Wenn man arbeitet, um anderen zu gefallen, kann es misslingen, die Dinge aber, die man gemacht hat, um selber zufrieden zu sein, haben stets Aussicht, irgendwann zu interessieren. Es ist unmöglich, daß es nicht Leute gäbe, die einiges Vergnügen an dem finden, was mir selber so viel Vergnügen bereitet hat.

MP, 1/2, 98

TR 93–5 (December 1993)
Contents

1 Introduction 1

2 KASKADE runtime interface 2
   2.1 C basics ........................................ 2
   2.2 Portability interface ................................ 2
   2.3 Application frame .................................. 4
   2.4 Fixed sized list management .......................... 4
   2.5 Application events .................................. 6
   2.6 Access rights ...................................... 8
   2.7 Command language .................................. 10
   2.8 Parameter handling .................................. 13
   2.9 Minigraphic ....................................... 15

3 Triangulation module 16
   3.1 Triangulations ..................................... 16
   3.2 Basic objects of a triangulation ..................... 17
   3.3 Organization of basic object lists .................... 20
   3.4 Global operations on a triangulation ................ 21
   3.5 Using a triangulation ................................ 22
   3.6 Procedures (Refinement/Deletion) .................... 22
   3.7 Customizing the module ............................. 26
   3.8 Command language interface ......................... 26

4 Node module 28
   4.1 Access management on basic objects .................. 28
   4.2 Node management ................................... 31
   4.3 Sparse matrices ................................... 34
   4.4 The matrix/vector multiplication method ............. 35
   4.5 Command language interface ......................... 37

5 Adaptive computations 38

6 Assembling module 39
   6.1 Defining the problem ............................... 39
6.2 Shape functions ........................................ 42
6.3 Local assembling .......................................... 44

7 Solve module ............................................. 48
  7.1 Interface for direct solvers ............................ 49
    7.1.1 Direct methods and their duties ................. 49
    7.1.2 Defining a direct method ....................... 51
    7.1.3 Command language interface .................... 52
  7.2 Interface for iterative solvers ...................... 52
    7.2.1 Iterative methods and their duties ............ 52
    7.2.2 Defining an iterative method .................. 53
    7.2.3 The preconditioner method ..................... 54
    7.2.4 Command language interface .................... 54
  7.3 Interface for error estimators ........................ 54
  7.4 Interface for refinement strategies .................. 55
  7.5 Interface for a completely adaptive solution ....... 56
    7.5.1 Break conditions ................................ 56
    7.5.2 Command language interface .................... 56

8 Numerical methods ...................................... 59
  8.1 Direct solvers ......................................... 59
  8.2 Iterative solvers ....................................... 59
  8.3 Estimator methods ..................................... 59
  8.4 Refinement methods .................................... 60
  8.5 Preconditioners ....................................... 60

9 Graphic module .......................................... 61
  9.1 Handling graphical ports .............................. 61
  9.2 Drawing ................................................. 61
  9.3 Command language interface .......................... 62

10 KASKADE applications .................................. 64
  10.1 ELLKASK 2D ........................................... 64
  10.2 ELLKASK 3D ........................................... 64
  10.3 KASTIO ............................................... 65
References

A: Command Language Interface
   A.1 Command language syntax .......................... 68
   A.2 Command language survey .......................... 68
   A.3 Alphabetical list of commands ....................... 69
   A.4 Graphic parameters ................................ 85

B: File formats ........................................ 87

C: Defining your own problem ............................. 90
   C.1 User functions for the stationary heat transfer equation .... 90
   C.2 Defining the new problem ........................... 92
   C.3 Invoking the new problem ........................... 93

Index .................................................. 95

List of Figures

1  Field of an edge .................................. 19
2  Position in a triangle ............................... 20
3  Red refinement ..................................... 23
4  Blue refinement ..................................... 24
5  Middle triangle will be red refined ................. 24
6  Right triangle will be red refined .................. 25
7  Green closure ..................................... 25
8  Node storage at points and edges ................. 32
9  Main iteration loop of an adaptive solution ....... 58
10 Triangulation of the example 'Kreis mit Schlitz' .... 89

List of Tables

1  KASKADE standard data types ....................... 2
<table>
<thead>
<tr>
<th>Page</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>KASKADE standard constants</td>
</tr>
<tr>
<td>3</td>
<td>Constants for fixed size list management</td>
</tr>
<tr>
<td>4</td>
<td>Error constants for fixed size list management</td>
</tr>
<tr>
<td>5</td>
<td>Constants for application event management</td>
</tr>
<tr>
<td>6</td>
<td>Error constants for application event management</td>
</tr>
<tr>
<td>7</td>
<td>Constants for access right management</td>
</tr>
<tr>
<td>8</td>
<td>Error constants for access right management</td>
</tr>
<tr>
<td>9</td>
<td>command data structure</td>
</tr>
<tr>
<td>10</td>
<td>Modes read from the command definition file</td>
</tr>
<tr>
<td>11</td>
<td>Constants for the command language module</td>
</tr>
<tr>
<td>12</td>
<td>Constants for parameter management</td>
</tr>
<tr>
<td>13</td>
<td>Error constants for parameter management</td>
</tr>
<tr>
<td>14</td>
<td>Fields of the triangulation data structure</td>
</tr>
<tr>
<td>15</td>
<td>Fields of the point data structure</td>
</tr>
<tr>
<td>16</td>
<td>Values of boundP fields</td>
</tr>
<tr>
<td>17</td>
<td>Fields of the edge data structure</td>
</tr>
<tr>
<td>18</td>
<td>Values of type fields</td>
</tr>
<tr>
<td>19</td>
<td>Fields of the triangle data structure</td>
</tr>
<tr>
<td>20</td>
<td>Additional fields of the triangulation data structure</td>
</tr>
<tr>
<td>21</td>
<td>Apply procedures</td>
</tr>
<tr>
<td>22</td>
<td>Selection codes</td>
</tr>
<tr>
<td>23</td>
<td>Application events of the triangulation module</td>
</tr>
<tr>
<td>24</td>
<td>Fields of the nodes data structure (all type int)</td>
</tr>
<tr>
<td>25</td>
<td>Access macros, first letter: data type, second letter: object</td>
</tr>
<tr>
<td>26</td>
<td>Access macros for node storage</td>
</tr>
<tr>
<td>27</td>
<td>Preset fields of nodes</td>
</tr>
<tr>
<td>28</td>
<td>Macros to access sparse matrices</td>
</tr>
<tr>
<td>29</td>
<td>The matMulMethod data structure</td>
</tr>
<tr>
<td>30</td>
<td>Fields of the problemType data structure</td>
</tr>
<tr>
<td>31</td>
<td>Fields of the problem data structure</td>
</tr>
<tr>
<td>32</td>
<td>Return codes for the problem defining user routines</td>
</tr>
<tr>
<td>33</td>
<td>The integData data structure</td>
</tr>
<tr>
<td>34</td>
<td>The localData data structure</td>
</tr>
</tbody>
</table>
1 Introduction

The code version KASKADE to be presented here is written in ANSI style C. The modules of KASKADE are designed to allow reuse and extensions. Well-known software engineering techniques like

1. information hiding,
2. object oriented interfacing,
3. portability etc.

are used. Thus we hope to support rapid prototyping of non trivial applications.

KASKADE consists of a collection of modules (a library) and some prototype applications (main programs and extensions). The functionality of each module is specified by

1. a set of data types,
2. macro definitions,
3. a collection of procedures, and
4. specification of events generated.

If necessary, the modules are extended by

1. a command language interface and
2. internal/external test features.

The manual includes recipes to change and extend KASKADE and a “complete” interface description of the modules. We try to hide information which might change in the near future. The program is still under development, there will be changes.

Errors, problem reports, or any comments should be forwarded to the authors at the Konrad–Zuse–Zentrum (ZIB), e-mail addresses

roitzsch@sc.zib-berlin.de
lang@sc.zib-berlin.de
erdmann@sc.zib-berlin.de.

This manual contains a description of the KASKADE 2-D stuff. KASKADE will evolve to a 2D and 3D FEM tool box. Many routines described here are valid in both “worlds”.

1
2 KASKADE runtime interface

The following modules are basically independent of the KASKADE application. They can be used by other C programs, too.

2.1 C basics

The reader should know C, at least he should be familiar with the concept of pointers (addresses). Reading the KASKADE sources is recommended. We use the following naming conventions.

1. procedure names start with a capital letter,
2. variable names start with a small letter,
3. constant, type, and macro names should use only capital letters (with some exceptions).

Some standard types are used in all KASKADE sources (they are defined in zibutil.h), see Table 1. The use of the KASKADE runtime library is recommended to benefit of the non–KASKADE specific software which is used by the KASKADE modules.

<table>
<thead>
<tr>
<th>real</th>
<th>floating point numbers</th>
<th>float or double</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>pointers</td>
<td>char*</td>
</tr>
<tr>
<td>proc</td>
<td>procedure</td>
<td>int(*)()</td>
</tr>
<tr>
<td>realproc</td>
<td>real procedure</td>
<td>real(*)()</td>
</tr>
<tr>
<td>ptrproc</td>
<td>address procedure</td>
<td>ptr(*)()</td>
</tr>
</tbody>
</table>

Table 1: KASKADE standard data types

These types should be used like the “popular” constants in Table 2.

2.2 Portability interface

It is good practice to use these routines to isolate system dependencies. Direct calls of malloc, standard input/output routines should be avoided to make the program more adaptable to different environments. All routines and global variables are declared in the zibutil.h header file.
Table 2: KASKADE standard constants

<table>
<thead>
<tr>
<th>nil</th>
<th>nil pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>ZERO</td>
<td>0.0</td>
</tr>
<tr>
<td>ONE</td>
<td>1.0</td>
</tr>
<tr>
<td>HALF</td>
<td>0.5</td>
</tr>
<tr>
<td>REALPI</td>
<td>3.14159265358979323846</td>
</tr>
<tr>
<td>REALPI2</td>
<td>1.57079632679489661923</td>
</tr>
<tr>
<td>SQRT2</td>
<td>1.41421356237309514547</td>
</tr>
</tbody>
</table>

void *ZIBAlloc(long size)
substitutes the standard malloc routine. The size parameter is now of type long, thus allowing the handling of bigger chunks of memory. The implementation assumes a small number of calls. (Each call might be expensive.) For frequent calls with small sizes use the routines of the fixed size list management (see Section 2.4). The amount of memory allocated can be accessed via the variable allocMem of type long.

void *ZIBFree(void *buf)
is the counter part to free. In the current implementation allocMem is updated (reduced)! A nil is always returned.

real ZIBSeconds()
returns the cpu-time in seconds with a machine dependent precision.

int ZIBReadFile(char *name, char **buffer, char *extension)
reads the complete file name (optional with an extension extension) into buffer. The storage buffer is allocated automatically (through ZIBAlloc) and may be freed by the user with ZIBFree.

void ZIBStdOut(char *s)
should be used to write messages. It depends on the environment how the request is fulfilled. Output may go to a special window, standard output, or a file (see ZIBOutFile). The technique used to substitute calls of printf is:

printf(........)  \Rightarrow  sprintf(globBuf,........);
ZIBStdOut(globBuf);
The buffer globBuf is externally defined and should be of sufficient size for standard text operations (actually 256 bytes).

```c
void ZIBOutFile(char *s)
```
can be used to redirect standard output via ZIBStdOut to a file with name s.

```c
char *ZIBStdLine(char *prompt)
```
reads a line from the standard input after prompting the user.

### 2.3 Application frame

We use the skeleton for application programs which is under development at ZIB. Documentation (in german) is available on request [8].

### 2.4 Fixed sized list management

The fixed sized list management includes routines to handle many memory blocks of fixed, small sizes as they are used to store the data for points, edges, triangles, boxes, etc. To reduce the overhead of direct calls to ZIBAlloc (which in some implementations is big), buckets (blocks of storage) are allocated. Each set of fixed size elements is identified by a positive integer.

All messages are done through ZIBStdOut, all storage allocation through ZIBAlloc.

