert, see [6]. The selection of the volumes can be influenced by the two parameters `alpha` and `min_verf`. `min_verf` is the minimal fraction of volumes to refine and `alpha` is the fraction of the maximal estimated error a volume must have to be marked.

$$\text{Mark if} \quad err_{vol} > \alpha \cdot err_{vol,max}.$$

If the number of marked volumes is less than `min_verf*NVOL` then `alpha` is reduced and a new marking is done (`NVOL` is the global number of volumes).

Extend marking The present version of the program can only handle regular meshes, which means in our case a mesh without irregular/hanging nodes. To avoid such irregular nodes a green mesh closure is produced by refining additional volumes in a green1, green2, or green3 way, see Figure 2.1. If more than 3 hanging nodes appear in a volume it is red/full refined into 8 pieces.

The initial marking is extended by the subroutine `SEL_MARK` such that all volumes, faces, and edges to refine (including the green closure) are marked by a number indicating the refinement type (red/green1/2/3, ...). During the run of `SEL_MARK` the consistency over processor borders is kept by a communication over coupling edges.

Load balancing The expected work load of each processor is estimated by the expected amount of volumes on each processor weighted with the computation time the processor has needed in the last step. If the imbalance between processors reaches a certain amount (at the moment 30%) and the amount of communication for re-balancing is less than the gain in computational speed a repartitioning is done.

Prediction Based on the marking the additional amount of memory needed during the refinement step is predicted by counting all marked volumes, faces, and edges. Then the additional memory on the mesh data arrays is allocated.

Refinement The actual refinement (which includes the green closure) is done in 3 steps. It starts with the marked edges, goes on with the marked faces and finishes with the marked volumes. After the refinement some temporarily needed space on the mesh data arrays is freed. The data organization on the mesh data arrays follows the guidelines explained in [2].

In certain cases it is not possible to accomplish the refinement in a single run and the procedure is restarted with the second step. A typical case is the marking of one half of a green edge. In such a case, in the first run, the father of the corresponding green volumes is virtually restored and red refined. Afterwards, in the second run, the green closure in the new subtetrahedrons is performed.

3.3.2 Parameters of A_REFINE

The procedure A_REFINE does the complete adaptive mesh refinement including a possibly necessary load balancing. It takes the present mesh and the corresponding solution and generates the next level.

```
SUBROUTINE A_REFINE(A,JCOOR,NUMNP,JDIR,NDIR,JNEUM,NNEUM,JTET,NUMEL,
                    JKANTE,NKANTE,JFACE,NFACE,JIGLOB,JKETTE1D,JKETTE2D,
                    JLC,JGEOM,NGEOM,JX,VFS,JDGRAPH,RTIMES,L_GROBNETZ,IER)
```

A	I/O	workspace vector
JCOOR	I/O	pointer to array of node coordinates COOR
NUMNP	I/O	NUMber of Nodal Points
JDIR	I/O	pointer to the Dirichlet data DIR
NDIR	I/O	number of Dirichlet faces
JNEUM	I/O	pointer to the Neumann data NEUM
NNEUM	I/O	number of Neumann faces
JVOL	I/O	pointer to array of volumes VOL
NVOL	I/O	Number of VOLumes
JKANTE	I/O	pointer to array of edges KANTE
NKANTE	I/O	Number of KANTEs (edges)
JFACE	I/O	pointer to array of faces FACE
NFACE	I/O	Number of FACEs
JIGLOB	I/O	pointer to array of global crosspoint names IGLOB
JKETTE1D	I/O	pointer to array of 1D kette data KETTE1D
JKETTE2D	I/O	pointer to array of 2D kette data KETTE2D
JLC	I/O	pointer to the hierarchical list
JGEOM	I/O	pointer to array of geometry data GEOM
NGEOM	I/O	Number of GEOMetry data sets
JX	I/O	pointer to the solution array
VFS	I/O	number of refinement steps
JDGRAPH	I/O	pointer to the dual graph of the mesh
RTIMES	I/O	array of computation times for each processor; in: measured time for last solution, out: guessed time for next solution
L_GROBNETZ	I/O	logical parameter determining whether the main mesh is fixed or not
IER	O	error indicator of the subroutine

The data ordering mentioned in 3.2.2 is always kept by the routine.

3.4 Parameters of the output tool AUSGABE

The subroutine AUSGABE is an output tool for several (mesh) data. Features of AUSGABE:

- graphical output of mesh data with GRAPE (3D)
- graphical output of mesh data with gebgraf (2D)
- graphical output of mesh data with Irix Explorer (3D)
- 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(SOLVED,COOR,NUMNP,KANTE,NKANTE,FACE,NFACE,
           VOL,NUMEL,DIR,NDIR,NEUM,NNEUM,KETTE1D,KETTE2D,
           VFS,X,LC,A,IER)
```

`SOLVED`	mesh status
`COOR`	array of node coordinates
`NUMNP`	number of nodal points
`KANTE`	array of edges
`NKANTE`	number of edges
`FACE`	array of faces
`NFACE`	number of faces
`VOL`	array of volumes
`NUMEL`	number of volumes
`DIR`	array of Dirichlet data
`NDIR`	number of Dirichlet faces (on coarse mesh)
`NEUM`	array of Neumann data
`NNEUM`	number of Neumann faces (on coarse mesh)
`KETTE1D`	array of 1D kette data
`KETTE2D`	array of 2D kette data
`VFS`	number of refinement steps
`X`	solution
`LC`	hierarchical list
`A`	workspace array
`IER`	error parameter

All variables except the error parameter `IER` are input.

3.5 Tree structure of the routines

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

3.5.1 `NET_0` for generating the user mesh

```
NET_0
    ↪ DATA_READ
    ↪ SET_RBCOM
    ↪ MEM_CHANGE
    ↪ K_CODES
        ↪ KA_CODE
        ↪ KAC_OPT
    ↪ ZUERST
```

```
          ↪ SET_KCHIELD
          ↪ PCORECT
          ↪ K_LC
          ↪ P_FACE
              ↪ GEMPKT
      ↪ TETORDNEN
          ↪ TESTORDN
              ↪ ECKPUNKTE
                  ↪ GEMPKT
                  ↪ GEMKANTE
      ↪ COM_PROB
```

3.5.2 SET_GROBNETZ for the main mesh fixing

```
SET_GROBNETZ
      ↪ CVBKLZ
      ↪ CHOVBZ
      ↪ GROB_RB0
      ↪ MEM_CHANGE
      ↪ GROB_RB1
      ↪ ZUERST *
      ↪ COM_PROB
      ↪ MAKE_DGRAPH
          ↪ BUILDHV
          ↪ BUILD_DGRAPH
          ↪ PACK_DGRAPH
```

3.5.3 N_SPLIT for distribution of the mesh

```
N_SPLIT
      ↪ GET_SPLIT
      ↪ D_SPLIT
      ↪ RECV_NODE_1
      ↪ SEND_NODE_1
      ↪ S_MARK2
          ↪ GET_GROB_NR
      ↪ T_KUERZ
      ↪ DATREDO
      ↪ POST_FRED
      ↪ POST_KRED
      ↪ POST_CRED
      ↪ MEM_CHANGE
      ↪ SET_COM_KN
```

3.5.4 A_REFINE for the adaptive mesh refinement

```
A_REFINE
      ↪ HCOM_SIZE
      ↪ NO_MARK
      ↪ SET_VMARK
      ↪ GEO_MARK
      ↪ EST_MARK
      ↪ SEL_MARK
      ↪ TET_VOR
      ↪ DREI_VOR
      ↪ K_VOR
```

```
    ↪ REBALANCE
    ↪ VOR_FEIN
    ↪ T_KANTEN
    ↪ T_DREI
    ↪ T_TET
    ↪ SEL_KUERZ
    ↪ SET_COM_KN
  EST_MARK
    ↪ ECKPUNKTE ∗
    ↪ RES_E
        ↪ E3LEHF
        ↪ E3INTG
        ↪ E3SHAP
        ↪ NORMAL_ABL
            ↪ JACOBIAN
            ↪ KREUZPROD
            ↪ NODE2FACE
        ↪ REC_GRADIENT
        ↪ FACE_AKK
        ↪ P_FACE ∗
        ↪ A_GET_XL
        ↪ GET_NEUM
        ↪ P2_GN
        ↪ A_FEMACC
    ↪ EST_ZZ
        ↪ E3LEHF
        ↪ E2INTG
        ↪ E2SHAP
        ↪ E3INTG
        ↪ E3SHAP
        ↪ JACOBIAN
        ↪ GET_CT
        ↪ M1APPROX
    ↪ MARKIEREN
SEL_MARK
    ↪ A_K3AKK_VOR
    ↪ SET_VROT
        ↪ GET_VTYP
    ↪ DREI_MROT
    ↪ A_K3AKK
    ↪ DREIMARK
        ↪ GET_KTYP
        ↪ DREI_MROT
    ↪ TETMARK
        ↪ GET_VTYP
        ↪ GTMARK
        ↪ SET_VROT
TET_VOR
    ↪ GET_VTYP
    ↪ GET_VDEP
        ↪ GET_KDEPTH
    ↪ GET_GROB_NR
DREI_VOR
    ↪ D_VOR_1
        ↪ GET_FDEP
            ↪ GET_KDEPTH
K_VOR
    ↪ K_VOR_1
        ↪ GET_KTYP
        ↪ GET_KDEPTH
REBALANCE
    ↪ COMPUTE_LOADS
    ↪ REPARTITION
        ↪ PARMETIS_REPARTLDIFFUSION
    ↪ COMPUTE_COMMLOADS
    ↪ VOR_TRANSFER
        ↪ SORTIEREN
    ↪ TRANSFER
    ↪ STORE_NEW_PART
VOR_FEIN
    ↪ LC_PLATZ
T_KANTEN
    ↪ GET_KTYP
    ↪ GET_KDEPTH
    ↪ SET_KCHIELD
    ↪ C_RENAME
    ↪ C_UPDATE
    ↪ T_K
        ↪ GET_KTYP
        ↪ GET_KDEPTH
        ↪ K_WRITE
            ↪ SET_KCHIELD
        ↪ PCORECT
        ↪ K_LC
T_DREI
    ↪ T_D
        ↪ GET_KTYP
        ↪ D_WRITE
        ↪ D_GRUEN
            ↪ GEMPKT
            ↪ GET_KDEPTH
            ↪ GET_KTYP
            ↪ PCORECT
            ↪ K_LC ∗
            ↪ K_WRITE ∗
            ↪ D_WRITE
        ↪ D_ROT
            ↪ GET_KDEPTH
            ↪ PCORECT
            ↪ K_LC ∗
            ↪ K_WRITE ∗
            ↪ D_WRITE
        ↪ GET_G_KANTEN
            ↪ GET_KTYP
```