The constants and routines are declared in the `fixedsize.h` header file. The constants are explained in the Tables 3 and 4. For complete documentation see the source listing `fixedsize.tex`.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_FIXED_LISTS</td>
<td>100</td>
<td>maximum numbers of lists</td>
</tr>
<tr>
<td>BAD_FIXED_LISTS_ID</td>
<td>-1</td>
<td>wrong list id number</td>
</tr>
<tr>
<td>MAX_SMALL_SIZE</td>
<td>161</td>
<td>greatest small list size</td>
</tr>
<tr>
<td>SMALL_BUCKET_SIZE</td>
<td>100</td>
<td>minimal bucket size</td>
</tr>
</tbody>
</table>

Table 3: Constants for fixed size list management

These routines are used by the software managing triangulations, nodes, and sparse matrices. If the command line interface is used, a command informing on the usage of these routines is supported. The command language interface is the `inffixed` command.
int InitFixedLists(int maxLists, void (*UserError)(int, char*))

initializes the module. It should only be called once at the start of the user program. Negative values of maxLists reserve space for MAX_FIXED_LISTS lists of fixed size elements. The user may supply a routine to handle error messages. This routine is called with an error number (Table 4) and a string as parameters. If UserError==nil the string will be printed with ZIBStdOut.

int CreateList(int lng, int bucketSize, char *name)

creates a single list name with the fixed size lng and returns the list identifier. Each time when no elements are available in the free list, bucketSize elements will be newly allocated.

void ReturnList(int no)

frees all elements of list no. The memory is returned to the operating system. The user is responsible to keep no pointers to one of these elements alive!

ptr GetElem(int no)

gets a pointer to a new element of list no, i.e. it has the size common to all elements of this list.

void ReturnElem(int no, ptr *elem)

returns an element elem of list no into the internal free list. Note that no memory is returned to the operating system. The user is responsible to keep no reference to this element alive!

ptr GetSmallElem(int size)

This routine replaces the function of malloc. Small elements are collected in buckets by using GetElem for elements of the same size.
void ReturnSmallElem(int size, ptr *elem)
        returns elem. The user must supply the correct size value.

void InformLists(void(*UserProc))(int no, int elemSize, int bucketSize, int noOfBuckets, int noOfFreeElems, char* name))
        calls the user routine UserProc for each defined list with parameters no, elemSize, bucketSize, noOfBuckets, noOfFreeElems, and name.

int InfFixed(command* cmd)
        prints all available information on existing lists in tabular form. The user can inform on the content of one list by supplying its name.

2.5 Application events

The application event routines include code to connect “loosely coupled” modules. Two methods are available to process events which are defined by the user program. One is the immediate processing of user routines at the occurrence of an event. The other is queuing of events leaving the user the task of writing a sort of main event loop.

The constants and routines are declared in the appevents.h header file. The constants are explained in Tables 5 and 6. For complete documentation see the source listing appevents.tex.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_EVENTS</td>
<td>100</td>
<td>maximum of events</td>
</tr>
<tr>
<td>MAX_EVENT_TYPES</td>
<td>100</td>
<td>maximum of event types</td>
</tr>
<tr>
<td>MAX_EVENT_PROCS</td>
<td>10</td>
<td>maximum of event procedures</td>
</tr>
<tr>
<td>POS_BEFORE</td>
<td>-1</td>
<td>insert procedure before</td>
</tr>
<tr>
<td>POS_SUBSTITUTE</td>
<td>0</td>
<td>substitute procedure</td>
</tr>
<tr>
<td>POS_AFTER</td>
<td>1</td>
<td>insert procedure after</td>
</tr>
</tbody>
</table>

Table 5: Constants for application event management

int InitAppEvents(int maxE, int maxT,
        void (*UserError)(int,char*))
        initializes the module. It should only be called once at the start of the user program. maxE is the maximal number of events which will be held in the queue (default MAX_EVENTS). Only maxT different event
types are allowed (default MAX_EVENT_TYPES). The user may supply a routine to handle error messages. This routine is called with an error number (Table 6) and a string as parameters. If UserError==nil the string will be printed with ZIBStdOut.

\begin{verbatim}
int DefineEventType(char *name, int immediateP)
defines a new application event type. An identifier for this type of event is returned. If immediateP=true new events getting in through SendAppEvent are processed immediately and are not hold in the event queue.

int SetEventProc(int type, int position, void (*RefProc)(ptr, int, char*),
void (*UserProc)(ptr, int, char*), char *name)
adds the user procedure UserProc at the beginning (if position is POS_BEFORE) or end (if position is POS_AFTER) of the procedure list for event type type if RefProc==nil. RefProc!=nil can be used to substitute or insert (before or after) RefProc.

int SendAppEvent(int type, char *buffer, int lng, char *name)
processes an event of type type either by calling immediately the user procedures or by putting the event in the event queue. The event is described by (buffer,lng) and a name. The name can be used for debugging purposes.

int GetAppEvent(int *type, char **buffer, int *lng, char **name)
releases an event from the event queue. Returning false denotes an empty queue.
\end{verbatim}
int InfEventTypes(void (*UserProc)(int, char*, char**, int, int))
calls UserProc for all defined event types with the parameters int
type, char *typeName, char **procNames, int maxProcNames, and
int noOfEvents.

int InfEvents(void(*UserProc)(int, char*, ptr, int, char*))
calls the procedure UserProc for all events in the event queue. Parame-
ters for UserProc are int type, char *typeName, ptr value, int
lng, char *EventName.

void CallProcs(int type, ptr value, int lng, char *name)
calls all procedures belonging to event type with parameters value,
lng, and name.

int SetEventTrace(int tr)
switches an internal trace on/off.

int EventsCmd(command* cmd)
prints information on the defined event types and pending events. At
the command language level, parameters pending and defined select
these features. Additionally an internal trace can be toggled.

2.6 Access rights

These routines allow the management of access to a workspace of a fixed
number of bytes. It is used by KASKADE to permit the access to the
associated arrays stored in the data structure for points, edges, and triangles.
To get positions in the associated arrays the access manager is asked for free
fields. All modules using the access manager should return the local storage
not used anymore by calls to the access manager.

The concept of an access list is rather simple. An access list consists of \( n \)
consecutive fields which are set available by a call to CreateAccessList.
Access rights to blocks of free fields are requested by the GetAccess routine.
The implementation is done by a prototype array of integers. A value of \(-1\)
denotes a free byte. A block of used space contains in each field the index of
the first byte. This sequence ends with a value different from the start value.
All messages are done through ZIBStdOut or a user routine, all storage allo-
cation through ZIBAlloc.

The constants and routines are declared in the access.h header file. The
constants are explained in the Tables 7 and 8. For complete documentation
see the source listing `access.tex`.

<table>
<thead>
<tr>
<th>MAX_ACCESSLIST</th>
<th>100</th>
<th>maximal number of access lists</th>
</tr>
</thead>
</table>

Table 7: Constants for access right management

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_SECOND</td>
<td>-1</td>
<td>second call of init routine</td>
</tr>
<tr>
<td>ACC_LNG_NEG</td>
<td>-2</td>
<td>list length negative</td>
</tr>
<tr>
<td>ACC_ALL_LISTS_USED</td>
<td>-3</td>
<td>all lists in use</td>
</tr>
<tr>
<td>ACC_BAD_INDEX</td>
<td>-4</td>
<td>bad access list index</td>
</tr>
<tr>
<td>ACC_NO_INDEX</td>
<td>-5</td>
<td>list not defined</td>
</tr>
<tr>
<td>ACC_NO_FIELD</td>
<td>-6</td>
<td>field not in use</td>
</tr>
<tr>
<td>ACC_MEMORY</td>
<td>-7</td>
<td>not enough memory</td>
</tr>
<tr>
<td>ACC_FULL</td>
<td>-8</td>
<td>request for space failed</td>
</tr>
</tbody>
</table>

Table 8: Error constants for access right management

```c
int InitAccess(int maxLists, void (*UserError)(int, char*))
```

initializes the access manager. The parameter `maxLists` defines the maximum number of access lists. This routine is called internally if necessary (with `maxLists=MAX_ACCESSLIST`). The user may supply a routine to handle error messages. This routine is called with an error number (Table 8) and a string as parameters. If `UserError=nil` the string will be printed with `ZIBStdOut`.

```c
int CreateAccessList(int lng, char *name)
```

initializes the access list `name` and returns an integer identifier for this list. The size of list will be `lng`. All its elements are free.

```c
void ReturnAccessList(int listId)
```

frees the internal memory for the prototype array of `listId`.

```c
int GetAccess(int listId, int lng, int alignment, char *elemName)
```

returns an identifier (the position) of a sequence of `lng` free elements in access list `listId`. The `alignment` parameter controls the alignment of the requested sequence of bytes. It should be set to the size of the type it will hold.
void ReturnAccess(int listId, int elemId)
    frees element elemId in access list listId for future use.

void InformAccessLists(void(*UserProc)(int,char*,int,int,int*,
    char**), int listId)
    calls the user routine UserProc for one or all (listId==-1) existing access lists with the parameters int listId, char *name, int accessLng, int maxUsed, int *protoTypes, and char **elemNames.

int AccessTrace(int mode)
    switches the internal trace mode on/off.

int InfAccess(command *cmd)
    prints all information about current access lists. Command language parameters traceon and traceoff can be used to toggle the internal trace on/off.

2.7 Command language

These routines are used to implement the interface between a user (at a terminal or writing a do-file, see Appendix A) and the application code. The interface uses the command data structure (see Table 9) to pass the information to application procedures.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>no</td>
<td>command number</td>
</tr>
<tr>
<td>int</td>
<td>noOfPars</td>
<td>number of parameters</td>
</tr>
<tr>
<td>char**</td>
<td>pars</td>
<td>array of parameters</td>
</tr>
<tr>
<td>char**</td>
<td>names</td>
<td>predefined parameter names</td>
</tr>
<tr>
<td>char*</td>
<td>results</td>
<td>return string (used in tcl embedding)</td>
</tr>
<tr>
<td>int</td>
<td>verbose</td>
<td>level of verbosity (0=silent)</td>
</tr>
</tbody>
</table>

Table 9: command data structure

The “first” parameter cmd.pars[0] is always the actual command name, the parameter list ends with an additional nil pointer. Therefore the length of cmd.pars is cmd.noOfPars+2. The user has to check the actual parameters against the predefined parameters. The routine CheckName will ease this task.

The list of predefined commands is read from a command definition file. This file consists of information for setting modes (i.e. the prompt), command
names followed by the command number and short description, and lists of parameters of commands. A command definition file consists of two parts separated by a blank line. In the first part some modes (see Table 10) may be set. Each of these lines start with ' ' and the mode name followed by the new value. The mode names are collected in Table 10.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Prompt</td>
<td>'string' command prompt</td>
</tr>
<tr>
<td>$MaxPar</td>
<td>number maximal number of parameters</td>
</tr>
<tr>
<td>$Escape</td>
<td>character escape character</td>
</tr>
<tr>
<td>$Comment</td>
<td>character comment character</td>
</tr>
<tr>
<td>$Quote</td>
<td>character additional quote character</td>
</tr>
<tr>
<td>$CmdDelim</td>
<td>character additional command delimiting character</td>
</tr>
<tr>
<td>$ParDelim</td>
<td>character additional parameter delimiting character</td>
</tr>
</tbody>
</table>

Table 10: Modes read from the command definition file

The second part defines the command names the runtime system should recognize. Each command name on a new line is followed by the command number and a string which will be used as a short description of the command. After the definition of the command names an optional list of keywords to given command numbers follows. Each list starts with a ' ' and the command number on an extra line.

After calling the routine InitCommands (reading the command definition file) the linking between the unique command number and the address of the application procedure has to be done dynamically by calls to SetCommand.