```
T_TET
    ↪ T_T
        ↪ GET_VTYP
        ↪ T_GRUEN1
            ↪ GEMKANTE
            ↪ GET_GROB_NR
            ↪ D_WRITE
            ↪ T_WRITE
                ↪ SET_VTYP
        ↪ G_V_G1
        ↪ T_GRUEN2
            ↪ GEMPKT
            ↪ GET_GROB_NR
            ↪ GET_KDEPTH
            ↪ N_KANTE
                ↪ K_LC *
                ↪ K_WRITE *
            ↪ D_WRITE
            ↪ T_WRITE *
        ↪ T_GRUEN3
            ↪ GET_GROB_NR
            ↪ D_WRITE
            ↪ T_WRITE *
        ↪ G_V_G2
        ↪ G_V_G3
        ↪ T_ROT
            ↪ GEMPKT
            ↪ GET_GROB_NR
            ↪ GET_KDEPTH
            ↪ N_KANTE *
            ↪ D_WRITE
            ↪ T_WRITE *
```

3.5.5 AUSGABE

```
AUSGABE
    ↪ IAUS
    ↪ A_VIS_GRAPE
        ↪ ECKPUNKTE *
        ↪ VCRFROMD
        ↪ GEBGRAPE
    ↪ A_VIS_X11
        ↪ ECKPUNKTE *
        ↪ OLD_KET
        ↪ DRAW3D
    ↪ A_VIS_EXPL
        ↪ ECKPUNKTE *
        ↪ OLD_KET
        ↪ OUT3DEXPL
    ↪ NETZDRUCK
        ↪ GET_VTYP
        ↪ GET_GROB_NR
        ↪ FSTRADDI
        ↪ FSTRADDR
        ↪ VRBPRINT
    ↪ KETPOUT
    ↪ WTABX
        ↪ PWTABX
    ↪ A_FNTAB
        ↪ A_FEHLER
```

3.6 Short description of the routines in *libaNetzA.a*

The following FORTRAN sources are located in *aNetzA*. The library substitutes *libNA.a* from previous versions of the program.

AUSGABE	*ausgabe.f*	frame for the output of several data (mesh, solution, error estimates)
A_VIS_EXPL	*a_visual3d.f*	prepares the data for visualization with the Irix 3D Explorer
A_VIS_GRAPE	*a_visual3d.f*	prepares the data for visualization with GRAPE
A_VIS_X11	*a_visual3d.f*	prepares the data for visualization with 2D X11 interface
A_YSFAKTOR	*cnetz.f*	determines the relative length of the sub-edges (factor in LC)
BUILDHV	*buildhv.f*	accumulation of an auxiliary array for building the dual graph of the mesh
BUILD_DGRAPH	*dgraph.f*	constructs the dual graph of the mesh
COMPUTE_COMMLOADS	*balance.f*	guesses the communication load for a load rebalancing (not implemented yet)

COMPUTE_LOADS	*balance.f*	guesses the computational load of the processors after an adaptive mesh refinement
COM_PROB	*com_prob.f*	sets the variables of the common block in *com_prob.inc*
DATRED	*kuerzen.f*	deletes faces/edges/nodes which are not referred in the volumes/faces/edges; generates IGLOB and deletes unused boundary condition data
D_WRITE	*AUpfein.f*	writes a face in the array of faces
EB2KUG	*pcorect.f*	determines the cut plane of two spheres
ECKPUNKTE	*AUpfein.f*	determines the vertices of a tetrahedron
FSTRADDI	*netzdruck.f*	generates a format string
FSTRADDR	*netzdruck.f*	generates a format string
GEMKANTE	*AUpfein.f*	determines the common edge of two faces
GEMPKT	*AUpfein.f*	determines the common node of two edges
GEOPRINT	*stdwrite.f*	output of the face geometry description
GET_FDEP	*AUpfein.f*	determines the refinement level of a face
GET_GROB_NR	*v_typ.f*	determines the name of the coarse grid volume to which the fine volume belongs to
GET_KDEPTH	*AUpfein.f*	determines the refinement level of an edge
GET_KTYP	*AUpfein.f*	determines the refinement type of an edge
GET_MINFO	*memo.f*	reads array data from info block of the workspace vector
GET_PART_OFF	*dgraph.f*	returns the offset in DGRAPH for the partitioning info
GET_VDEP	*AUpfein.f*	determines the refinement level of a volume
GET_VTYP	*v_typ.f*	determines the refinement type of a volume
GET_WGT_OFF	*dgraph.f*	returns the offset in DGRAPH for the weights
G_MEM_USE	*memo.f*	returns the percentage use of the workspace vector
IAUS	*ausgabe.f*	displays the output menu and returns the user choice
IER_TEST	*ier_set.f*	tests the error indicator IER, displays an error message, and sets IER new
ITAUSCH	*AUpfein.f*	swaps two integer values
KAC_OPT	*geom.f*	optimizes the geometry codes of an edge
KA_CODE	*geom.f*	generates the geometry code of an edge from a given face geometry
KEESCHNITT	*pcorect.f*	computes the coordinates of a node situated on the cut between a cone and a plane
KEGPROJ	*pcorect.f*	computes the coordinates of a node situated on a cone
KESCHNITT	*pcorect.f*	computes the coordinates of a node situated on the cut between a sphere and a plane
KETPOUT	*netzdruck.f*	output of kette data (for kette see [1])
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.)
KUGPROJ	*pcorect.f*	projection of a node onto a sphere
K_CODES	*geom.f*	generates edge geometry codes
K_LC	*AUpfein.f*	writes nodes into LC and computes a start solution for the new edge midpoint
K_WRITE	*AUpfein.f*	writes an edge into the array of edges
LIES	*standard.f*	reads and analyses a row of the file of program control variables (*control.adapt*)
MAKE_DGRAPH	*dgraph.f*	frame for the creation of the data structure DGRAPH

MEMO_INIT	*memo.f*	initializes the workspace vector for the memory management
MEMO_OUT	*memo.f*	displays all data from the info block of the workspace vector
MEMO_USE	*memo.f*	displays the present percentage use of the workspace vector
MEM_CHANGE	*mem_change.f*	executes all changes in memory usage on the permanently used arrays on the workspace vectorand gives the new offsets
MG_NAME	*memo.f*	determines the name string of a given array
MIT3DGRAFIK	*a_visual3d.f*	dummy function .TRUE. for *libGraf.a* and .FALSE. for *libNoGraf.a*
MOVE	*cnetz.f*	realization of a coordinate transformation for special applications
M_CH_COPY	*memo.f*	auxiliary routine for changes on the workspace vector
M_CH_MAIN	*memo.f*	executes memory changes on the workspace vector
M_CH_POST	*memo.f*	auxiliary routine for changes on the workspace vector
M_CH_PRE	*memo.f*	auxiliary routine for changes on the workspace vector
M_CH_TEST	*memo.f*	auxiliary routine for changes on the workspace vector
M_CH_VAL	*memo.f*	auxiliary routine for changes on the workspace vector
M_DEL	*memo.f*	deletes an array from the info block of the workspace vector
M_D_OUT	*memo.f*	auxiliary routine for MEMO_OUT
M_FREE_GET	*memo.f*	returns the first free address on the workspace vector
M_H_OUT	*memo.f*	auxiliary routine for MEMO_OUT
M_LENG	*memo.f*	auxiliary routine, returns the maximal size of an array of a given amount of memory (bytes)
M_NAME	*memo.f*	returns the name of an array by its number
M_NEW	*memo.f*	returns the start address of a new array on the workspace vector
M_N_FREE	*memo.f*	returns the first free address on the workspace vector possible for the given array type and gives the maximal array size for this type
M_OFF_END	*memo.f*	auxiliary routine for changes on the workspace vector
M_OFF_GET	*memo.f*	returns the start address of a given array on the workspace vector
M_O_GET	*memo.f*	auxiliary routine of M_OFF_GET
M_WHERE	*memo.f*	auxiliary routine of the memory management
NETZDRUCK	*netzdruck.f*	output of the full data structure
NET_0	*net_0.f*	frame for generation of the user mesh
N_KANTE	*AUpfein.f*	generates a new inner edge including the middle node
OLD_KET	old_arr.f	generates the kette array as defined in the former versions of *SPC-PMPo 3D*(KETDIM=7) from the present arrays
OLD_LC	old_arr.f	generates the hierarchical list as defined in the former versions of *SPC-PMPo 3D*(LCDIM=4) from the present list
OUTKETTE	*netzdruck.f*	displays the part of the kette data (for kette see [1]) of one processor
OUTSTANDARD	*standard.f*	displays program control variables
OUT	*netzdruck.f*	displays the part of the solution vector of one processor
OUT_COM_PROB	*outprob.f*	displays problem information from the common block in *com_prob.inc*
PACK_DGRAPH	*dgraph.f*	converts the dual graph from the static structure used in BUILD_DGRAPH to the later used dynamic structure and frees unused memory
PCORECT	*pcorect.f*	determines the middle point of an edge

`PRINT_DGRAPH`	*dgraph.f*	displays the structure `DGRAPH`
`PROJ1FACE`	*pcorect.f*	projection of a node onto one special geometry
`PROJ2ANY`	*pcorect.f*	projection of a node onto two arbitrary geometries; position determined in an iterative process
`PROJ2FACE`	*pcorect.f*	projection of a node onto two special geometries; position must be computable
`PWTABX`	*netzdruck.f*	displays one row of the table of the solution/error
`P_FACE`	*AUpfein.f*	determines the nodes of a face
`RDIRPRINT`	*netzdruck.f*	output of Dirichlet data (RDS, not longer used)
`REBALANCE`	*balance.f*	frame for dynamic load balancing
`REPARTITION`	*balance.f*	determines a repartitioning by calling ParMetis
`RNDPRINT`	*stdwrite.f*	output of boundary condition data
`ROTPROJ`	*pcorect.f*	projection of a node onto a hyperboloid or ellipsoid
`SETFILE`	*setfile.f*	input of the filename
`SETSTANDARD`	*standard.f*	sets the program control variables using file *control.adapt*
`SET_IER`	*ier_set.f*	sets the error parameter `IER` to a given value and displays an error message
`SET_KCHIELD`	*AUpfein.f*	returns the combined refinement type and level of an edge for the data entry in `KANTE`, see 2.2.3
`SET_MINFO`	*memo.f*	auxiliary routine of the memory management
`SET_RBCOM`	*set_rbcom.f*	sets the values of the variables in the common block `RB` in the file *net3ddat.inc*
`SET_VTYP`	*v_typ.f*	sets the refinement type of a volume
`SORTIEREN`	*balance.f*	sorts an array in increasing order
`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
`STORE_NEW_PART`	*balance.f*	corrects the partitioning in the structure `DGRAPH`
`STORE_PARTITION`	*dgraph.f*	writes a full partitioning into the structure `DGRAPH`
`TORPROJ`	*pcorect.f*	projection of a node onto a torus
`T_KUERZ`	*kuerzen.f*	deletes all volumes which are not marked with the own processor number (array `MARK`)
`T_WRITE`	*AUpfein.f*	writes a volume into the array of volumes
`VERSION`	*version.f*	displays the title of the program
`VOR_TRANSFER`	*balance.f*	generates the transfer list for the repartitioning
`VRBPRINT`	*netzdruck.f*	output of Dirichlet data (FDS)
`WTABX`	*netzdruck.f*	prints table of the solution
`ZESCHNITT`	*pcorect.f*	computes the coordinates of a node situated on the cut between a cylinder and a plane
`ZWEIIWERTE`	*standard.f*	reads two integer values from an string variable
`ZYLPROJ`	*pcorect.f*	projection of a node onto a cylinder
`ZZSCHNITT`	*pcorect.f*	computes the coordinates of a node situated on the cut between two cylinders

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

The FORTRAN sources are located in *aNetzT*. The library substitutes *libNT.a* from previous versions of the program.

`A_REFINE`	*a_refine.f*	frame for the adaptive mesh refinement
`C_RENAME`	*t_kante.f*	moves/renames nodes on the kette arrays during the refinement step

`C_UPDATE`	*t_kante.f*	updates the array of edges and the `LC` array after renaming the nodes
`DATREDO`	*n_split.f*	compares two array and erases all non matching data
`DREIMARK`	*sel_mark.f*	red/green marking of faces depending on its edges
`DREI_MROT`	*sel_mark.f*	red marking of a face and its edges
`DREI_VOR`	*drei_vor.f*	frame for counting marked faces
`D_GRUEN`	*t_drei.f*	green refinement of a face
`D_ROT`	*t_drei.f*	red refinement of a face
`D_SPLIT`	*n_split.f*	determines pairs of processors for the data splitting
`D_VOR_1`	*drei_vor.f*	counts marked faces
`EST_MARK`	*est_mark.f*	frame for the marking of tetrahedra by its estimated errors
`FEHLOUT`	*est_mark.f*	displays the estimated error per volume
`GET_G_KANTEN`	*t_drei.f*	determines the green edges of a face which should be red refined
`GET_SPLIT`	*n_split.f*	determines if a processor can split its data
`GROB_RB0`	*set_grobnetz.f*	determines the new number of boundary conditions after fixing the main mesh and definition of the new coarse mesh
`GROB_RB1`	*set_grobnetz.f*	computes the new boundary conditions data
`GTMARK`	*sel_mark.f*	determines the number of red/green marked faces of a tetrahedron
`G_V_G1`	*t_gruen1.f*	reconstruction of the father from two green1 tetrahedra
`G_V_G2`	*t_gruen2.f*	reconstruction of the father from 4 green2 tetrahedra
`G_V_G3`	*t_gruen3.f*	determines for 4 green3 tetrahedra the faces of the father tetrahedron
`HCOM_SIZE`	*a_refine.f*	determines size of auxiliary work space for `SELFEIN`
`K_VOR`	*k_vor.f*	frame for counting marked edges
`K_VOR_1`	*k_vor.f*	counts marked edges
`LC_PLATZ`	*vor_fein.f*	makes space for new nodes in `LC`
`MARKIEREN`	*est_mark.f*	marks volumes by its errors for red refinement
`NO_MARK`	*a_refine.f*	returns `.TRUE.` if nothing is marked for refinement
`N_SPLIT`	*n_split.f*	frame for the data splitting between processors
`POST_CRED`	*n_split.f*	corrects some arrays after shortening of the array of nodes
`POST_FRED`	*n_split.f*	corrects some arrays after shortening of the array of faces
`POST_KRED`	*n_split.f*	corrects some arrays after shortening of the array of edges
`SEL_KUERZ`	*sel_kuerz.f*	erases unused refined faces and edges
`SEL_MARK`	*sel_mark.f*	frame for the extension of the marking to all involved volumes, faces, and edges
`SET_COM_KN`	*a_refine.f*	corrects the common block in *com_prob.inc* after the refinement
`SET_GROBNETZ`	*set_grobnetz.f*	frame for fixing the main mesh
`SET_NETDIM`	*control.F*	sets constants (especially array dimensions) for tetrahedral meshes
`SET_NGR_T`	*n_split.f*	corrects some arrays after the shortening of the `FACE` array
`SET_VMARK`	*a_refine.f*	marks volumes by user input
`SET_VROT`	*sel_mark.f*	marks a tetrahedron and its faces for red refinement
`SHOW_MARK`	*est_mark.f*	displays the volume marking vector
`STWERTE`	*control.F*	presets the program control variables with standard values and opens the file *control.adapt*
`S_MARK2`	*n_split.f*	marks volumes during the splitting processes
`TESTORDN`	*tetordnen.f*	returns `.TRUE.` until the shortest diagonal is in the right place in the data structure of the tetrahedra
`TETMARK`	*sel_mark.f*	checks and marks faces of non red marked volumes
`TETORDNEN`	*tetordnen.f*	sorts the coarse mesh tetrahedra in a way that the shortest diagonal is taken at the first refinement

`TET_VOR`	*tet_vor.f*	counts marked volumes
`TRANSFER`	*transfer.f*	moves (the children of) coarse mesh tetrahedra between processors
`T_DREI`	*t_drei.f*	frame for refining faces
`T_D`	*t_drei.f*	refines a face
`T_GRUEN1`	*t_gruen1.f*	refines a tetrahedron green1
`T_GRUEN2`	*t_gruen2.f*	refines a tetrahedron green2
`T_GRUEN3`	*t_gruen3.f*	refines a tetrahedron green3
`T_KANTEN`	*t_kante.f*	frame for refining edges
`T_K`	*t_kante.f*	refines an edge
`T_ROT`	*t_rot.f*	refines a tetrahedron red
`T_TET`	*t_tet.f*	frame for refining tetrahedra
`T_T`	*t_tet.f*	refines a tetrahedron
`VOR_FEIN`	*vor_fein.f*	determines the size and allocates data arrays prior the refinement
`ZUERST`	*zuerst.f*	prepares the coarse mesh for the refinement (sets initial values of variables etc.)

Chapter 4

Assembly of the equation system

4.1 Changes against version 2.x of *SPC-PMPo 3D*

4.1.1 General remarks

There is mainly one change in the assembly of the stiffness matrix since version 2.x, see [2]. Now, the FDS is used instead of the RDS, which effects only the determination of the nodes of the tetrahedra. The same routines are used for the numerical integration and for the shape functions. The steps in the assembly are also the same.

Other changes are the reorganization of the library and the changed coarse grid matrix.

4.1.2 Library reorganization

The library *libAssem.a* described in [2] had been reorganized for version v3.x of *SPC-PMPo 3D*. This organization is kept by version v4.x.

All element based routines for numerical integration, shape functions, and inhomogeneous Neumann boundary conditions and the element routines itself are now contained in *libElem3D.a*.

The source file *bsp.f* with the user supplied routines for function and its derivatives had been moved out of the library to the main directory of the program. This was done to avoid user specific versions of the library.

The assembly of the equation system and the solver are now decoupled and all solver related routines are now in *libaSolve.a*.

4.1.3 Coarse grid matrix

The handling of the coarse grid matrix and their assembly have been completely changed in version v4.x of *SPC-PMPo 3D*. Until the main mesh is fixed the solver works without any coarse grid solution, and coarse grid matrix respectively.

During the main mesh fixing the present stiffness matrix, or better their Cholesky factorization, is stored as the matrix of the new coarse mesh. Then the solver uses this coarse grid solution.

4.2 Tree structures

```
A_ASSEMBLE
     ↪ KLZ_INIT[1]
     ↪ A_ASSEM
          ↪ MAKEKZU[1]
```

[1]in *libKLZ.a*, see [2], Section 6.1.

```
        ↪ E3LEHF
        ↪ E2INTG
        ↪ E2SHAP
            ↪ PHI2BQ, PHI2L, PHI2Q, P2L, P2Q
        ↪ E3INTG
        ↪ E3SHAP
            ↪ PHI3L, PHI3Q, PHI3TQ, P3L, P3Q, P3TQ, PTL, PTQ
        ↪ ECKPUNKTE
        ↪ A_ELEMENT
            ↪ IHPT
            ↪ A_ELS
                ↪ F[2]
                ↪ JACOBIAN
                ↪ SETPMAT
            ↪ ELAST
                ↪ F[2]
                ↪ SETEMAT
                ↪ JACOBIAN
        ↪ AKKUS, AKKUEL
            ↪ AKKUIJFEST
        ↪ FAKKU, FAKKUEL
        ↪ P_FACE
        ↪ A_NEUMANN
            ↪ IVD
            ↪ A_GET_XL
            ↪ GET_NEUM
            ↪ E3RSOB
                ↪ G[2]
                ↪ USERNEUM
        ↪ PACKKLZ[1]
        ↪ SORTKZU[1]
    ↪ DIRI2A
        ↪ A_K1AKK_VOR[3],A_K2AKK_VOR[3],A_K3AKK_VOR[3]
        ↪ P_FACE
        ↪ AIIKLZ
            ↪ JFROMA
            ↪ AWITHJ
        ↪ DIRINTPO
            ↪ U
            ↪ TRCLOSE
        ↪ A_FEMACC
        ↪ X_UP_IH