The constants and routines are declared in the commands.h header file. The constants are explained in Table 11. For complete documentation see the source listing commands.c.

```c
int InitCommands(char *path, char *name)
    InitCommand returns true if the command file is read successfully.

int DoCommands(char *line, void(*UserProc)(char*))
    puts the string line on the command stack. UserProc is called after
    the last command from line is executed. (Can be used to return the
    storage for line.)

int ExecCommand(command *cmd)
    executes a command by calling the procedure identified by the com-
    mand number in cmd.
```
command* GetNextCommand()
extracts the next command from the command stack. This procedure
does the syntax analysis and returns the result as a pointer to the
cmd data structure.

void CmdMainLoop()
processes commands from some standard input.

void SetCommand(int no, int (*UserProc)(command*))
links the procedure address UserProc to the command number no.

int AddCmdPar(char *cmdName, char *parName)
adds parName to the list of parameters of the cmdName command. The
position of the parameter in this list is returned.

int SetVerbose(int newVerbose)
sets the default value for the verbose field of the cmd data structure.

int CheckName(char **text, char **names, int charClass)
tests if a string is in a keyword list.

char *ConvertString(char **text, char *target, int max)
converts (substitutes escape sequences) string *text to *target. The
pointers are updated. max is used to stop conversion.

int ParsCheck(command *cmd, int min, int max)
checks if min <= cmd->noOfPars <= max and prints an appropriate error
message.

<table>
<thead>
<tr>
<th>MAX_COM</th>
<th>1000</th>
<th>maximum number of commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_PAR</td>
<td>100</td>
<td>maximum number of parameters</td>
</tr>
<tr>
<td>endClass</td>
<td>127</td>
<td>character class: file end</td>
</tr>
<tr>
<td>commandClass</td>
<td>10</td>
<td>character class: command end</td>
</tr>
<tr>
<td>parameterClass</td>
<td>5</td>
<td>character class: parameter end</td>
</tr>
<tr>
<td>nameClass</td>
<td>5</td>
<td>character class: name end</td>
</tr>
<tr>
<td>quoteClass</td>
<td>1</td>
<td>character class: string end</td>
</tr>
</tbody>
</table>

Table 11: Constants for the command language module
int ReadIntPar(command *cmd, int k, int *intVal)
    converts an integer from (cmd->pars)[k] and stores the value to *intVal.

int ReadRealPar(command *cmd, int k, real *realVal)
    converts a real from (cmd->pars)[k] and stores the value to *realVal.

void SetLibAddresses(int sel)
    predefines the basic commands of the modules described in Chapter 2.
    sel==-1 defines all commands.

2.8 Parameter handling

The parameter module includes routines to handle named parameter lists. A parameter list itself contains a list of parameter values of fixed size. A list of parameter names and a list of named values may be maintained. All messages are done through ZIBStdOut or a user output procedure (see InitParams). All storage is allocated by ZIBAlloc.

The constants and routines are declared in the params.h header file. The constants are explained in the Tables 12 and 13. For complete documentation see the source listing params.tex.

<table>
<thead>
<tr>
<th>MAX_PARAMSLISTS</th>
<th>100</th>
<th>maximum number of parameter list</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_INT</td>
<td>0</td>
<td>constant denoting type int</td>
</tr>
<tr>
<td>T_REAL</td>
<td>1</td>
<td>constant denoting type real</td>
</tr>
<tr>
<td>T_BOOL</td>
<td>2</td>
<td>constant denoting type bool</td>
</tr>
</tbody>
</table>

Table 12: Constants for parameter management

| EPM_DOUBLE_INIT | -1 | second call of init routine     |
| EPM_MEMORY      | -2 | not enough memory               |
| EPM_ALREADY_USED| -3 | parameter list name already in use |
| EPM_TOO_MANY    | -4 | too many parameter lists        |
| EPM_NOT_DEFINED | -5 | parameter list not defined      |
| EPM_NOT_FOUND   | -6 | parameter name not defined      |

Table 13: Error constants for parameter management
int InitParam(int maxParams, void (*UserError)(int,char*))
    should be called once for initialization. The routine is called internally
    with maxParams=MAX_PARAMS_LISTS by some of the other routines (if
    the user did not initialize the module). The user may supply a routine
    to handle error messages. This routine is called with an error number
    (Table 13) and a string as parameters. If UserError=nil the string
    will be printed with ZIBStdOut.

ptr NewParamList(ptr buf, char *listName, int noOfParams,
    int valueSize, char **names, int type, int noOfValNames,
    char **valNames, ptr vals,
    int(*UserParamChanged)(char*,char*,int),
    int(*UserListChanged)(char*))

uses buf as storage for a parameter list or, if buf=nil allocates new
storage. listName is checked for double definitions. The result of the
routine is the address to an array of noOfParams blocks of valueSize
bytes of memory.

Parameter values may be named to allow a user-friendly input via the
setpar command. A list of name-value pairs with length noOfValNames
is defined by valNames and vals.

If the user wants to be notified on changes of parameters or the complete
list a user routine UserParamChanged or respectively UserListChanged
may be supplied.

int ReturnParamList(char *listName)
    releases the memory of a parameter list listName.

int GetParam(char *listName, char *name, ptr *value)
    returns the address of the value of a parameter which is defined by the
    parameter listName and name.

GetParamList(char *listName, char ***names, ptr *values,
    int *noOfParams, int *type, int *noOfValNames,
    char ***valNames, ptr *vals)

returns the pointer to the arrays of parameter names, parameter values,
and the pointers to information on named parameter values.

char **GetListNames(int *lng)
    returns a pointer to an array of all list names.
int ChangeParam(char *listName, char *name, ptr value, char *valName)
calls UserParamChanged with parameters listName, name, value, and valName. If valName!=nil the named parameter value is looked up in the valNames table of the parameter list and is set to the new value value.

int ChangedParamList(char *listName)
notifies the parameter management module of a change of values. The routine UserListChanged is called.

int InfPar(command *cmd)
informs on parameter lists and parameter values on the command language level.

int SetPar(command *cmd)
sets parameter values on the command language level.

2.9 Minigraphic

See the report by Andreas Wendt, Rainer Roitzsch [14].
3 Triangulation module

The triangulation module of KASKADE handles all basic operations related to the data structures of a triangulation. It includes methods to operate on sets of points, edges or triangles too. These sets are defined internally by the module (like “all edges”) or may be defined by the user through list operations or by predicate functions. Furthermore, this module contains the procedures to refine a triangulation (including the generation of green refined triangles to preserve regularity) and the procedures to delete triangles.

Triangulations are described by a global data type which allows the usage of more than one triangulation in a program. This includes the possibility to freeze a triangulation, working on a copy, or solving with different algorithms on a triangulation.

This document contains a description of the module, listing all relevant data types, constants, operations on these data structures, and procedures to use a triangulation. It should include all necessary information to use the module.

The constants and routines are declared in the triang2.h header file. The module is initialized by a call of InitTriang.

The data structures are developed by P. Leinen [11]. The following routines initialize the module.

```c
int InitTriang()
    initializes the module.

void SetTriAddresses(int sel)
    makes the command language interface available (sel==1).
```

3.1 Triangulations

A triangulation consists of sets of points, edges, and triangles which form a subdivision of a two-dimensional area. The triangles should not overlap. The data structure used to describe a triangulation contains many fields which should be hidden to the normal user. Such fields are needed to find the data for points, edges, and triangles. Other fields are open to the user and allow fast access to some descriptive data, see Table 14.

The data for the triangulation currently in use is defined via the global variable actTriang. All triangulations are linked, starting with firstTriang. actTriang is changed by calls to the procedures which create, delete, or select triangulations, see Section 3.4.
The set of procedures to access the basic elements of a triangulation are described in Section 3.5. The handling of the adaptive refinement of an existing triangulation is presented in Section 3.6.

### 3.2 Basic objects of a triangulation

The data type for points includes at least the \((x, y)\)-coordinates, a boundary type descriptor, some internal data, and a byte array (the associated array). The list of fields is collected in Table 15.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec</td>
<td>char*</td>
<td>associated storage</td>
</tr>
<tr>
<td>boundP</td>
<td>char</td>
<td>boundary condition type, see Table 16</td>
</tr>
<tr>
<td>mark</td>
<td>char</td>
<td>marking field</td>
</tr>
<tr>
<td>classA</td>
<td>char</td>
<td>additional classification</td>
</tr>
<tr>
<td>indexP</td>
<td>int</td>
<td>number of point</td>
</tr>
<tr>
<td>x</td>
<td>real</td>
<td>(x)-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>real</td>
<td>(y)-coordinate</td>
</tr>
<tr>
<td>next</td>
<td>PT*</td>
<td>pointer to the next point</td>
</tr>
<tr>
<td>last</td>
<td>PT*</td>
<td>pointer to the previous point</td>
</tr>
</tbody>
</table>

Table 15: Fields of the point data structure PT

The usage of the associated array vec is clarified in Chapter 4 on node management. The usage of the classA field allows the handling of geometrically dependent, problem specific information (like material constants). The mark field is free for (software) local use, but should be handled carefully to avoid multiple use.
INTERIOR 0 inner element
DIRICHLET 1 Dirichlet boundary
NEUMANN 2 Neumann boundary
CAUCHY 3 Cauchy boundary

Table 16: Values of boundP fields

The structural data next and last should not be used.

The data type for edges includes at least two pointers to the end points, a boundary type descriptor, refinement type descriptor, some internal data, and a byte array. The list of fields is collected in Table 17.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec</td>
<td>char*</td>
<td>associated storage</td>
</tr>
<tr>
<td>boundP</td>
<td>char</td>
<td>boundary condition type, see Table 16</td>
</tr>
<tr>
<td>mark</td>
<td>char</td>
<td>marking field</td>
</tr>
<tr>
<td>type</td>
<td>char</td>
<td>type of edge, see Table 18</td>
</tr>
<tr>
<td>classA</td>
<td>char</td>
<td>additional classification</td>
</tr>
<tr>
<td>isBlueDiagonal</td>
<td>char</td>
<td>blue diagonal refined</td>
</tr>
<tr>
<td>depth</td>
<td>short</td>
<td>depth of edge</td>
</tr>
<tr>
<td>p1</td>
<td>PT*</td>
<td>first point</td>
</tr>
<tr>
<td>p2</td>
<td>PT*</td>
<td>second point</td>
</tr>
<tr>
<td>pm</td>
<td>PT*</td>
<td>midpoint</td>
</tr>
<tr>
<td>t1</td>
<td>PT*</td>
<td>first neighbor triangle</td>
</tr>
<tr>
<td>t2</td>
<td>PT*</td>
<td>second neighbor triangle</td>
</tr>
<tr>
<td>next</td>
<td>EDG*</td>
<td>pointer to the next edge</td>
</tr>
<tr>
<td>last</td>
<td>EDG*</td>
<td>pointer to the previous edge</td>
</tr>
<tr>
<td>father</td>
<td>EDG*</td>
<td>pointer to father edge</td>
</tr>
<tr>
<td>firstSon</td>
<td>EDG*</td>
<td>pointer to first son edge</td>
</tr>
</tbody>
</table>

Table 17: Fields of the edge data structure EDG

The geometric information of edges is depicted in Figure 1.

The usage of the associated array vec is clarified in Chapter 4 on node management. The usage of the classA field allows the handling of geometrically dependent, problem specific information (like material constants). The mark field is free for (software) local use, but should be handled carefully to avoid multiple use. The type tag gives some information on the generation of the edge, see Table 18.
<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T.WHITE</td>
<td>0</td>
</tr>
<tr>
<td>T.GREEN</td>
<td>1</td>
</tr>
<tr>
<td>T.RED</td>
<td>2</td>
</tr>
<tr>
<td>T.BLUE</td>
<td>3</td>
</tr>
<tr>
<td>T.NOTHING</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 18: Values of type fields

![Figure 1: Field of an edge](image)

The structural data `next`, `last`, `father`, and `firstSon` should not be used. The midpoint of an edge should be accessed with the `PM` macro.

The data type for triangles includes at least three pointers to the edges, refinement type descriptor, an integer number to specify the refinement depth, some internal data, and a byte array. For convenience the pointers to the vertices are stored too. The list of fields is collected in Table 19.

The geometric information of triangles is depicted in Figure 2.

The usage of the associated array `vec` is clarified in Chapter 4 on node management. The usage of the `classA` allows the handling of geometrically dependent, problem specific information (like material constants). The definition of `depth` is given in Section 3.6. The `mark` field is free for (software) local use, but should be handled carefully to avoid multiple use. The `type` tag gives some information on the generation of the triangle.

The structural data `next`, `last`, `father`, and `firstSon` should not be used. The sons of a triangle should be accessed with the `TREDxSON` macros (`x` is 1, 2, 3, or 4).
vec char* associated storage
mark char marking field
type char type of triangle, see Table 18
classA char additional classification
depth short depth of triangle
p1 PT* first point
p2 PT* second point
p3 PT* third
e1 EDG* first edge
e2 EDG* second edge
e3 EDG* third edge
next TR* pointer to the next triangle
last TR* pointer to the previous triangle
father TR* pointer to father triangle
firstSon TR* pointer to first son triangle

Table 19: Fields of the triangle data structure TR

Figure 2: Position in a triangle

3.3 Organization of basic object lists

Points, edges, and triangles are organized in double linked lists. New points are inserted at the end of the point list which can be accessed through the firstPoint and lastPoint fields of the TRIANGULATION data structure (see Table 20). The points of the coarse mesh are collected as an array in the initPoints field.

The efficient implementation of hierarchical basis and bpx preconditioners yield a more sophisticated handling of edges and triangles. New edges and
triangles are appended in sublists which correspond to their depth. These sublists start at the \texttt{depthFirstEdges} and \texttt{depthFirstTriangles} fields. All sublists for edges (and triangles) are linked to one list respectively. The beginning and end is stored in the \texttt{firstEdge}, \texttt{lastEdge}, \texttt{firstTriangle}, and \texttt{lastTriangle} fields.

\begin{tabular}{|l|l|}
\hline
\texttt{maxDepthFirsts} & \texttt{int} & \texttt{max depth} \\
\texttt{noOfInitPoints} & \texttt{int} & \texttt{number of initial points} \\
\texttt{noOfInitEdges} & \texttt{int} & \texttt{number of initial edges} \\
\texttt{noOfInitTriangles} & \texttt{int} & \texttt{number of initial triangles} \\
\texttt{firstPoint} & \texttt{PT*} & \texttt{first point} \\
\texttt{lastPoint} & \texttt{PT*} & \texttt{last point} \\
\texttt{initPoints} & \texttt{PT[]} & \texttt{array of initial points} \\
\texttt{firstEdge} & \texttt{EDG*} & \texttt{first edge} \\
\texttt{lastEdge} & \texttt{EDG*} & \texttt{last edge} \\
\texttt{initEdges} & \texttt{EDG[]} & \texttt{array of initial edges} \\
\texttt{depthFirstEdges} & \texttt{EDG**} & \texttt{array of sublists} \\
\texttt{firstTriangle} & \texttt{TR*} & \texttt{first triangle} \\
\texttt{lastTriangle} & \texttt{TR*} & \texttt{last triangle} \\
\texttt{initTriangles} & \texttt{TR[]} & \texttt{array of initial triangles} \\
\texttt{depthFirstTriangles} & \texttt{TR**} & \texttt{array of sublists} \\
\hline
\end{tabular}

Table 20: Additional fields of the \texttt{TRIANGULATION} data structure

### 3.4 Global operations on a triangulation

Procedures to create, delete, or select (via name) triangulations are available.

\texttt{TRIANGULATION *CrTri(char *name)}

creates a new \texttt{TRIANGULATION} data structure and initializes the fields. Additionally a list of triangulations is maintained.

\texttt{int SelTri(char *name)}

makes the triangulation \texttt{name} the current one, i.e. sets \texttt{actTriang}. If no triangulation is found \texttt{false} is returned. (Not yet implemented.)

\texttt{int CloseTri(TRIANGULATION **trgul)}

deletes the triangulation \texttt{trgul}, updates \texttt{actTriang} if necessary, and
sets *trigul to nil. All data for point, edges, triangles and their associated byte arrays are returned.

3.5 Using a triangulation

The recommended way of working with the elements of a triangulation is the following:

- Write a procedure to handle one object.
- Apply this procedure to all objects of a specified set.

This method should be clarified by a simple example (counting all edges of a triangulation):

```c
static int edgeCounter;
static int CountEdge(EDG *ed)
{
    edgeCounter++;
    return true;
}
...
...
edgeCounter = 0;
ApplyE(CountEdge, all);
    sprintf(globBuf,"'"%s' has %d edges\n", actTriang->name, edgeCounter);
ZIBStdOut(globBuf);
...
...
```

The `ApplyE` routine “knows” all about the internal linking of lists to find the edges of the current triangulation (defined through `actTriang`) and calls the procedure `CountEdge` with pointer to the corresponding edge data structure. The constant `all` defines some selection criteria, in this case all edges of the current triangulation.

Table 21 shows all apply procedures and Table 22 the selection codes.

3.6 Procedures (Refinement/Deletion)

The user can specify triangles to be refined. The “red” refinement method used in KASKADE was first introduced by R. Bank [3]. Triangles divide
ApplyP  apply on points
ApplyE  apply on edges
ApplyT  apply on triangles
ApplyPDepth  apply on points at triangles of depth $d$
ApplyEDepth  apply on edges of depth $d$
ApplyTDepth  apply on triangles of depth $d$

Table 21: Apply procedures

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>select all</td>
</tr>
<tr>
<td>allBackward</td>
<td>select all, backwards</td>
</tr>
<tr>
<td>allHist</td>
<td>select all, include fathers</td>
</tr>
<tr>
<td>allHistBackward</td>
<td>select all, backwards, include fathers</td>
</tr>
<tr>
<td>initial</td>
<td>select initial</td>
</tr>
<tr>
<td>dirichlet</td>
<td>select objects with Dirichlet condition</td>
</tr>
<tr>
<td>notDirichlet</td>
<td>select objects with no Dirichlet condition</td>
</tr>
<tr>
<td>initialNotDirichlet</td>
<td>select initial objects with no Dirichlet condition</td>
</tr>
<tr>
<td>boundInit</td>
<td>select all initial boundary</td>
</tr>
<tr>
<td>allReds</td>
<td>select all red refined triangles</td>
</tr>
</tbody>
</table>

Table 22: Selection codes

in four similar triangles, see Figure 3. This method preserves the numerical quality of the triangles, i.e. the new triangles have the same angles as the starting triangle. The depth field of the new triangles and edges gets the value of the father triangle increased by 1. The initial value is zero.

Figure 3: Red refinement

For anisotropic problems a different method [9] can be selected, see Figure 4.
We call this method “blue” in contrast to the “red” refinement of Bank. Note that a blue refinement is not always possible. An example is given in [12]. The new triangles have a different quality in respect to their angles which is hopefully the desired one. Two triangles generated by the triangulation of a rectangle (e.g. by the BOXES program) are tied together through their common edge, the “blue” diagonal. The field $\textit{isBlueDiagonal}$ denoting this inherits this attribute in case of (red and blue) refinement.

![Figure 4: Blue refinement](image)

These methods will generate incompatible triangulation which have to be “closed” in some way. Triangles with two refined edges are refined red, see Figure 5.

![Figure 5: Middle triangle will be red refined](image)

Triangles having an edge twice refined are refined red too, see Figure 6. This leaves some triangles with just one edge refined once. These triangles are refined “green”, Figure 7. Consecutive green closures will generate arbitrary bad angles. Therefore, at the start of a refinement step, green triangles are always removed.

```c
int OpenRef()
```
Figure 6: Right triangle will be refined red

Figure 7: Green closure

prepares a triangulation for refinement: all green triangles are removed. The refLevel counter is incremented.

```c
int RefTr(TR *t)
    marks a triangle for red refinement.

int RefBlue(TR *t, EDG *eInner, EDG *eRefine)
    marks two triangles for blue refinement.

int CloseRef(int verboseP)
    refines all marked triangles and generates the (green) closure.

int OpenDel()
    prepares a triangulation for deletion: all green triangles are removed.

int DelTr(TR *t)
    marks triangle t and its brothers for deletion.

int CloseDel(int verboseP)
    deletes all marked triangles and generates the (green) closure.
```
3.7 Customizing the module

The triangulation module generates a set of events (Table 23), which allow the user to insert procedures at these events using the application event module (see Section 2.5).

<table>
<thead>
<tr>
<th>Event name</th>
<th>associated data</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewPoint</td>
<td>(PT *p)</td>
<td>new point</td>
</tr>
<tr>
<td>ReturnPoint</td>
<td>(PT *p)</td>
<td>obsolete point</td>
</tr>
<tr>
<td>NewEdge</td>
<td>(EDG *ed)</td>
<td>new edge</td>
</tr>
<tr>
<td>ReturnEdge</td>
<td>(EDG *ed)</td>
<td>obsolete edge</td>
</tr>
<tr>
<td>NewTriangle</td>
<td>(TR *t)</td>
<td>new triangle</td>
</tr>
<tr>
<td>ReturnTriangle</td>
<td>(TR *t)</td>
<td>obsolete triangle</td>
</tr>
<tr>
<td>RefineEdge</td>
<td>(EDG *ed)</td>
<td>refined edge</td>
</tr>
<tr>
<td>RefineTriangle</td>
<td>(TR *t)</td>
<td>refined triangle</td>
</tr>
<tr>
<td>TriCreate</td>
<td>(TRIANGULATION *triang)</td>
<td>new triangulation</td>
</tr>
<tr>
<td>TriSelect</td>
<td>(TRIANGULATION *triang)</td>
<td>select triangulation</td>
</tr>
<tr>
<td>TriClose</td>
<td>(TRIANGULATION *triang)</td>
<td>delete triangulation</td>
</tr>
<tr>
<td>TriRefined</td>
<td>(TRIANGULATION *triang)</td>
<td>refined triangles</td>
</tr>
<tr>
<td>TriRenumbered</td>
<td>(TRIANGULATION *triang)</td>
<td>renumbered</td>
</tr>
<tr>
<td>TriDeleted</td>
<td>(TRIANGULATION *triang)</td>
<td>deleted triangles</td>
</tr>
<tr>
<td>TriRead</td>
<td>(TRIANGULATION *triang)</td>
<td>read triangulation</td>
</tr>
</tbody>
</table>

Table 23: Application events of the triangulation module

The TriRefined and TriDeleted are used by the graphic module to update automatically pictures of changed triangulations. TriRenumbered signals a renumbered triangulation, i.e. sparse matrices have to be restructured.

3.8 Command language interface

Some of the global actions on triangulations are available on the command language level.

```c
int TriInf(command*)
    prints some data of the current triangulation and lists the names of other triangulations in core.

int TriSel(command*)
    selects another triangulation which has to be in core.
```
int TriDel(command*)
    deletes the current triangulation.

int TriChk(command*)
    makes some consistency checks on the current triangulation and prints statistics about the refinement history.
4 Node module

In this section we describe the management of data for Finite Element computation on triangulations. The data is held at the basic objects (points, edges, triangles) via the associated arrays. The length of these arrays and the access to their components are controlled by the access management routines (see Section 2.6). The definition of the routines is held in the header file nodes.h.

The data of the current nodes structure is available via the global nodesState variable (see Table 24). The size of the array in the node data structure is defined by the constant MAX_NODE_GROUPS.

The module has to be initialized.

```c
int InitNodes()
{
    initializes the node module
}
```

4.1 Access management on basic objects

The length of the associated arrays (in this KASKADE version) can only be defined prior to the creation of the first triangulation. Resizing is not implemented.

The node module creates access lists (accPoint, accEdge, accTriangle) for the basic objects (points, edges, triangles) when the first triangulation is created. These identifiers should be used (by calls to the access right routines) to get the allowance to read and write certain parts of the associated arrays. The actual access is implemented with macros, see Table 25.

A typical part of a program using this feature is

```c
static int triArea=-1;
...
... /* Getting access right before first use */
... if (triArea<0)
    triArea = GetAccess(nodesState->accTriangle,
                        sizeof(REAL),sizeof(REAL),"area");
...
... /* Accessing data via RT macro */
... RT(t,triArea) = ComputeArea(t);
```
Table 24: Fields of the nodes data structure (all type int)

... /* Freeing access right after use */ ...
...
    ReturnAccess(nodesState->accTriangle,triArea);
    triArea = -1;
The following routine can be used to change the default storage sizes.

```c
int SetWS(int ptSize, int edgSize, int trSize, int ndSize)
```

initializes the sizes for the associated arrays. The usage of `ndSize` is explained in the next section.

```c
int GetWS(int *ptSize, int *edgSize, int *trSize, int *ndSize)
```

returns the sizes for the associated arrays. The usage of `ndSize` is explained in the next Section.