A_GROBGIT
    ↪ A_COARSMAT3
        ↪ A_ASSCOARS3
            ↪ MAKEKZU
```

[2]to be supplied in *bsp.f*

[3]in *libaCom.a*, see Chapter 7

↪ `AKKUS`, `AKKUEL`
↪ `PACKKLZ`, `SORTKZU`
↪ `CVBKLZ`

4.3 Short description of the subroutines

4.3.1 Description of the subroutines in *libaAssem.a*

All source files of the library *libaAssem.a* are located in the subdirectory *aAssem*. The library is *no* substitution of the older *libAssem.a*, it is just an extension.

`AIIKLZ`	*aiiklz.f*	writes values on main diagonal of a matrix of KLZ storage type
`A_ASSCOARS3`	*a_coarse.f*	assembles an approximated coarse grid matrix
`A_ASSEM`	*a_assem.f*	assembles the equation system
`A_ASSEMBLE`	*a_assemble.f*	frame for the assembly of the equation system and the handling of the Dirichlet boundary conditions
`A_COARSMAT3`	*a_coarse.f*	frame for `ASSCOARS3`
`A_GET_XL`	*a_neumann.f*	extracts the coordinates of given nodes out of `COOR`
`A_GROBGIT`	*a_grobgit.f*	frame for the assembly coarse grid matrix; not used in the present version
`A_NEUMANN`	*a_neumann.f*	computes the element right hand side for a face
`A_OUTX`	*diri2a.f*	output of the solution vector
`DIRI2A`	*diri2a.f*	marks Dirichlet boundary conditions in the matrix and sets the start vector
`DIRINTPO`	*diri2a.f*	determines values on Dirichlet nodes
`Get_NEUM`	*a_neumann.f*	extracts Neumann data out of `NEUMF`
`X_UP_IH`	*diri2a.f*	auxiliary routine for the handling of Dirichlet boundary conditions in parallel

4.3.2 Description of the subroutines in *libElem3D.a*

The source files of the library *libElem3D.a* are located in the subdirectory *Elem3D*. The library contains all element related routines which are formerly part of *libAssem.a*. It can be used with the adaptive version v4.x and the older version v3.x.

`A_ELEMENT`	*a_element.f*	adaptive version of `ELEMENT`
`A_ELS`	*a_els.f*	adaptive version of `ELS`
`E2INTG`	*e2intg.f*	determines integration points and weights, 2D
`E2SHAP`	*e2shap.f*	determines the shape functions/derivatives in the integration points, 2D
`E3INTG`	*e3intg.f*	determines integration points and weights, 3D
`E3LEHF`	*e3lehf.f*	allocates memory for the arrays `QGST2`, `QGST3`, `SHP2`, `SHP3`, `S`, and `P`
`E3SHAP`	*e3shap.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`

`ELS`	*els.f*	computes element stiffness matrix and the right hand side (Poisson)
`IHPT`	*ihpt.f*	integer function, determines whether an element is a hexahedron (1), a pentahedron (2), or a tetrahedron (3)
`IVD`	*ivd.f*	integer function, determines whether an face is a quadrilateral (1), or a triangle (2)
`JACOBIAN`	*jacobian.f*	determines the Jacobian functional matrix J, its inverse J^{-1}, and its determinant for one integration point in an element
`P2L`	*p2.f*	computes the values of all shape functions/derivatives in a point (linear triangle)
`P2Q`	*p2.f*	computes the values of all shape functions/derivatives in a point (quadratic triangle)
`P3L`	*p3.f*	computes the values of all shape functions/derivatives in a point (linear pentahedron)
`P3Q`	*p3.f*	computes the values of all shape functions/derivatives in a point (quadratic pentahedron)
`P3TQ`	*p3.f*	computes the values of all shape functions/derivatives in a point (quadratic pentahedron; 18 nodes)
`PHI2BQ`	*phi2.f*	computes the values of all shape functions/derivatives in a point (quadratic quadrilateral; 9 nodes)
`PHI2L`	*phi2.f*	computes the values of all shape functions/derivatives in a point (linear quadrilateral)
`PHI2Q`	*phi2.f*	computes the values of all shape functions/derivatives in a point (quadratic quadrilateral; 8 nodes)
`PHI3L`	*phi3.f*	computes the values of all shape functions/derivatives in a point (linear hexahedron)
`PHI3Q`	*phi3.f*	computes the values of all shape functions/derivatives in a point (quadratic hexahedron)
`PHI3TQ`	*phi3.f*	computes the values of all shape functions/derivatives in a point (quadratic hexahedron; 27 nodes)
`PTL`	*pt.f*	computes the values of all shape functions/derivatives in a point (linear tetrahedron)
`PTQ`	*pt.f*	computes the values of all shape functions/derivatives in a point (quadratic tetrahedron)
`SETEMAT`	*setmat.f*	set material dependent values for each material range (Elasticity)
`SETPMAT`	*setmat.f*	set material dependent values for each material range (Poisson)

Chapter 5

Solving the problem with the Parallel Preconditioned Conjugate Gradient Method (PPCG)

5.1 The Solver

In Version v4.x the assembly of the equation system and the solving process are completely decoupled. The routine `ASSLOES`, known from [2] is now replaced by `A_ASSEMBLE` (see Chapter 4) and `A_LOESEN`.

The solver in `A_LOESEN` is the PPCG solver with the concept of non-overlapping domain decomposition and data storage described in [2].

Due to the adaptivity of the program there are some major changes in the preconditioners especially in the BPX. These changes will be described in the Sections 5.3 – 5.4.

The subroutine `A_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.

Some specific initializations for the CG method and the preconditioners are realized in the subroutine `A_PREVOR`. First the subroutine `D_OUT_KLZ` (see [7]) 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. Since we have fixed and factorized the real stiffness matrix on level 0 (main mesh) within `SET_GROBNETZ` the special handling of Dirichlet boundary conditions and its factorization is not longer necessary in `A_PREVOR`.

At the end the subroutine `A_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 the Oxer 1.D+40 to zero (see [2] Section 4.3.1, Step 5). In case of the BPX preconditioner D^{-1} is expanded according to V_q^E, see 5.4.

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

5.2 The Jacobi preconditioner

The Jacobi preconditioner is the simplest preconditioner. 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 within the subroutine `A_PRLOES`.

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. Note that a change of `Delta` is only possible until the main mesh is fixed		
`z`	control of the amount of screen output, see `ion` in [3, Table 2.1]		
`p`	switches the plot of the CG-Iterations on/off; for information see [5]		

Table 5.1: Control parameters for the solver.

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 `A_FEMACC`, see Section 7 and [1]. 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 [9] 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

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

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

$$C^{-1} = SD^{-1}S^T \tag{5.2}$$

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.

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

$$C^{-1} = SA_0^{-1}S^T, \quad \text{with} \tag{5.3}$$

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

LL^T is the Cholesky decomposition of the matrix C_0, and C_0 is the finite element assembly of the stiffness matrix on level 0 (main mesh), which is stored by the routine `SET_GROBNETZ`. The coarse grid matrix can be scaled by a factor `Delta` before it is factorized (until now there are no experiences what a good `Delta` could be). The matrix $\tilde{D}$ is the part of the diagonal D of the stiffness matrix not belonging to the coarse grid.

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 also realized within the subroutine `A_PRLOES`. The transformation with the matrices S and S^T is carried out in the subroutines `A_HiSmulYser` and `A_HSTmulYser`, respectively:

Routine	Description
`A_HiSmulYser(Nfg,Nk,X,Liste)`	$X = SX$
`A_HSTmulYser(Nfg,Nk,X,Liste)`	$X = S^T X$

Here, `Nfg` denotes the number of degrees of freedom, `Nk` is the number of nodes on the subdomain (node) `k`, `X` is the vector of the length `N` = `Nk` ∗ `Nfg`, and `Liste` is the hierarchical list on the subdomain, which is generated by the mesh refinement procedures, see Chapter 3. `Liste` is the two-dimensional array `LC` described in 2.2.10, which has in the case of the Yserentant the following form:

array	Description
`Liste[LC_LEN,Nk]`	`Liste[1,*]` - node number
	`Liste[2,*]` - left father
	`Liste[3,*]` - right father
	`Liste[4,*]` - coefficient
	`Liste[5,*]-Liste[9,*]` - not used

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 `A_PRLOES` 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^{-1} (therefore in the subroutine `A_PREVOR` the inverse of D is computed). In case we use a coarse grid solver only the part of w not belonging to coarse grid nodes is multiplied with the corresponding part of the D.

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 only stored on processor 0 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 the equations (5.2) and (5.3) we compute 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 [4] 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

$$\left(\mathcal{I}_{k-1}^k\right)_{ij} = \begin{cases} 1 & \text{if } i = j, \quad i,j = 1,2,\ldots \mathtt{N}_{k-1} \\ \frac{1}{2} & \text{if } j = i_1 \text{ and } j = i_2, \text{ where } P^{(i)} \text{ is the middle point between } P^{(i_1)} \text{ and } P^{(i_2)} \text{ which are the end points of an edge of a tetrahedron from the mesh } \mathcal{T}_{k-1} \\ 0 & \text{else} \end{cases} \tag{5.4}$$

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

$$C^{-1} = \hat{S}\hat{D}^{-1}\hat{S}^T \tag{5.5}$$

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.

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

$$C^{-1} = \hat{S}\hat{A}_0^{-1}\hat{S}^T, \quad \text{with} \tag{5.6}$$

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

For δLL^T see Section 5.3. The matrix $\check{D}$ is the part of the expanded diagonal $\hat{D}$ of the stiffness matrix not belonging to the coarse grid.

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 on the other hand the condition number of $C^{-1}K$ is $\mathcal{O}(1)$ for the BPX preconditioner.

Before we can use the subroutine `A_PRLOES` for the BPX preconditioning some additional initialization steps are necessary. At first we have to predict the dimension of V_q^E which determines the length of the expanded residuum w and the expanded diagonal $\hat{D}$. This is done by the function `GET_WLEN_BPX`. Moreover, this function generates the entries `K_O_BPX` and `K_L_BPX` in the kette arrays and initializes the vector `START` needed by `A_HB2BPX`. Now we extend the hierarchical list with additional entries in the `LC_DAT` part in the following way:

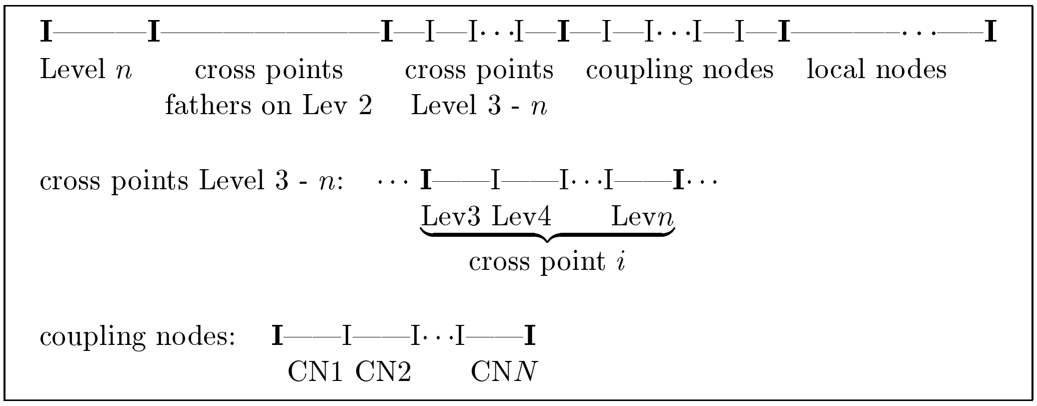
array	Description
`Liste[LC_LEN,Nk]`	`Liste[1,*]` - node number `Liste[2,*]` - left father `Liste[3,*]` - right father `Liste[4,*]` - Son `Liste[5,*]` - right from in zone $i+1$ `Liste[6,*]` - left from in zone $i+1$ `Liste[7,*]` - right to in zone $i-1$ `Liste[8,*]` - left to in zone $i-1$ `Liste[9,*]` - not used

Note that the values in `Liste[5,*] - Liste[8,*]` are positions on the extended w or $\hat{D}$ respectively.

The list extention is done by the subroutine `A_HB2BPX`:

`A_HB2BPX(Liste,KETTE,V_BIT,START,Mfr,Mto)`		
	input	output
`Liste`	hierarchical list	hierarchical list with BPX data in the `LC_DAT` part
`KETTE`	`KETTE` list	—
`V_BIT`	masked fatherhood of the nodes taken from last column of `COOR`	—
`START`	start of coupling node data on the extended vectors	corrected start points
`Mfr`	auxiliary array	—
`Mto`	auxiliary array	—

For the interprocessor communication we need a special order of crosspoints and nodes on coupling edges and faces to preserve the same position on the extended vector w on each processor. The extended vectors are organized as follows:



This structure exists separately for each degree of freedom. The inverse diagonal of the stiffness matrix D^{-1} is extended to $\hat{D}^{-1}$ in the same way by the routine `DIA2BPX`.

According to the new handling of the additional BPX data there is no need to provide additional memory for the hierarchical list and the Kettes as in previous versions of *SPC-PMPo 3D*. But you have to provide two additional vectors `V_BIT` and `START` and you still have to provide enough memory for the auxiliary vector w in the BPX preconditioner and the vector of the main diagonal of the stiffness matrix (better: its inverse D^{-1}) 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 `A_HiSmulBPX`, `PRE_HSTmulBPX`, and `A_HSTmulBPX`:

Routine	Description
`A_HiSmulBPX(W,Liste,KETTE,M,V_BIT,START)`	$X = \hat{S}X$
`PRE_HSTmulBPX(W,Liste,KETTE,V_BIT)`	extends X according to V_q^E
`A_HSTmulBPX(Nfg,W,Liste,M)`	$X = \hat{S}^T X$

Like in the Yserentant case the subroutine `A_PRLOES` 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 $\hat{D}^{-1}$ ($\hat{D}$ is the extended main diagonal of the stiffness matrix). If a coarse grid solver is used, only the part of w not belonging to the coarse grid is multiplied with $\check{D}^{-1}$

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 the 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 $w = \hat{S}w$, which 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 routine

In the case of a BPX preconditioning the initialization subroutine `A_HB2BPX` is called in the subroutine `A_LOESEN`. The PCG method is realized by the subroutine `A_PPCGM`:

```
A_LOESEN
  ↪ OUT_COM_PROB
  ↪ A_STARTWR3D
      ↪ A_FEMACC[2]
      ↪ CUBE_DOD[5]
      ↪ PLOT_INIT
      ↪ PLOT_NAME
      ↪ PLOT_CMD
      ↪ TREE_DOWN_0[5]
  ↪ A_FEMACC[2]
  ↪ GET_WLEN_BPX
      ↪ GET_NV
  ↪ A_HB2BPX
  ↪ A_YSFAKTOR
  ↪ A_PPCGM
      ↪ DSCAPR
      ↪ A_PREVOR
          ↪ D_OUT_KLZ[1]
          ↪ A_TREEUP_DOD[2]
          ↪ OXCOPYVBZ
          ↪ VDMULT[3]
          ↪ CHOVBZ[4]
          ↪ TREE_DOWN_0[5]
          ↪ A_FEMACC[2]
          ↪ A_HISCALE3D
          ↪ VDDIVO[3]
          ↪ PRE_HSTMULBPX
              ↪ GET_NV
          ↪ DIA2BPX
      ↪ AXMKLZ[1]
      ↪ VDMINUS[3]
      ↪ A_PRLOES
          ↪ VDOMUL[3]
          ↪ A_HSTMULYSER
```

[1]in *libKLZ.a*, see [2], Section 6.1

[2]in *libaCom.a*, see Section 7

[3]in *libvbasmod.a*, see [2], Section 6.3

[4]in *libMbasmod.a*, see [2], Section 6.4

[5]in *libCubecom.a*, see [2], Section 6.2

```
    ↪ PRE_HSTMULBPX                        ↪ A_K3AKKP²
    ↪ A_HSTMULBPX                          ↪ A_TREE_DOWN²
    ↪ VDMULT³                              ↪ A_HISMULYSER
    ↪ A_FEMACC²                            ↪ A_HISMULBPX
    ↪ A_TREEUP_DOD²                        ↪ VDOMUL³
    ↪ RUEVBZ⁴                            ↪ A_TREE_DOD²
    ↪ VORVBZ⁴                            ↪ VDAXPY³
    ↪ A_K3AKK²                           ↪ ZWISCH
```

5.6 Description of the routines

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

`A_HB2BPX`	*bpx_hiemul.f*	fills the `LC_DAT` part of the hierarchical list with data for the BPX
`A_HISCALE3D`	*hiemul.f*	scaling of the main diagonal elements with the mesh size of the corresponding zone
`A_HISMULBPX`	*bpx_hiemul.f*	multiplication with the transformation matrix $\hat{S}$
`A_HISMULYSER`	*hiemul.f*	multiplication with the transformation matrix S
`A_HSTMULBPX`	*bpx_hiemul.f*	multiplication with the transformation matrix $\hat{S}^T$
`A_HSTMULYSER`	*hiemul.f*	multiplication with the transformation matrix S^T
`A_LOESEN`	*a_loesen.f*	frame for solving the equation system
`A_PPCGM`	*a_ppcgm.f*	parallel preconditioned conjugate gradient method
`A_PREVOR`	*a_prevor.f*	initializations depending on the kind of the chosen preconditioner
`A_PRLOES`	*a_prloes.f*	preconditioning depending on the kind of the chosen preconditioner
`A_STARTWR3D`	*a_startwr3d.f*	provides the possibility to change solver parameters
`DIA2BPX`	*bpx_hiemul.f*	expands the extracted diagonal D to $\hat{D}$ needed by the BPX
`GET_NV`	*bpx_hiemul.f*	determines the number of fatherhoods
`GET_WLEN_BPX`	*bpx_hiemul.f*	determines the temporary length of the preconditioned residual vector during the BPX; corresponds to the former value `NBPX`
`OUT_LISTE`	*bpx_hiemul.f*	auxiliary display routine
`PRE_HSTMULBPX`	*bpx_hiemul.f*	initializes the multiplication with the transformation matrix $\hat{S}^T$
`VIOR`	*a_loesen.f*	combines to vectors by a logical OR
`ZWISCH`	*zwisch.f*	displays the values of the CG parameters

Chapter 6

Memory management

6.1 Introduction

Within FORTRAN77 programming the memory management concept of the workspace vector is widely used. At the start of the program a very large vector is allocated and the storage on this vector is managed by the user via offsets. This is a very efficient way of memory management but it is also often the reason of hardly to find bugs.

For the new version 4.x of *SPC-PMPo 3D* a set of functions and routines was written by F. Milde to make the handling of the workspace vector simpler and more reliable. Simple operations like allocating and de-allocating arrays are provided but also more complex operations like increasing or decreasing the size of arrays.

All the functions and routines are contained in *aNetzA/memo.f*, which also needs the include file *include/memo.inc*. The arrays on the workspace vector are managed using an info block at the beginning of the vector. This info block must be initialized at the start of the program. At this point the maximal number of arrays to be managed is fixed.

The functions in *memo.f* are described in the following sections. The description of each function/routine consists of the calling sequence, the explanation of the parameters, and a short description of the function.

6.2 Basic functions

`SUBROUTINE MEMO_INIT(A,TYPE,LENGTH,NA_MAX,IER)`

`A`	I/O	Workspace vector
`TYPE`	I	Type of `A`; Bytes per element of `A`
`LENGTH`	I	Length of `A` in `TYPE` units
`NA_MAX`	I	Maximum number of arrays to be managed on `A`
`IER`	I/O	Error parameter; zero if no error appears

Initializes the workspace vector for the memory management.

`INTEGER*4 FUNCTION M_FREE_GET(A)`

`A`	I	workspace vector

Returns the offset for the free part of the workspace vector.

INTEGER*4 FUNCTION M_NEW(A,TYPE,DIM,NUM,NAME,A_NUM,O_IND,IER)

A	I/O	Workspace vector
TYPE	I	Type of A; Bytes per element of A
DIM	I	Dimension of the array; number of TYPE block per entry
NUM	I/O	Length of the array (in DIM*TYPE blocks)
NAME	I	String with the name of the array
A_NUM	O	Number of the array in the info block
O_IND	I	Original address of the array or -1
IER	I/O	Error parameter; zero if no error appears

Allocating a new array on the workspace vector. The function returns the offset for the array on A. If NUM is -1 on input all the remaining space is allocated and NUM returns the length of the array. The parameter O_IND should normally set to -1. It can be used to integrate an already existing array in the workspace management. In this case O_IND must be the original address of the array.

SUBROUTINE M_DEL(A,NAME,A_NUM,IER)

A	I/O	Workspace vector
NAME	I	String with the name of the array
A_NUM	I	Number of the array in the info block
IER	I/O	Error parameter; zero if no error appears

Deletes an array specified by A_NUM and/or NAME on the workspace vector. The data is not removed physically, only the entry in the management info block is freed.

INTEGER*4 FUNCTION M_OFF_GET(A,A_NUM,NAME,IER)

A	I	Workspace vector
A_NUM	I	Number of the array in the info block
NAME	I	String with the name of the array
IER	I/O	Error parameter; zero if no error appears

Returns the offset of the array given by A_NUM and/or NAME on the workspace vector.

6.3 Getting information on the workspace vector

REAL FUNCTION G_MEM_USE(A)

A	I	workspace vector

Returns the usage of the workspace vector in percent.

SUBROUTINE MEMO_USE(A)

A	I	workspace vector

Output of the the usage of the workspace vector (percentage).

`SUBROUTINE MEMO_OUT(A)`

`A`	I	workspace vector

Output of the usage of the workspace vector (percentage + information about all arrays).

6.4 Changing array sizes

Because of the linear storage of the data on the workspace vector every change in the size of an array (except the last) cause data movement. To keep this movement as small as possible the change of array sizes is done in several steps:

1. Initialization of an auxiliary array
2. Registration of all concerned arrays
3. Data movement
4. Removing the auxiliary array
5. Getting the new offsets

The associated routines are the following:

`SUBROUTINE M_CH_PRE(A,IH,HL,IER)`

`A`	I/O	workspace vector
`IH`	O	offset of the auxiliary array on `A`
`HL`	O	length of the auxiliary array
`IER`	I/O	error parameter; zero if no error appears

Initializes an auxiliary array at the end of the workspace vector for changing array sizes.

`SUBROUTINE M_CH_VAL(A,H,A_NUM,NAME,NUM,IER)`

`A`	I/O	Workspace vector
`H`	I/O	Auxiliary array created by `M_CH_PRE`
`A_NUM`	I	Number of the array in the info block
`NAME`	I	String with the name of the array
`NUM`	I	New length of the array
`IER`	I/O	Error parameter; zero if no error appears

Registers the new length of the array specified by `A_NUM` and `NAME` for data movement.

`SUBROUTINE M_CH_MAIN(A,H,WHAT,IER)`

`A`	I/O	Workspace vector
`H`	I	Auxiliary array created by `M_CH_PRE`
`WHAT`	I	String; either `'MORE'` or `'LESS'`
`IER`	I/O	Error parameter; zero if no error appears

Performs the size changes by moving the data. To keep things simple there are only two possibilities for changing array sizes:

`WHAT='LESS'`: All array sizes decrease or remain unchanged.

`WHAT='MORE'`: All array sizes increase or remain unchanged.

This allows an unidirectional data movement to preserve efficiency.

`SUBROUTINE M_CH_POST(A,HL)`

`A`	I/O	Workspace vector
`HL`	I	Length of the auxiliary array created by `M_CH_PRE`

Removes the auxiliary array from the workspace vector.

To obtain the new offsets of the arrays the function `M_OFF_GET` should be used, see 6.2.

6.5 Management in the program

There are the following 15 fixed and managed arrays in the present version of *SPC-PMPo 3D*:

`G_LC`	Row pointer vector for the coarse grid matrix (VBZ)
`G_CC`	Row data of the coarse grid matrix (VBZ)
`GEOM`	geometry data, see 2.2.11
`DIR`	Dirichlet boundary conditions, see 2.2.6
`NEUM`	Neumann boundary conditions, see 2.2.6
`IGLOB`	Global crosspoint names, see 2.2.5
`KETTE1D`	1D kettes, see 2.2.7
`KETTE2D`	2D kettes, see 2.2.7
`COOR`	Array of nodes, see 2.2.4
`VOL`	Array of volumes, see 2.2.1
`KANTE`	Array of edges, see 2.2.3
`FACE`	Array of faces, see 2.2.2
`X`	Solution vector, see 2.2.12
`LC`	Hierarchical list, see 2.2.10
`DGraph`	Partitioning information, see 2.2.14

Usually, most of these arrays grow in size during an adaptive refinement step. The other ones might change their location (offset). All these changes should be managed using the routine `mem_change`. It is defined as follows:

```
SUBROUTINE MEM_CHANGE(A,IER,WHAT,
NCP,NUMNP,NKANTE,NFACE,NVOL,
NK1,NK2,NDIR,NNEUM,JDIR,JNEUM,N_GLC,
N_GCC,NGEOM,J_GLC,J_GCC,J_GEOM,JIGLOB,
JKETTE1D,JKETTE2D,JCOOR,JLC,JVOL,JKANTE, JDREI,JX,JDGRAPH)
```

`A`	I/O	Workspace vector
`IER`	I/O	Error parameter; zero if no error appears
`WHAT`	I	String; either `'MORE'` or `'LESS'`
`NCP`	I	Number of crosspoints
`NUMNP`	I	Number of nodes
`NKANTE`	I	Number of edges
`NFACE`	I	Number of faces
`NVOL`	I	Number of volumes
`NK1`	I	Number of 1D kettes
`NK2`	I	Number of 2D kettes
`NDIR`	I	Number of Dirichlet boundary conditions
`NNEUM`	I	Number of Neumann boundary conditions
`N_GLC`	I	Dimension of the coarse grid matrix
`N_GCC`	I	Number of entries in the coarse grid matrix
`NGEOM`	I	Number of geometry data sets
`J_GLC`	I	Offset for the row offset vector of the coarse grid matrix
`J_GCC`	I/O	Offset for the data of the coarse grid matrix
`J_GEOM`	I/O	Offset for the geometry data
`JIGLOB`	I/O	Offset for the global crosspoint names
`JKETTE1D`	I/O	Offset for the 1D kettes
`JKETTE2D`	I/O	Offset for the 2D kettes
`JCOOR`	I/O	Offset for the nodes
`JLC`	I/O	Offset for the hierarchical list
`JVOL`	I/O	Offset for the volumes
`JKANTE`	I/O	Offset for the edges
`JDREI`	I/O	Offset for the faces
`JX`	I/O	Offset for the solution vector
`JDGRAPH`	I/O	Offset for the partitioning data

The routine takes the parent array offsets and the new array lengths as input, performs the changes according to `WHAT` and gives the new array offsets back. The error parameter `IER` is set, if there is not enough space for the changes on the workspace vector. It is highly recommended to use this routine for changing the size of any of this arrays to keep the data structures consistent.

6.6 Usage example

In the following we give a little example to demonstrate the usage of the memory management routines:

```
      INTEGER*4 LENGTH
      PARAMETER (LENGTH=50000)
      INTEGER*4 A(LENGTH)
      INTEGER*4 M_FREE_GET, M_OFF_GET, M_NEW
      EXTERNAL M_FREE_GET, M_OFF_GET, M_NEW

C INITIALISATION OF THE MEMORY MANAGEMENT; 35 ARRAYS MAXIMUM
      CALL MEMO_INIT(A,4,LENGTH,35,IER)

C GET A POINTER TO A NEW INTEGER ARRAY
      JMARK = M_NEW(A,4,FIELDDIM,LEN  ,'Mark   ',NR,-1,IER)
C GET A POINTER TO A NEW REAL*8 ARRAY
      J_X   = M_NEW(A,8,NDF     ,NUMNP,'XValues',K ,-1,IER)

C USE THE ARRAYS IN A SUBROUTINE
      CALL MARKING(A(JMARK),A(J_X), ...)

C OUTPUT OF THE MEMORY USAGE
      CALL MEMO_OUT(A)

C CHANGE ARRAY SIZES (ALL INCREASE OR ALL DECREASE, NEVER MIXED)
C X INCREASES
      N_X_NEW = NUMNP + 400
      CALL M_CH_PRE(A,IH,HL,IER)
C K=2 IS THE ARRAY NUMBER OF X, NEITHER JMARK NOR J_X CHANGES IN THIS CASE
      CALL M_CH_VAL(A,A(IH),K,'XValues',N_X_NEW,IER)
      CALL M_CH_MAIN(A,A(IH),'MORE',IER)
      CALL M_CH_POST(A,HL)

C GET THE POINTER TO THE FREE SPACE ON A
      JFREE = M_FREE_GET(A)
C USE IT
      CALL SET_X(A(J_X), ... , A(JFREE))

C MARK DECREASES
      LEN_NEW = LEN - 10
      CALL M_CH_PRE(A,IH,HL,IER)
C NR=1 IS THE ARRAY NUMBER OF MARK
      CALL M_CH_VAL(A,A(IH),NR,'Mark',LEN_NEW,IER)
      CALL M_CH_MAIN(A,A(IH),'LESS',IER)
      CALL M_CH_POST(A,HL)
C GET NEW POINTERS TO THE INFLUENCED ARRAYS; ARRAY NUMBER K=2
      J_X = M_OFF_GET(A,K,'XValues',IER)
      JFREE = M_FREE_GET(A)

C OUTPUT OF THE PERCENTAGE OF MEMORY USED
      CALL MEMO_USE(A)