Routines to print a string `text` followed by the values stored in the associated arrays exist.

```c
void PrintPointValues(char *text, int maxVals)
```

prints data of the associated array for the first `maxVals` points.

```c
void PrintEdgeValues(char *text, int maxVals)
```

prints data of the associated array for the first `maxVals` edges.

```c
void PrintTriangleValues(char *text, int maxVals)
```

prints data of the associated array for the first `maxVals` triangles.

The actual usage of storage is displayed by the `infaccess` command. The default storage sizes can be changed with the `workspace` command.
4.2 Node management

Nodes are a further abstraction from our basic objects. The basic object is now “node storage” and not points, edges, or triangles. Nodes may exist on points, edges, or triangles. The access to nodes is organized in analogy to the handling of triangulations, i.e. it is possible to write routines which are applied by calls to ApplyNode to all nodes. These routines get a pointer to the node data as parameter. The node setup defines how many nodes are used at the basic objects. The routine getNodeAddresses is used to implement the matrix–vector multiplication with the stiffness matrix (or to assemble the sparse matrix). The macros to access fields from a node storage are collected in Table 26. Some fields are automatically set, see Table 27. These values are taken from the basic objects, respectively are computed to fit into the ApplyNode routine.

<table>
<thead>
<tr>
<th>access id</th>
<th>macro</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>iIndex</td>
<td>IV(vec,iIndex)</td>
<td>index of node vec</td>
</tr>
<tr>
<td>cClassA</td>
<td>CV(vec,cClassA)</td>
<td>index of node vec</td>
</tr>
<tr>
<td>cBoundP</td>
<td>CV(vec,cBoundP)</td>
<td>index of node vec</td>
</tr>
<tr>
<td>rX</td>
<td>RV(vec,rX)</td>
<td>x of node vec</td>
</tr>
<tr>
<td>rY</td>
<td>RV(vec,rY)</td>
<td>y of node vec</td>
</tr>
<tr>
<td>rSol</td>
<td>RV(vec,rSol)</td>
<td>solution at node vec</td>
</tr>
</tbody>
</table>

Table 26: Access macros for node storage

<table>
<thead>
<tr>
<th>access id</th>
<th>macro</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int NodeSetUp(int nodeWorkSpace, int nodesAtPoint, int nodesAtEdge, int nodesAtTriangle, int noOfEquations)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defines the internal data structure of working space at nodes. The size nodeWorkSpace is reserved for each equation through the access manage-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ment nodesAtPoint times at points, nodesAtEdge times at edges,</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NodesAtTriangle} times at triangles. Figure 8 shows the local storage setup at points and edges after a call of \texttt{NodeSetUp(60,1,1,0,2)}.

![Diagram of node storage at points and edges](image)

Figure 8: Node storage at points and edges

The offsets of the individual nodes can be computed from the fields \texttt{startPointNodes}, \texttt{startEdgeNodes}, and \texttt{startTriangleNodes}, of the \texttt{nodesState} variable. Thus the first index of node 4 in the preceding example will be \texttt{nodesState->startEdgeNodes[1]}. The corresponding equation numbers are stored in the \texttt{eqnPointNodes}, \texttt{eqnEdgeNodes}, and \texttt{eqnTriangleNodes} array. This means that in the array of equation numbers \texttt{nodesState->eqnEdgeNodes[1]} will have the value 1 (the second equation), at least in our example.

\texttt{void ReturnNodes()}

releases node setup, i.e. frees the node storage at the associated arrays for points, edges, and triangles.

\texttt{int ApplyNode(int (*UserProc)(char*), int sel)}

applies \texttt{UserProc} to all nodes, starting with the nodes at points, edges, and triangles. All nodes for one object (point, edge, or triangle) are always processed before advancing to the next object. That means that two nodes at a point have incremented indices. Nodes at different objects get indices far away. This routine is used to implement all the \texttt{NodeXxxx} routines. The parameter to \texttt{UserProc} is a pointer to the node storage. The selection code \texttt{sel} should be applicable to points, edges, and triangles.
int GetNodeAddresses(TR *t, char **nodes, int no)
returns the addresses of all nodes at the points, edges, and the triangle t. The length of the array of address nodes is no. This allows the association of nodes to the local stiffness matrix. The sequence is: all nodes for each equation. Thus we obtain a nice block structure of the local stiffness matrix.

int GetNoNodes()
returns the number of nodes in the current triangulation.

int AssRHS(int x, real *(*LocAssB)(TR*))
assembles the right-hand side to node field x. The routine LocAssB computes the local right-hand side.

int AssDiag(int x, real **(*LocAssA)(TR*))
assembles the diagonal of the stiffness matrix to vector x. The routine LocAssA computes the local stiffness matrix.

int NodeAssign(int a, int b)
assigns field b to a.

int NodeAssNeg(int a, int b)
assigns field -b to a.

int NodeLin(int a, int b, int c, real x)
assigns field b+x*c to a.

int NodeLinNeg(int a, int b, int c, real x)
assigns field -b+x*c to a.

int NodeAdd(int a, int b, int c)
assigns field b+c to y.

int NodeSub(int a, int b, int c)
assigns field b-c to a.

real NodeScalProd(int a, int b)
returns the scalar product of fields a and b.

int NodeSetZero(int a)
sets field a to 0.0.
int NodeSetBoundZero(int a)
    sets field a to 0.0 for all nodes on the boundary.

int NodeSetBound(int a)
    sets field a to the boundary values for all nodes on the Dirichlet boundary.

int PrintNodeValues(char *text, int max)
    prints the node values for the first max nodes. The output starts with text.

4.3 Sparse matrices

Sparse matrices are stored as lists of real numbers which correspond to a list of neighbor nodes. This list is generated when the first matrix is created. A sparse matrix is identified via an integer variable and a set of macros returning the length of these lists and the starting addresses of the corresponding arrays (see Table 28).

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_OF_NODE_NEIGHBORS(vec)</td>
<td>retrieve the number of neighbors from vec</td>
</tr>
<tr>
<td>NODE_NEIGHBOR_VEC(vec)</td>
<td>retrieve the array of neighbor nodes from vec</td>
</tr>
<tr>
<td>NODE_SPARSE_VEC(vec, index)</td>
<td>retrieve from vec the line of sparse matrix index</td>
</tr>
</tbody>
</table>

Table 28: Macros to access sparse matrices

The algorithm to generate the sparse matrix now implemented is straightforward:

- at each node, set the counter for the number of neighbors noOfNodeNeigh to zero;
- scan all triangles and increase noOfNodeNeigh according to the node-setup;
- scan all edges and subtract the double entries;
- allocate the array of neighbor nodes adrOfNodeNeigh;
- scan all edges and store all neighbor nodes in adrOfNodeNeigh, sort them with respect to the node attribute iIndex.
int MakeSparseMatrix(int pos, char *name)
generates the frame of a sparse matrix. If not yet available the node neighborhood relationship is constructed and arrays to store the matrix lines are allocated. An integer identifying the matrix is returned, -1 denotes a failure.

int AssembleSparse(int index, char *name)
allocates storage for an additional sparse matrix and assembles without respecting Dirichlet boundary conditions.

int BoundSparse(int index)
forces Dirichlet boundary conditions on matrix index.

void ReturnSparseMatrix(int index)
returns the storage for the matrix index including the frame.

Set0MatSparse(int index, int n, int symP)
sets all matrix elements of index to 0.0. The parameters n and symP are ignored, they are just included for compatibility with the full and envelope matrix routines.

void PrintMatSparse(char *s, int index, int n, int symP)
prints the string s and all non-zero elements of the sparse matrix index. The parameters n and symP are ignored, they are just included for compatibility with the full and envelope matrix routines.

4.4 The matrix/vector multiplication method

The matMulMethod data structure contains a set of routines with some data, see Table 29.
The routines of the method have the following duties.

int InitMatMul(int stiff, real** (*NumAss)(TR*), int rhs, real* (*NumAssB)(TR*))
initializes the matrix/vector multiplication method with routines NumAss and NumAssB doing local assembling of the stiffness matrix and the right-hand side of the linear system. stiff and rhs identifies a path to the stiffness matrix, respectively the right-hand side vector which is computed if rhs! =-1.
Table 29: The `matMulMethod` data structure

<table>
<thead>
<tr>
<th>name</th>
<th>char*</th>
<th>name of the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>int</td>
<td>identifier right-hand side</td>
</tr>
<tr>
<td>stiff</td>
<td>int</td>
<td>identifier sparse matrix</td>
</tr>
<tr>
<td>InitMatMul</td>
<td>int(<em>)(int,real**(</em>)(TR*), int,real**(<em>)(TR</em>))</td>
<td>initialize method</td>
</tr>
<tr>
<td>MatMul</td>
<td>int(*)(int,int)</td>
<td>multiply stiffness matrix</td>
</tr>
<tr>
<td>CloseMatMul</td>
<td>int(*)()</td>
<td>close method</td>
</tr>
</tbody>
</table>

int MatMul(int x, int y)

computes the product of stiffness matrix and the node field `x` and stores the result in `y`.

int CloseMatMul()

may return storage which is not used anymore.

A new method can be defined.

int DefMatMulMethod(char *name,
  int (*InitMatMul)(int,real**(*)(TR*),int,real**(*)(TR*)),
  int (*MatMul)(int,int),int (*CloseMatMul)())

defines a new matrix/vector multiplication method with name `name`.

The local matrix/vector multiplication implements the product by multiplying with local stiffness matrices and adding their contributions. No global stiffness matrix is used.

int InitMatMul(int index, real **(*LocAssA)(TR*), int rhs,
  real **(*LocAssB)(TR*))

preserves the computation of the multiplication of a vector with the stiffness matrix and assembles the right-hand side vector `rhs`. `index` is a dummy parameter. The routines `LocAssA` and `LocAssB` compute the local stiffness matrix and the right-hand side corresponding to a triangle. A negative value of `rhs` suppresses the computation of the right-hand side. `true` is always returned.

int AXMul(int x, int y)
computes the product of node field $x$ with the stiffness matrix and stores the result in $y$. Dirichlet conditions are ignored if $\text{compNormP}$ in $\text{nodesState}$ is set.

**The sparse matrix/vector multiplication** assembles the complete stiffness matrix at the call of $\text{InitSparseMul}$ and uses this matrix on the following calls of $\text{AXSparseMul}$. The sparse matrix is removed on events $\text{TriRefined}$, $\text{TriRenumbered}$, and $\text{TriRefined}$.

```c
int InitSparseMul(int index, real **(*LocAssA)(TR*), int rhs, 
real *(*LocAssB)(TR*))
```

assembles the stiffness matrix $\text{index}$ and the right-hand side $\text{rhs}$. The routines $\text{LocAssA}$ and $\text{LocAssB}$ compute the local stiffness matrix and the right-hand side corresponding to a triangle. The Dirichlet boundary conditions are fully integrated in the complete matrix. Negative values of $\text{index}$ or $\text{rhs}$ suppress the computation of the stiffness matrix or the right-hand side. The routine returns the index of the stiffness matrix. A negative value denotes a failure to collect the stiffness matrix.

```c
int AXSparseMul(int x, int y)
```

computes the product of node field $x$ with the stiffness matrix and stores the result in $y$.

### 4.5 Command language interface

The $\text{infaccess}$ can be used to get information on the node structure. The structure of the sparse matrices can be displayed by functions of the graphic module.

```c
int WorkSpace(command*)
```

changes the amount of memory allocated at points, edges, triangles, and nodes.
5 Adaptive computations

This chapter will contain information on the module which handles the adaptive computation of geometric (and other values) in KASKADE. The user can add his own code to the predefined procedures.