C DELETE THE ARRAYS, START WITH LAST!
      CALL M_DEL(A,K ,'XValues',IER)
      CALL M_DEL(A,NR,'Mark'   ,IER)
```

Chapter 7

Enhanced communication routines

7.1 The concept

The new communication routines contained in in the library *libaCom.a* are generalized versions of well known routines from *libDDCMcom.a*. According to the increased requirements the routines include not only the communication over nodes, but also over edge and faces in single or double precision. They support also every vector operation from *libvbasmod.a* and not only addition.

To obtain this functionality a new calling scheme was introduced. Every routine takes an input string called `WAS`. It consists of two capital letters denoting the action to take and the data type. The possibilities for the first letter are given in table 7.1. The second letter is either `S` (real*4), `I` (integer*4) for single precision or `D` (real*8) for double precision.

Letter	Description
`F`	Communication over faces
`E`	Communication over edges
`N`	Communication over nodes
`W`	Communication according to the BPX vector w

Table 7.1: Possibilities for the first letter of the communication descriptor `WAS`.

The desired arithmetical operation is also given as input. The routines take a pointer to a function from *libvbasmod.a*.

The new routines are also capable to communicate just over the current hypercube dimension `LOC_CUBE` (in cases where only part of the nodes have already data). `LOC_CUBE` can vary between 0 and `NCUBE` which is the maximal cube dimension.

7.2 Communication over kettes

The routine names are derived from the original corresponding routines, just an *A_* was added to denote *adaptivity* which stands for the new program version.

The routines are defined as follows:

SUBROUTINE A_FEMACC(WAS,OPER,CDIM,VAR,Nfg,RC,K1D,K2D,IGLOB, H,PROT)

`WAS`	I	Action descriptor as explained in sec. 7.1
`OPER`	I	Operation to execute; for example `VDplus`
`CDIM`	I	Dimension of crosspoint matrix
`VAR`	I	`Kett_Akk` variant
`Nfg`	I	Degrees of freedom
`RC`	I/O	Vector to accumulate
`K1D`	I	1D kette
`K2D`	I	2D kette
`IGLOB`	I	list of global crosspoint names
`H`	H	work array as large as possible
`PROT`	I/O	Protocol array to speed up communication

The routine corresponds to `femakk` with an improved functionality. The operation `OPER` is applied to the vector `RC` on processor borders. The input `WAS` and `OPER` is explained in the previous section. The value `CDIM` could be '+' the dimension of the crosspoint matrix, zero, or '-' the dimension of the crosspoint matrix. In the first case crosspoint communication and kette communication takes place, in the second case just kette communication and in the third just crosspoint communication. The value `VAR` corresponds to `FEMAKKVAR` known from previous program versions. `FEMAKKVAR`=1 is not yet supported.

The size of the auxiliary vector `H` should be as large as possible. If it is large enough all will be OK, if not, a segmentation fault might occur. A rough upper limit for the length is $2 * $ `Nfg` times the sum of the local nodes over the processors.

SUBROUTINE A_K1AKK_VOR(KETTE,H)

`KETTE`	I/O	1D kette
`H`	H	Large auxiliary vector

Generates the communication information `PWEGID` for 1D kettes. The routine handles the new structure of the kettes. It corresponds to `KettAkk_Vor`.

SUBROUTINE A_K2AKK_VOR

The corresponding routine to `Kett2Akk_Vor` is not yet provided.

SUBROUTINE A_K3AKK_VOR(KETTE,H)

`KETTE`	I/O	2D kette
`H`	H	Large auxiliary vector

Generates the communication information `PWEGID` for 2D kettes. The routine handles the new structure of the kettes. It corresponds to `Kett3Akk_Vor`.

SUBROUTINE A_K1AKK

The corresponding routine to `Kett1Akk` is not yet provided.

SUBROUTINE A_K3AKK(WAS,Nfg,RC,Kette,H,OPER)

`WAS`	I	Action descriptor as explained in 7.1
`Nfg`	I	Degrees of freedom
`RC`	I/O	Vector to accumulate
`Kette`	I	1D and 2D kettes; they must be stored continuously
`H`	H	work array as large as possible
`OPER`	I	Operation to execute; for example `VDplus`

The routine applies the operation `OPER` to the vector `RC` on processor borders. Communication takes place over 1D and 2D kettes. The routine corresponds to `Kett3Akk`.

SUBROUTINE A_K3AKKP(WAS,Nfg,RC,Kette,H,OPER,PROT)

`WAS`	I	Action descriptor as explained in 7.1
`Nfg`	I	Degrees of freedom
`RC`	I/O	Vector to accumulate
`Kette`	I	1D and 2D kettes; they must be stored continuously
`H`	H	work array as large as possible
`OPER`	I	Operation to execute; for example `VDplus`
`PROT`	I/O	Protocol array to speed up communication

This routine provides the same functionality as `A_K3AKK` but at the first call of the routine the actual communication routes are logged to the array `PROT` to speed up all further runs of the routine. It corresponds to `Kett3AkkP`.

7.3 Cube communication

As already mentioned in section 7.1 a specialty of *SPC-PM Po 3D*version 4.x is the distinction between `NCUBE` and `LOC_CUBE`. Until the maximal hypercube dimension `NCUBE` is reached the standard communication routines from *libCubecom.a* would be slower. However, the functionality and the parameters stay the same. For a more detailed description the reader might refer to [5]. Only a small subset of adjusted routines is provided:

SUBROUTINE A_TREE_DOWN(N,WORDS)

`N`	I	Number of words
`WORDS`	I/O	vector of words with length `N`

Distributes the vector `WORDS` tree downwards to all processors. The routine corresponds to `TREE_DOWN`.

SUBROUTINE A_Tree_DoD(N,X,Y,H,VDop)

`N`	I	Vector length
`X`	O	result vector
`Y`	I	Input vector
`H`	H	Auxiliary vector
`VDop`	I	Operation to execute; for example `VDplus`

The vector operation `VDop` is carried out over all processors. The input and output vectors are double precision. The routine corresponds to `Tree_DoD`.

`SUBROUTINE A_TreeUp_DoD(N,X,Y,H,VDop)`

`N`	I	Vector length
`X`	O	result vector
`Y`	I	Input vector
`H`	H	Auxiliary vector
`VDop`	I	Operation to execute; for example `VDplus`

The routine provides the same functionality as `A_Tree_DoD` but the result arises only on processor 0! The routine corresponds to `TreeUp_DoD`.

Chapter 8

Auxiliary and tool routines

8.1 Preface

A large set of auxiliary and tool routines is provided for the unification of heavily used functionalities. The most important set of such routines is the memory management described in chapter 6. Additionally, there exist routines for an unified error handling and a lot of tool routines for the manipulation of various data sets. To keep readability and compatibility the user is requested to use these routines when ever possible.

8.2 Error handling

Most of the routines in *SPC-PMPo 3D*V4.x take and give back an error indicator named `IER`. This parameter is normally zero and carries a certain nonzero value if an error occurs. So for an efficient error handling it is necessary to set `IER` on an error and later to check if an error has occurred. For this we provide two functions:

`LOGICAL*4 FUNCTION SET_IER(IER,PROG,VAL)`

`IER`	O	Error indicator
`PROG`	I	String (usually) containing the name of the calling routine
`VAL`	I	Value the error indicator should be set to

The function sets the error indicator to the given value and displays a error message like:

`Proz.` *X*: `ERROR IN` *PROG* `:` *VAL*

This functionality is provided as a logical function to enable calling sequences like:

```
      IF(SET_IER(IER,'my_buggy_routine',1)) RETURN
```

The function always returns `.TRUE.`

`LOGICAL*4 FUNCTION IER_TEST(IER,PROG,VAL)`

`IER`	I/O	Error indicator
`PROG`	I	String (usually) containing the name of the calling routine
`VAL`	I	Value the error indicator should be set to

In difference to `SET_IER` this function takes `IER` as input and checks if the error indicator is already set by an previous routine. If the error indicator is zero the function return `.FALSE.`. If not, the error indicator is set to `VAL` and an error message is displayed:

`Proz.` *X*: `ERROR IN` *PROG* `:` *VAL*

In this case it returns `.TRUE.`. The function is intended to be used like:

```
      ...
      CALL BUGGY_ROUTINE( ... ,IER)
      IF (IER_TEST(IER,'THIS_ROUTINE',1)) THEN
C Error handling
         ...
      ENDIF
```

or in the easiest case:

```
      ...
      CALL BUGGY_ROUTINE( ... ,IER)
      IF (IER_TEST(IER,'THIS_ROUTINE',1)) RETURN
      ...
```

8.3 Auxiliary routines

8.3.1 Set special data fields

`SUBROUTINE K_LC(COOR,X,LC,TIEFE,P,V1,V2)`

COOR	I/O	Array of nodes
X	I/O	Solution vector
LC	I/O	Hierarchical list
TIEFE	I	Level depth of the node
P	I	Number of node
V1	I	Number of father 1
V2	I	Number of father 2

The routine registers a node in the hierarchical list, sets the fatherhood bits at father 1 and 2 and interpolates a solution for the son from those of the two father nodes.

`SUBROUTINE K_WRITE(KANTE,NR,A,B,M,ZEIG,TYP,DEP,PM)`

KANTE	I/O	Array of edges
NR	I/O	Number of the edge
A	I	First node of the edge
B	I	Last Node of the edge
M	I	Middle node of the edge
ZEIG	I	Geometry types of the edge; Output of `KA_CODE`
TYP	I	Refinement type of the edge
DEP	I	Refinement depth of the edge
PM	I	Value which is added to `NR`

The routine writes a complete edge data set into the edge array. For a more detailed explanation of the input values the reader might refer to section 2.2.3. The input value `PM` could be useful in an loop over `NR`. The routine returns `NR` = `NR` + `PM`.

INTEGER*4 FUNCTION SET_KCHIELD(TYP,TIEFE)

TYP	I	Refinement type of the edge
TIEFE	I	Refinement depth of the edge

The function return an 32 Bit integer value containing the information TYP and TIEFE. The exact encoding is described in 2.2.3.

SUBROUTINE KA_CODE(KCODE,FCODE,IER)

KCODE	I/O	Geometry types of the edge
FCODE	I	Geometry types of a face belonging to the edge
IER	O	Error indicator

The routine writes the geometry information from the face to KCODE. This can be done at most for two faces belonging to an edge. If a third face geometry should be added IER will be set.

SUBROUTINE KAC_OPT(KCODE,GEOM,NGEOM,IER)

KCODE	I/O	Geometry types of the edge
GEOM	I	Geometry data set
NGEOM	I	Number of possible geometries
IER	O	Error indicator

The routine tries to optimize the edge geometry in a certain sense.

SUBROUTINE D_WRITE(FACE,NR,A,B,C,ZEIG,CHIELD,PM)

FACE	I/O	Array of faces
NR	I/O	Number of face to write
A	I	Number of first edge
B	I	Number of second edge
C	I	Number of third edge
ZEIG	I	Geometry type of the face
CHIELD	I	Refinement type of the face
PM	I	Value which is added to NR

The routine writes a complete face data set to the face array. For a more detailed explanation of the input values the reader might refer to section 2.2.2. The input value PM could be useful in an loop over NR. The routine returns NR = NR + PM.

SUBROUTINE T_WRITE(VOL,NR,A,B,C,D,REG,TYP,G_NR,PM)

VOL	I/O	Array of volumes
NR	I/O	Number of volume to write
A	I	Number of the first face
B	I	Number of the second face
C	I	Number of the third face
D	I	Number of the fourth face
REG	I	Number of material
TYP	I	Refinement type
G_NR	I	Name of the coarse volume the volume belongs to
PM	I	Value which is added to NR

The routine writes a complete volume data set to the volume array. For a more detailed explanation of the input values the reader might refer to section 2.2.1. The input value PM could be useful in an loop over NR. The routine returns NR = NR + PM.

```
SUBROUTINE SET_VTYP(VOL,NR,TYP,GROB_NR)
```

VOL	I/O	Array of volumes
NR	I	Number of the volume
TYP	I	Refinement type
GROB_NR	I	Name of the coarse volume the volume belongs to

The routine stores the information TYP and GROB_NR in the data section of the volume NR. The exact encoding is described in 2.2.1.

8.3.2 Read special data fields

```
INTEGER*4 FUNCTION GET_KTYP(KANTE,NR)
```

KANTE	I	Array of edges
NR	I	Number of an edge

The function returns the refinement type of the edge NR.

```
INTEGER*4 FUNCTION GET_KDEPTH(KANTE,NR)
```

KANTE	I	Array of edges
NR	I	Number of an edge

The function returns the refinement depth of the edge NR.

```
INTEGER*4 FUNCTION GET_FDEP(FACE,NR,KANTE)
```

FACE	I	Array of faces
NR	I	Number of an edge
KANTE	I	Array of edges

The function returns the refinement depth of the face NR.

```
INTEGER*4 FUNCTION GET_VDEP(TET,NR,FACE,KANTE)
```

TET	I	Array of volumes
NR	I	Number of an edge
FACE	I	Array of faces
KANTE	I	Array of edges

The function returns the refinement depth of the volume NR.

```
INTEGER*4 FUNCTION GET_VTYP(VOL,NR)
```

VOL	I	Array of volumes
NR	I	Number of the volume

The function returns the refinement type of the volume NR.

INTEGER*4 FUNCTION GET_GROB_NR(VOL,NR)

`VOL`	I	Array of volumes
`NR`	I	Number of the volume

The function returns the name of the coarse volume the volume `NR` belongs to.

8.3.3 Tools

SUBROUTINE ECKPUNKTE(TET,FACE,KANTE,P,MP,IER)

`TET`	I	volume data set
`FACE`	I	Array of faces
`KANTE`	I	Array of edges
`P`	O	corner/all nodes of `TET`
`MP`	I	Get middle nodes (1) or not (0)
`IER`	O	Error indicator

The routine determines the 4 corner nodes of the tetrahedron `TET`. If the `MP` flag is set and `TET` is a quadratic element, additionally, the six middle nodes of the edges are written to `P(5)` to `P(10)`. Thus, the size of the array `P` must be 4 or 10.

SUBROUTINE P_FACE(FACE,KANTE,ENR,KZAHL,MP,IER)

`FACE`	I	Face data set
`KANTE`	I	Array of edges
`ENR`	O	corner/all nodes of `FACE`
`KZAHL`	I	Number of edges per face
`MP`	I	Get middle nodes (1) or not (0)
`IER`	O	Error indicator

The routine determines the corner nodes of `FACE` (its edges build a closed polygonal track) to `ENR`. If the `MP` flag is set the middle nodes of the edges are also returned on `ENR` beginning at position `KZAHL` + 1. Thus, the size of `ENR` must be `KZAHL` or 2 * `KZAHL`.

INTEGER*4 FUNCTION GEMKANTE(FACE,F1,F2,KZAHL,IER)

`FACE`	I	Array of faces
`F1`	I	First face
`F2`	I	Second face
`KZAHL`	I	Number of edges per face
`IER`	O	Error indicator

The function returns the number of the common edge of the faces `F1` and `F2`. If there is no common edge, `IER` is set and the function returns -1.

INTEGER*4 FUNCTION GEMPKT(KANTE,K1,K2,IER)

`KANTE`	I	Array of edges
`K1`	I	First edge
`K2`	I	Second edge
`IER`	O	Error indicator

The function returns the number of the common node of the edges `K1` and `K2`. If there is no common node `IER` is set and the function returns -1.

SUBROUTINE ITAUSCH(A,B)

`A,B`	I/O	Integer or real values

The routine swaps the values `A` and `B`.

INTEGER*4 FUNCTION Get_Part_Off(DGraph)

`DGraph`	I	DGraph data structure

The function returns the offset of the partitioning vector in `DGraph`. For more detail see section 2.2.14. Note that `Get_Part_Off`(`DGraph`) + 1 is the first position of the partition array.

INTEGER*4 Function Get_Wgt_Off(DGraph)

`DGraph`	I	DGraph data structure

The function returns the offset of the weight vector in `DGraph`. For more detail see section 2.2.14. Note that `Get_Wgt_Off`(`DGraph`) + 1 is the first position of the weight array.

SUBROUTINE Store_Partition(DGraph,Part)

`DGraph`	I/O	DGraph data structure
`Part`	I	Partition vector

The routine stores the current partitioning `Part` in the `DGraph` data stucture.

SUBROUTINE Print_Dgraph(DGraph)

`DGraph`	I	DGraph data structure

The routine displays all data stored in `DGraph`.

Chapter 9

Schematic program run

Figure 9.1 shows an schematic run of the program. Most of the functional blocks are associated with exactly one subroutine as indicated in Table 9.1.

There are two conditionals in the global program loop. One for the main mesh fixing and one for the data splitting. The first conditional switches the logical parameter `L_GROBNETZ`, which is only true if the main mesh is fixed. The fixing is done, if the number of elements exceeds the parameter `VOL_SOLL` which is defined as `VOL_SOLL` $= 2^{\texttt{NCUBE}} *$ `N_JE_PROC`. Note that the program runs just on processor 0 till the main mesh is fixed.

The second conditional checks first if `L_GROBNETZ` is true. Then it checks if the current hypercube dimension `LOC_CUBE` has already reached the maximal dimension `NCUBE`. If not the memory utilization is checked. If it exceeds the parameter `SPLIT_WERT` on a processor the data is split. Note that the memory utilization can range from 0 to 1, which corresponds to 0% to 100%.

Function block	Corresponding subroutine
Get user mesh	`NET_0`
Get parameters	`SETSTANDARD`
Refine mesh	`A_REFINE`
Assemble equation system	`A_ASSEMBLE`
Solve	`A_LOESEN`
Set main mesh	`SET_GROBNETZ`
Split data if needed and possible	`N_SPLIT`

Table 9.1: Main function blocks of *SPC-PMPo 3D*v4.x and the corresponding subroutines.

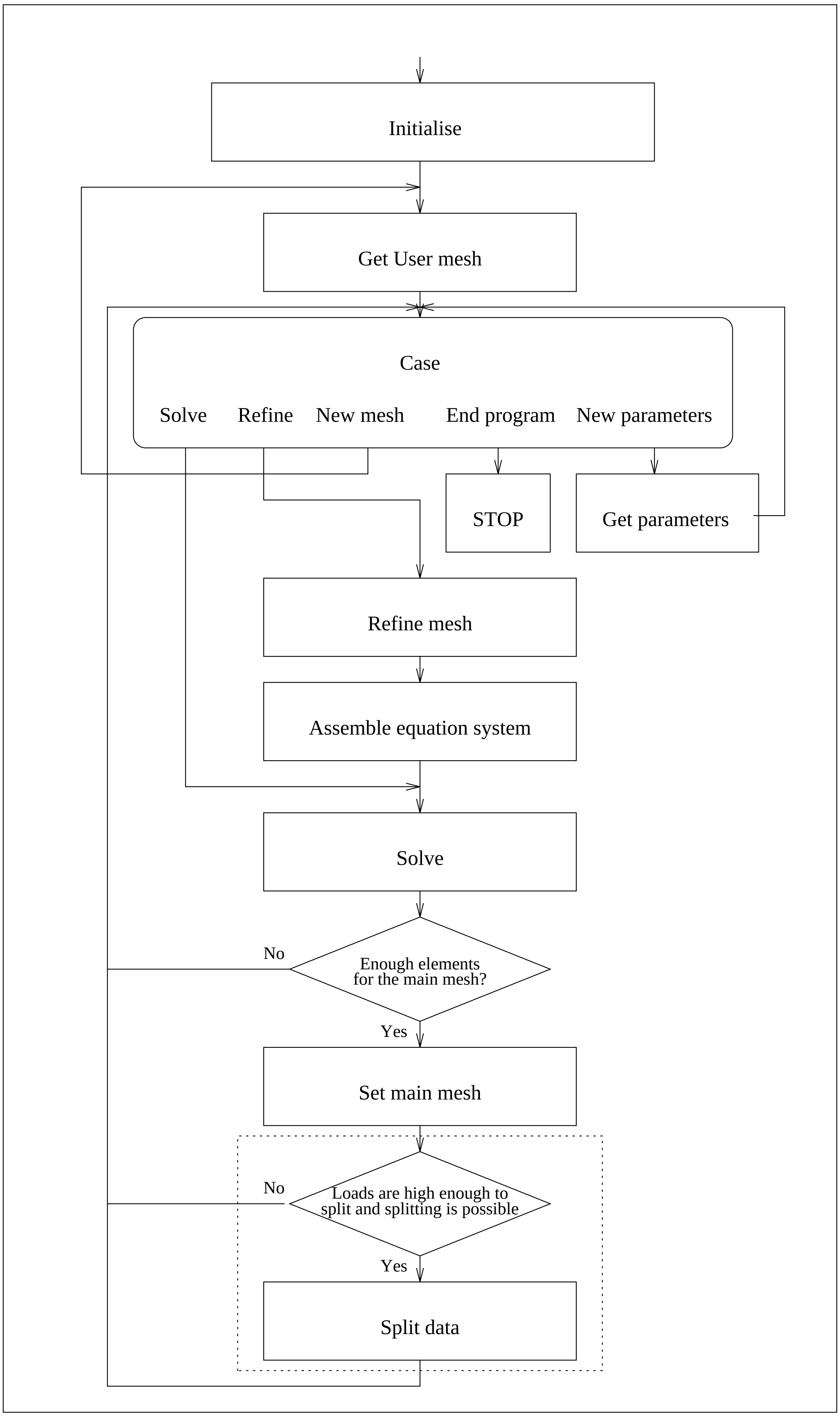


Figure 9.1: Flow chart of the schematic program run.

Bibliography

[1] 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.

[2] Th. Apel, F. Milde, and M. Theß. *SPC-PMPo 3D* — Programmer's Manual. Preprint SPC95_34, TU Chemnitz-Zwickau, 1995.

[3] Th. Apel and U. Reichel. *SPC-PMPo 3D V3.3* — User's Manual. Preprint SFB393/99-06, TU Chemnitz, Februar 1999.

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

[5] G. Haase, Th. Hommel, A. Meyer, and M. Pester. Bibliotheken zur Entwicklung paralleler Algorithmen. Preprint SPC95_20, TU Chemnitz–Zwickau, 1995. Updated version of SPC94_4 and SPC93_1.

[6] G. Kunert. *A posteriori error estimation for anisotropic tetrahedral and triangular finite element meshes.* PhD thesis, TU Chemnitz, 1999. Logos, Berlin, 1999.

[7] A. Meyer and M. Pester. Verarbeitung von Sparse-Matrizen in Kompaktspeicherform (KLZ/KZU). Preprint SPC94_12, TU Chemnitz–Zwickau, 1994.

[8] M. Pester. Behandlung gekrümmter Oberflächen in einem 3D-FEM-Programm für Parallelrechner. Preprint SFB393/97-10, TU Chemnitz, April 1997.

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

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.