The design of the interface is still in discussion. The first version is documented in the source compadapt.tex.
6 Assembling module

6.1 Defining the problem

Let $D^k_i$ be defined

$$D^k_i u^i := - \text{div}(P^k_i \text{grad } u^i) + \beta^k_1 u^i_x + \beta^k_2 u^i_y + q^k u^i$$

with $P^k = \begin{pmatrix} p^k_{11} & p^k_{12} \\ p^k_{21} & p^k_{22} \end{pmatrix}$

then KASKADE variants handle (systems of $n$) partial differential equations of the type

$$\sum_{i=0}^{n-1} D^k_i u^i = g^k \quad \text{in } \Omega \quad i = 0, \ldots, n - 1$$

$$(P^k_i \text{grad } u^i) \cdot \mathbf{n} + \eta^i u^i = \xi^i \quad \text{on } \Gamma_0 \subset \partial \Omega$$

$$(P^k_i \text{grad } u^i) \cdot \mathbf{n} = 0 \quad \text{on } \partial \Omega \setminus (\Gamma_0 \cup \Gamma_1)$$

$p^k_{ij}$, $q^k$, $\beta^k_1$, $\beta^k_2$, $g^k$, $\gamma^i$, $\eta^i$ and $\xi^i$ are piecewise continuous real valued functions on $\Omega$, respectively the corresponding boundaries. $\Omega \subset \mathbb{R}^2$ is a bounded polygonal domain. Each KASKADE variant has its own assumptions on these functions. Here we describe the method to formulate the problem. Some information on the class of problems are collected in the problemType data structure (see Table 30).

| char* name | name of the problem class |
| int maxNoOfProblems | max number of problems |
| int symP | problems have to be symmetric? |
| int noOfEquations | number of equations |
| int nodeEntries | length of node memory |

Table 30: Fields of the problemType data structure

```c
int InitProblem(char *name, int maxNoOfProblems, int noOfEquations, int nodeEntries, int symP)
```

defines a problemType data structure which can be accessed by the global variable theProblem. This routine should be called only once.
A set of user functions defines a concrete problem. Such a problem is defined by a call to `SetProblem` which defines a problem data structure (see Table 31). An actual set for computation is selected by the command language function `problem`.

```c
int SetProblem(char *name, char **varNames,
               int (*Laplace)(real,real,int,real*,int,int),
               int (*Convection)(real,real,int,real*,int,int),
               int (*Helmholtz)(real,real,int,real*,int,int),
               int (*Source)(real,real,int,real*,int),
               int (*Obstacle)(real,real,int,real*,int),
               int (*Cauchy)(real,real,int,real*,int),
               int (*Dirichlet)(real,real,int,real*,int),
               int (*Sol)(real,real,int,real*,int),
               real** (*LocAss)(TR*),real* (*LocAssB)(TR*))
```

defines a problem data structure which can be accessed by the global variable `actProblem`. The parameters are routine addresses which are specified in the following. The new problem is added to the array of available problems (`problems`). A special problem can be selected by the `problem` command.

```c
void SetProblemAddresses()
```

initializes the command language interface, i.e. the `problem` and `infp` command.

Table 31: Fields of the problem data structure
void SetStdProblem()
    predefines some example problems.

After initializing the problem class with InitProblems, the user has to define
his version of the following functions to define the differential equations.

int Laplace(real x, real y, int classA, real fVals[4],
    int equation, int variable)
    returns the coefficients $p_{ik}^{i}(x,y)$ in the array elements of fVals, i.e.
    $p_{11}^{i}(x,y)$ in fVals[0] etc. equation(index $k$) and variables(index $i$)
    are valid for systems of equations. classA gives additional information
    of the geometric sub domain which includes $(x,y)$. That should be
    useful to implement material constants.

int Convection(real x, real y, int classA, real fVals[2],
    int equation, int variable)
    returns the coefficients $\beta_{1k}^{i}(x,y)$ and $\beta_{2k}^{i}(x,y)$ in the array elements
    fVals[0] and fVals[1].

int Helmholtz(real x, real y, int classA, real *fVal,
    int equation, int variable)
    returns the coefficient $q_{ik}^{i}(x,y)$ in fVal.

int Source(real x, real y, int classA, real *fVal,
    int equation)
    returns the source value $g_{i}(x,y)$ in fVal.

int Obstacle(real x, real y, int classA, real fVals[2],
    int equation, int variable)
    returns the upper and lower obstacle in the array elements fVals[0]
    and fVals[1].

int Cauchy(real x, real y, int classA, real fVals[2],
    int variable)
    returns $\eta_{i}(x,y)$ and $\xi_{i}(x,y)$in the array elements fVals[0] and fVals[1].

int Dirichlet(real x, real y, int classA, real *fVal,
    int variable)
    returns the boundary values $\gamma_{i}(x,y)$ in fVal.
int Sol(real x, real y, int classA, real fVals[3],
        int variable)

returns the solution \( u(x, y) \) and its derivatives \( u_x(x, y) \) and \( u_y(x, y) \) in
the array elements \( fVals[0] \), \( fVals[1] \), and \( fVals[2] \). This feature
is used to test the computed solution against the real solution if it is
known.

Each of these user routines may be missing (nil). The return code of these
routines have to be one of the constants collected in Table 32.

| F_CONSTANT | values are constant on a triangle or an edge |
| F_VARIABLE | values are not constant |
| F_IGNORE   | no values supplied |
| F_FAILED   | User failed to supply values |

Table 32: Return codes for the problem defining user routines

The user can define his problem by supplying his own local discretization
with the following routines.

real** LocAss(TR *t)
    returns a pointer to the local stiffness matrix.

real* LocAssB(TR *t)
    returns a pointer to the local right-hand side.

The problem and problemTypedef data structures for the currently selected
problem are accessed via the actProblem and theProblem pointers. A
change of the current problem raises an event (ProblemChanged).
Two commands are defined to select (and inform) on the current problem,
InfPrb and Problem.

6.2 Shape functions

Some subroutines to supply values of the shape functions and their derivatives
at points in the standard triangle are predefined.

void StdShape(real x,real y,int no,real *f,
defines the values of the standard shape functions at \((x, y)\)-coordinates. The last three entries define the “hierarchical” extension of linear elements by quadratic “bump” functions.

\[
\begin{align*}
1 - x - y \\
x \\
y \\
4xy \\
4y(1 - x - y) \\
4x(1 - x - y)
\end{align*}
\]

void StdQShape(real x, real y, int no, real *f, real *fx, real *fy, real *fxx, real *fxy, real *fyy)

defines the values of the standard quadratic shape functions at \((x, y)\)-coordinates.

\[
\begin{align*}
2(1 - x - y)(1/2 - x - y) \\
x(2x - 1) \\
y(2y - 1) \\
4xy \\
4y(1 - x - y) \\
4x(1 - x - y)
\end{align*}
\]

void StdCShape(real x, real y, int no, real *f, real *fx, real *fy, real *fxx, real *fxy, real *fyy)

defines the values of the standard cubic shape functions at \((x, y)\)-coordinates.

\[
\begin{align*}
(1 - x - y)(3(1 - x - y) - 1)(3(1 - x - y) - 2)/2 \\
x(3x - 1)(3x - 2)/2 \\
y(3y - 1)(3y - 2)/2 \\
9xy(3x - 1)/2 \\
9xy(3y - 1)/2 \\
9y(1 - x - y)(3y - 1)/2 \\
9(1 - x - y)y(3(1 - x - y) - 1)/2 \\
9(1 - x - y)x(3(1 - x - y) - 1)/2 \\
9x(1 - x - y)(3x - 1)/2 \\
27xy(1 - x - y)
\end{align*}
\]
6.3 Local assembling

Two data structures \texttt{integData} and \texttt{localData} are used to ease local assembling over triangles. The first one (see Table 33) holds global information like precomputed function values of shape functions and is created by a call of the \texttt{InitAss} routine. The second one (see Table 34) holds the transformation data of a triangle to the reference triangle \(\{0,0\}, (0,1), (1,0)\).

<table>
<thead>
<tr>
<th>integPointX</th>
<th>real*</th>
<th>(x_k) values of integration points</th>
</tr>
</thead>
<tbody>
<tr>
<td>integPointY</td>
<td>real*</td>
<td>(y_k) values of integration points</td>
</tr>
<tr>
<td>integWeight</td>
<td>real*</td>
<td>weights at the integration points</td>
</tr>
<tr>
<td>lineIPX</td>
<td>real**</td>
<td>(x_{ik}) for edge (i)</td>
</tr>
<tr>
<td>lineIY</td>
<td>real**</td>
<td>(y_{ik}) for edge (i)</td>
</tr>
<tr>
<td>lineIW</td>
<td>real**</td>
<td>weights for line integration</td>
</tr>
<tr>
<td>shape</td>
<td>real**</td>
<td>values of shape functions (\phi^i)</td>
</tr>
<tr>
<td>shapeX</td>
<td>real**</td>
<td>values of shape functions (\phi^i_x)</td>
</tr>
<tr>
<td>shapeY</td>
<td>real**</td>
<td>values of shape functions (\phi^i_y)</td>
</tr>
<tr>
<td>lineVals</td>
<td>real***</td>
<td>values (v_{ikm}) of (\phi^i_y) on edge (k)</td>
</tr>
<tr>
<td>noOfIPoints</td>
<td>int</td>
<td>number of integration points</td>
</tr>
<tr>
<td>noOfLineIP</td>
<td>int</td>
<td>number of integration points</td>
</tr>
<tr>
<td>noOfShapeFunc</td>
<td>int</td>
<td>number of shape functions</td>
</tr>
<tr>
<td>symP</td>
<td>int</td>
<td>true if symmetric</td>
</tr>
</tbody>
</table>

Table 33: The \texttt{integData} data structure

Pointers to the current data structures are held in the external variables \texttt{actIntegData} and \texttt{actLocalData}. The following routine precomputes the fields of an \texttt{integData} data structure.

\[
\text{integData } \ast \text{NewIData}(\text{int } iFormula, \text{ int } SF, \\
\text{ void } (*\text{ShapeF})(\text{real,real,int,real*,real*,real*,
real*,real*),
real*,real*),\text{int } \text{symP})}
\]

precomputes values of the shape functions and their derivatives at the integration points. The shape functions are defined by the routine \texttt{ShapeF}, examples are \texttt{StdShape} and \texttt{StdQShape}. The parameter \texttt{iFormula} selects the set of integration points, see Table 35. The user can supply his own set of integration points by changing the values of the variables \texttt{userNoIP}, \texttt{userIPX}, \texttt{userIPY}, and \texttt{userIW}. 

44
Table 34: The localData data structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
<td>real*</td>
<td>(x_k, y_k) values in triangle</td>
</tr>
<tr>
<td>(p_{11}, p_{12}, p_{21}, p_{22})</td>
<td>real*</td>
<td>(p_{jl}(x_k, y_k)) values</td>
</tr>
<tr>
<td>(q)</td>
<td>real*</td>
<td>(q(x_k, y_k)) values</td>
</tr>
<tr>
<td>(g)</td>
<td>real*</td>
<td>(g(x_k, y_k)) values</td>
</tr>
<tr>
<td>(\beta_{1}, \beta_{2})</td>
<td>real*</td>
<td>(\beta_j(x_k, y_k)) values</td>
</tr>
<tr>
<td>(f_{11}, f_{12}, f_{21}, f_{22})</td>
<td>real</td>
<td>transformation data</td>
</tr>
<tr>
<td>(\text{area})</td>
<td>real</td>
<td>area of the triangle</td>
</tr>
<tr>
<td>(\text{classA})</td>
<td>int</td>
<td>class identifier of the triangle</td>
</tr>
<tr>
<td>(\text{equation})</td>
<td>int</td>
<td>number of equation</td>
</tr>
<tr>
<td>(\text{variable})</td>
<td>int</td>
<td>number of variable</td>
</tr>
<tr>
<td>(t)</td>
<td>TR*</td>
<td>pointer to the triangle</td>
</tr>
</tbody>
</table>

int InitAss()
precomputes the values of the data structures for the integration routines, i.e. calls NewIData for a standard set of integData and localData data structures.

<table>
<thead>
<tr>
<th>Name</th>
<th>#IPs</th>
<th>(x) of IPs</th>
<th>(y) of IPs</th>
<th>weights of IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANKIP</td>
<td>3</td>
<td>1/6, 2/3, 1/6</td>
<td>1/6, 1/6, 2/3</td>
<td>1/6, 1/6, 1/6</td>
</tr>
<tr>
<td>LINIP</td>
<td>1</td>
<td>1/3</td>
<td>1/3</td>
<td>1/2</td>
</tr>
<tr>
<td>QUADIP</td>
<td>3</td>
<td>1/2, 1/2, 0</td>
<td>0, 1/2, 1/2</td>
<td>1/6, 1/6, 1/6</td>
</tr>
<tr>
<td>USERIP</td>
<td></td>
<td>userNoIP</td>
<td>userIPX</td>
<td>userIPY</td>
</tr>
</tbody>
</table>

Table 35: Sets of integration points (IPs)

The following routines can be used to construct a local stiffness matrix for a triangle.

int OpenAss(TR *t)
preamptly computes the data for the transformation from the triangle \(t\) to the reference triangle and prepares computations of the local integration of the stiffness matrix, mass matrix, right-hand side and more. OpenAss initializes all data used by GetDomainVal((real x, real y, int rIndex, real *fvals) routine, which interpolates variable rIndex at \((x, y)\) in the inner of the triangle \(t\) for all equations.

45
int CloseAss()
can be used to save the result of the local stiffness matrix.

int CompEllOp(real **locA)
computes the local stiffness matrix. The result is added to the array locA[]. The function actProblem->Laplace is used to compute

\[ A_{ik} = \int_T \left( \sum_{j,l=1}^n p_{jl} \frac{\partial}{\partial x_j} \phi_i \frac{\partial}{\partial x_l} \phi_k \right) d(x, y) \]

with the shape functions \( \phi_k \) and \( x_1 = x, x_2 = y \). The Cauchy boundary conditions are applied if necessary:

\[ A_{ik} + = \int_{\partial T \cap \Gamma_1} \eta \phi_i \phi_k ds . \]

The GetNodeAddresses routine returns the dimension \( (n) \) of the local stiffness matrix.

int CompMass(real **locA)
computes the local mass matrix. The result is added to the array locA[]. The function actProblem->Helmholtz is used to compute

\[ A_{ik} = \int_T q \phi_i \phi_k d(x, y) . \]

int CompConvTerm(real **locA)
computes the local convection matrix. The result is added to the array locA[]. The function actProblem->Convection is used to compute

\[ A_{ik} = \int_T \left( \beta_1 \phi_k \frac{\partial}{\partial x} \phi_i + \beta_2 \phi_k \frac{\partial}{\partial y} \phi_i \right) d(x, y) . \]

(Not yet implemented.)

int CompRs(real *locB)
computes the local right-hand side. The result is added to the array locB[]. The function actProblem->Source is used to compute

\[ B_k = \int_T q \phi_k d(x, y) . \]

The Cauchy boundary conditions are applied by using actProblem->Cauchy routine.

\[ B_k + = \int_{\partial T \cap \Gamma_1} \xi \phi_k ds . \]
These routines can be used as prototypes for different discretizations. Standard NumAss and NumAssR functions are included which call these components in a loop over the equations and variables, and which return (in the case of linear shape functions) a \((3n) \times (3n)\) matrix \((n\) the number of equations).

\[
\text{real } **\text{NumAss}(TR *t)
\]

assembles the local matrix of triangle \(t\) for a linear system of equations.

\[
\text{real } *\text{NumAssR}(TR *t)
\]

assembles the local right-hand side of triangle \(t\) for a linear system of equations.

Sometimes a user function needs access to interpolated values of other variables on the current triangle. In this case the function GetDomainVal should be used.

\[
\text{int GetDomainVal(real, real y, int rIndex, real *fvals)}
\]

interpolates the value \(rIndex\) on the current triangle (actLocalData->t) for point \(x, y\). The result is stored in \(fvals[k]\), \(k\) varying over the number of equations.
7 Solve module

Global information (i.e. accumulated timing information, selection solution components for direct and iterate solvers, error estimators and refinement strategies) is collected in the `solve` data structure, see Table 36. The actual status can be accessed via the `solveState` variable of type `solve*`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char*</td>
<td>name of the solver</td>
</tr>
<tr>
<td>verboseP</td>
<td>int</td>
<td>level of verbosity</td>
</tr>
<tr>
<td>totalTime</td>
<td>real</td>
<td>time (seconds)</td>
</tr>
<tr>
<td>assTime</td>
<td>real</td>
<td>time for assembly</td>
</tr>
<tr>
<td>dirTime</td>
<td>real</td>
<td>time for direct solving</td>
</tr>
<tr>
<td>iteTime</td>
<td>real</td>
<td>time for iterative solving</td>
</tr>
<tr>
<td>estiTime</td>
<td>real</td>
<td>time for error estimation</td>
</tr>
<tr>
<td>refTime</td>
<td>real</td>
<td>time for refining</td>
</tr>
<tr>
<td>drawTime</td>
<td>real</td>
<td>time for drawing</td>
</tr>
<tr>
<td>renumbTime</td>
<td>real</td>
<td>time for renumbering</td>
</tr>
<tr>
<td>eNorm</td>
<td>real</td>
<td>energy norm</td>
</tr>
<tr>
<td>Direct</td>
<td>int(*)</td>
<td>solves direct</td>
</tr>
<tr>
<td>Iterate</td>
<td>int(*)</td>
<td>solves iterative</td>
</tr>
<tr>
<td>Estimate</td>
<td>int(*)</td>
<td>estimates error</td>
</tr>
<tr>
<td>Refine</td>
<td>int(*)</td>
<td>refines</td>
</tr>
<tr>
<td>Solve</td>
<td>int(*)</td>
<td>adaptive solve</td>
</tr>
<tr>
<td>ansatz</td>
<td>femAnsatz</td>
<td>see Table 37</td>
</tr>
<tr>
<td>director</td>
<td>dirMethod*</td>
<td>direct data structure</td>
</tr>
<tr>
<td>itor</td>
<td>dirMethod*</td>
<td>iterative data structure</td>
</tr>
<tr>
<td>estor</td>
<td>dirMethod*</td>
<td>error estimate data structure</td>
</tr>
<tr>
<td>reftor</td>
<td>refMethod*</td>
<td>refine data structure</td>
</tr>
</tbody>
</table>

Table 36: Some fields of the `solve` data structure

<table>
<thead>
<tr>
<th>femAnsatz</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>linear ansatz</td>
</tr>
<tr>
<td>quadratic</td>
<td>quadratic ansatz</td>
</tr>
<tr>
<td>cubic</td>
<td>cubic ansatz</td>
</tr>
</tbody>
</table>

Table 37: The `femAnsatz` data type
int SolSetUp(int bytesOnNode, femAnsatz ansatz)
prepares data structures to solve a problem. The node module is initialized with nodes at points, edges and triangles corresponding to ansatz (see Table 37). The amount of bytes is given by bytesOnNode. solveState is allocated and the fields of this solve data structure are set. The event types of the solve module are defined, see Table 38.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewSol</td>
<td>solution computed</td>
</tr>
<tr>
<td>NewErrEsti</td>
<td>error estimated</td>
</tr>
<tr>
<td>ProblemChanged</td>
<td>the problem has changed</td>
</tr>
</tbody>
</table>

Table 38: Events of the solve module

7.1 Interface for direct solvers

Direct solvers are called by the Direct driver routine. This routine

- checks the applicability of the selected direct solver (i.e. only positive definite systems),
- renumbers the triangulation (calling Renumber),
- assembles the linear system with the appropriate matrix structures (calling Assemble),
- decomposes the system (calling Decomp),
- extracts the solution (calling FBSubst), and
- generates a NewSol event.

Additional timing information is assembled and a progress report is printed on request. The data structure controlling this process is dirMethod, see Table 39. The actual direct method is accessed via actDirector.

7.1.1 Direct methods and their duties

int Decomp()
name char* name of the direct solver
renumberName char* name of the renumbering method
verboseP int level of verbosity
onlyPosDef int only for positive definite systems
onlySym int only for symmetric systems
dim int dimension of the solution
env int* vector of envelope offsets
stiffSparse int id of sparse matrix
decompSparse int id of sparse matrix
stiff real** stiffness matrix
rhs real* right-hand side
decomp real** decomposed matrix
diagonal real* diagonal elements
solution real* solution
nodes char** array of node addresses
method storeMethod full, envelope, or sparse
(see Table 40)
Decomp int(*)() decomposition method
FBSubst int(*)() for/backward substitution method
CheckSol int(*)() check solution method
Assemble int(*)() assemble matrix method
Renumber int(*)(printMode) renumber triangulation
PrintLS int(*)(printMode) print matrix method
(see Table 41)
ReleaseAll int(*)() release matrix method

Table 39: The dirMethod data structure

factorizes the stiffness matrix

\[ A = LDL^T. \]

The matrices and vectors are identified depending on the storeMethod selected.

int FBSubst()
    computes the forward substitution

\[ Ly = b \]

50
and the backward substitution
\[ DL^T x = y \]
and stores the result in `solveState->rSol`.

`int CheckSol()` checks the solution. This method is gracefully skipped, if not available.

`int Assemble()` assembles the stiffness matrix and right-hand side according to `storeMethod` (see Table 40).

```
<table>
<thead>
<tr>
<th>full</th>
<th>full matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>envelope</td>
<td>envelope matrix</td>
</tr>
<tr>
<td>sparse</td>
<td>sparse matrix</td>
</tr>
</tbody>
</table>
```

Table 40: The `storeMethod` data type

`int PrintLS(printMode mode)` prints the matrices and vector corresponding to `mode`. The `printMode` data type defines the print selection (see Table 41).

```
| printNothing  | print nothing |
| printLinSys   | print linear system $Ax = b$ |
| printDecomp   | print decomposed matrix $LDL^T$ |
| printSol      | print solution $x$ |
```

Table 41: The `printMode` data type

`int ReleaseAll()` releases all data for matrices and vectors.

### 7.1.2 Defining a direct method

`int DefDirMethod(char *name, char *renumberName, storeMethod method, int onlyPosDef, int onlySym, int (*Decomp)(), int (*FBSubst)(), int (*CheckSol)(),

51
defines a new direct method, i.e. generates a new \texttt{dirMethod} data structure which can be accessed via \texttt{actDirector} or \texttt{solveState-&gt;director}. All defined direct methods are collected in the array \texttt{dirMethods[]} of \texttt{dirMethod*}. The current direct method may be called by \texttt{Direct()}.  

7.1.3 Command language interface

\texttt{direct} is the command language interface. The actual direct solver can be changed with the \texttt{seldirect} command.

7.2 Interface for iterative solvers

Iterative solvers are called by the \texttt{Iterate} driver routine. This routine

- assembles the right-hand side of the linear system (calling directly \texttt{AssRHS}),
- initializes the iterative solver (calling \texttt{InitMethod}),
- calls the iterative solver (calling \texttt{Method}),
- closes the iterative solver (calling \texttt{CloseMethod}), and
- generates a \texttt{NewSol} event.

The \texttt{InitMethod} is responsible for initializing a suitable matrix/vector multiplication. If needed preconditioners have to be initialized too. Additional timing information is assembled and a progress report is printed on request. The data structure controlling this process is \texttt{iteMethod}, see Table 42. The actual iterative method is accessed via \texttt{actIter}.

7.2.1 Iterative methods and their duties

\texttt{int InitMethod()}

initializes the iterative process, i.e. checks for correct application, initializes matrix/vector multiplication and preconditioner methods, performs onestep, etc.
name char* name of the iterative solver
verboseP int level of verbosity
needsMatrix int needs the sparse stiffness matrix
computesRes int computes the residue
onlyPosDef int only for positive definite systems
onlySym int useful only for symmetric systems
iteCount int number of iterations
minSteps int minimum of iteration steps
maxSteps int maximum of iteration steps
eps real requested iteration error
res real iteration error
iteFactor real safety factor
InitMethod int(*)(iteMethod*) initialization method
Method int(*)(iteMethod*) iteration method
CloseMethod int(*)(iteMethod*) closing method
matMul matMulMethod* selected matMul method
preCond preCondMethod* selected preCond method

Table 42: The iteMethod data structure

int Method()
    performs the required iterative step and updates the data in solveState and actItor.

int CloseMethod()
    closes the matrix/vector multiplication and preconditioner methods.

7.2.2 Defining an iterative method

int DefIteMethod(char *name, int needsMatrix, int computesRes,
    int onlyPosDef, int onlySym, int (*InitMethod)(iteMethod*),
    int (*Method)(iteMethod*), int (*CloseMethod)(iteMethod*))

defines a new iterative method, i.e. generates a new iteMethod data structure which can be accessed via actItor or solveState->itor. All defined iterative methods are collected in the array iteMethods[] of iteMethod*. The current iterative method may be called by Iterate().
7.2.3 The preconditioner method

The *preConMethod* data structure contains a set of routines with some data, see Table 43.

<table>
<thead>
<tr>
<th>name</th>
<th>char*</th>
<th>name of the preconditioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitPreCond</td>
<td>int(*)()</td>
<td>initialize</td>
</tr>
<tr>
<td>PreCond</td>
<td>int(*)(int,int)</td>
<td>preconditioning</td>
</tr>
<tr>
<td>ClosePreCond</td>
<td>(*)()</td>
<td>close</td>
</tr>
</tbody>
</table>

Table 43: The *preConMethod* data structure

The routines of the method have the following duties.

```c
int InitPreCond()
    initializes the preconditioner.

int PreCond(int x, int y)
    transforms x to y (preconditioning).

int ClosePreCond()
    cleans everything up.
```

A new method can be defined.

```c
int DefPreConMethod(char *name, int (*InitPreCond)(),
                    int (*PreCond)(int, int), int (*ClosePreCond)())
    defines a new preconditioning method with name *name*.
```

7.2.4 Command language interface

*iterate* is the command language interface. The actual iterative solver can be changed with the *seliterate* command. The *selprecond* commands is available too.

7.3 Interface for error estimators

Error estimators are called by the *Estimate* driver routine. This routine

- initializes the error estimator (calling *InitMethod*),
• calls the error estimator (calling Method), and
• closes the error estimator (calling CloseMethod).

Additional timing information is assembled and a progress report is printed on request. The data structure controlling this process is estiMethod see Table 44.

<table>
<thead>
<tr>
<th>name</th>
<th>char*</th>
<th>name of the error estimator</th>
</tr>
</thead>
<tbody>
<tr>
<td>verboseP</td>
<td>int</td>
<td>level of verbosity</td>
</tr>
<tr>
<td>ed_rq</td>
<td>int</td>
<td>storage for $A_{QL} \bar{U}_L$</td>
</tr>
<tr>
<td>ed_diag</td>
<td>int</td>
<td>storage for $d_Q$</td>
</tr>
<tr>
<td>ed.rhs</td>
<td>int</td>
<td>storage for $b_Q$</td>
</tr>
<tr>
<td>ed_res</td>
<td>int</td>
<td>storage for $r_Q$</td>
</tr>
<tr>
<td>globError</td>
<td>real</td>
<td>estimated global error</td>
</tr>
<tr>
<td>mBar</td>
<td>real</td>
<td>weighted residual $\bar{m}$</td>
</tr>
<tr>
<td>sFactor</td>
<td>real</td>
<td>need $s(2.0)$ times new points</td>
</tr>
<tr>
<td>sigma</td>
<td>real</td>
<td>safety factor $\sigma(0.95)$</td>
</tr>
<tr>
<td>InitMethod</td>
<td>int(<em>)(iteMethod</em>)</td>
<td>initialization method</td>
</tr>
<tr>
<td>Method</td>
<td>int(<em>)(iteMethod</em>)</td>
<td>estimation method</td>
</tr>
<tr>
<td>CloseMethod</td>
<td>int(<em>)(iteMethod</em>)</td>
<td>closing method</td>
</tr>
</tbody>
</table>

Table 44: The estiMethod data structure

estimate is the command language interface. The actual estimator can be changed with the seleestimate command.

### 7.4 Interface for refinement strategies

Refinement strategies are called by the Refine driver routine. This routine

•-initializes the refinement strategy (calling InitMethod),
• calls the refinement strategy (calling Method), and
• closes the refinement strategy (calling CloseMethod).

Additional timing information is assembled and a progress report is printed on request. The data structure controlling this process is refMethod, see Table 45.
The available refinement strategies are based on [6] (mean value) and extrapolation.

refine is the command language interface. The actual refinement strategy can be changed with the selrefine command.

### 7.5 Interface for a completely adaptive solution

The full-adaptive solution is computed the solveState->Solve routine. Figure 9 shows a flow diagram of steps taken by the Solve procedure.

#### 7.5.1 Break conditions

The solve process calls the routine solveState->BreakCond to determine a stopping condition. The standard BreakCond routine checks the number of solve steps, number of unknowns, the estimated global errors, the failure of the substeps (Direct, Estimate, Refine and Iterate). The reason for a break is held in the breakReason of the solveState variable. A message explaining the reason can be printed with the solveState->PrintBreakCond routine.

#### 7.5.2 Command language interface

The commands solve and setbreak constitute the command language interface.
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>breakReason</td>
<td>int</td>
<td>break reason, see Table 47</td>
</tr>
<tr>
<td>maxSteps</td>
<td>int</td>
<td>maximal number of steps</td>
</tr>
<tr>
<td>curStep</td>
<td>int</td>
<td>current step</td>
</tr>
<tr>
<td>maxDepth</td>
<td>int</td>
<td>maximal depth of triangulation</td>
</tr>
<tr>
<td>maxPoints</td>
<td>int</td>
<td>maximal number of points</td>
</tr>
<tr>
<td>dirFail</td>
<td>int</td>
<td>direct solver failed</td>
</tr>
<tr>
<td>estiFail</td>
<td>int</td>
<td>estimator failed</td>
</tr>
<tr>
<td>refFail</td>
<td>int</td>
<td>refinement failed</td>
</tr>
<tr>
<td>iteFail</td>
<td>int</td>
<td>iterative solve failed</td>
</tr>
<tr>
<td>reqGlobError</td>
<td>real</td>
<td>requested accuracy</td>
</tr>
<tr>
<td>BreakCond</td>
<td>int(*)(action)</td>
<td>check break condition</td>
</tr>
<tr>
<td>PrintBreakCond</td>
<td>void(*)()</td>
<td>print break reason</td>
</tr>
<tr>
<td>lastOp</td>
<td>action</td>
<td>see Table 48</td>
</tr>
</tbody>
</table>

Table 46: The break condition fields of the `solve` data structure

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>global requested error reached</td>
</tr>
<tr>
<td>3</td>
<td>max. nodes reached</td>
</tr>
<tr>
<td>2</td>
<td>max. depth reached</td>
</tr>
<tr>
<td>-1</td>
<td>direct solver failed</td>
</tr>
<tr>
<td>-2</td>
<td>iterative solver failed</td>
</tr>
<tr>
<td>-3</td>
<td>error estimator failed</td>
</tr>
<tr>
<td>-4</td>
<td>refinement failed</td>
</tr>
</tbody>
</table>

Table 47: Values for the `breakReason` field

<table>
<thead>
<tr>
<th>Field</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>opStart</td>
<td>no last action</td>
</tr>
<tr>
<td>opDir</td>
<td>last action Direct</td>
</tr>
<tr>
<td>opEsti</td>
<td>last action Estimate</td>
</tr>
<tr>
<td>opRef</td>
<td>last action Refine</td>
</tr>
<tr>
<td>opIte</td>
<td>last action Iterate</td>
</tr>
</tbody>
</table>

Table 48: The `Action` data type
$N$ Number of nodes
$\varepsilon_{\text{tol}}$ Requested global error (user supplied)
$\varepsilon_{\text{est}}$ Estimated global error
$\varepsilon_{\text{new}}$ Predicted global error on the new mesh
$\varepsilon_{\text{req}}$ Required accuracy for the PCG iteration
$s$ Factor for the number of new points (parameter)
$\rho$ Iteration safety factor (parameter)

Figure 9: Main iteration loop of an adaptive solution
8 Numerical methods

8.1 Direct solvers

Available direct solvers are rational Cholesky decomposition\cite{15} in full and envelope mode. They are defined by \texttt{DefFullChol()} respectively \texttt{DefEnvChol()} calls. The envelope solver resorts the nodes internally with the Reverse Cuthill/McKee algorithm\cite{7}.

8.2 Iterative solvers

Available iterative solvers are the preconditioned conjugate gradient method ([1], page 49), the preconditioned bicgstab method ([13]), and symmetric Gauß-Seidel. They are defined by \texttt{DefCGMethod()}, \texttt{DefBiCGStabMethod()}, or \texttt{DefGaussMethod()} calls.

8.3 Estimator methods

The most simple method – doing nothing – is available. It is included in the KASKADE standard error estimator defined by \texttt{DefEllErrMethod()}. The actual KASKADE error estimator \cite{6} uses the hierarchical basis for quadratic finite elements. The higher order system, whose solution should be compared with the linear solution (computed by the direct or iterative solver) is

$$\begin{pmatrix} A_{LL} & A_{LQ} \\ A_{QL} & A_{QQ} \end{pmatrix} \begin{pmatrix} U^*_L \\ U^*_Q \end{pmatrix} = \begin{pmatrix} b_L \\ b_Q \end{pmatrix}.$$ 

The difference to the linear solution $\tilde{u}_L$ is given by

$$\begin{pmatrix} d_L \\ d_Q \end{pmatrix} := \begin{pmatrix} U^*_L \\ U^*_Q \end{pmatrix} - \begin{pmatrix} \tilde{U}_L \\ 0 \end{pmatrix}$$

and satisfies

$$\begin{pmatrix} A_{LL} & A_{LQ} \\ A_{QL} & A_{QQ} \end{pmatrix} \begin{pmatrix} d_L \\ d_Q \end{pmatrix} = \begin{pmatrix} b_L - A_{LL} \tilde{U}_L \\ b_Q - A_{QQ} \tilde{U}_Q \end{pmatrix} := \begin{pmatrix} r_L \\ r_Q \end{pmatrix}.$$ 

Solving this system is too expensive, it is substituted by the reduced system

$$\begin{pmatrix} D_{LL} & 0 \\ 0 & D_{QQ} \end{pmatrix} \begin{pmatrix} \tilde{d}_L \\ \tilde{d}_Q \end{pmatrix} = \begin{pmatrix} r_L \\ r_Q \end{pmatrix}.$$
with $D_{LL}$ the nearly diagonal matrix

$$
\begin{pmatrix}
A_0 & 0 \\
0 & D
\end{pmatrix}
$$

which is used in context of the preconditioner, and $D_{QQ}$ the diagonal part of $A_{QQ}$. $A_0$ is the inverse on the coarse grid. To measure the size of global error, the energy norm is chosen and the error is approximated by

$$
|A^{1/2}d|^2 = (d, Ad) \approx (\tilde{d}, B\tilde{d}) = (D^{-1}_{LL} r_L, r_L) + (D^{-1}_{QQ} r_Q, r_Q).
$$

The term $(D^{-1}_{LL} r_L, r_L)$ is computed by the iterative solver. (In case of the availability of a direct solution 0.0 is assumed.) The other term $(D^{-1}_{QQ} r_Q, r_Q)$ is computed newly. The sum is stored to `globEps`.

As an additional result a weighted residual $\tilde{m} = (D^{-1}_{QQ} r_Q, r_Q)/n$ is computed to be used by the refinement process. The local residuals $r_Q$ are stored at `ed_res`.

### 8.4 Refinement methods

The most simple refinement method – refining all – is available. It is included in the KASKADE standard error estimator definition `DefEllErrMethod()`. Additional to methods to use the error indicators computed by the error estimator for a refinement are defined. They meanvalue strategy is described in [6], for the extrapolation strategy see [2].

### 8.5 Preconditioners

Table 49 shows the preconditioners and how they are defined.

<table>
<thead>
<tr>
<th>none</th>
<th>DefNonePreCond()</th>
<th>no preconditioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag</td>
<td>DefDiagPreCond()</td>
<td>the direct solution or $1/A_{kk}$</td>
</tr>
<tr>
<td>hb</td>
<td>DefHBPreCond()</td>
<td>hierarchical bases</td>
</tr>
<tr>
<td>bpx</td>
<td>DefBXPXPreCond()</td>
<td>BPX preconditioner (see [4])</td>
</tr>
</tbody>
</table>

Table 49: Available preconditioners


9 Graphic module

The graphic functions of KASKADE are implemented using the MiniGraphik routines. The higher level functions can be used to draw triangulations, solutions (level lines, temperature), sparse matrix structures, and simple distributions of numbers. Better graphics should be a part of an external graphic system.

9.1 Handling graphical ports

A graphical port in the KASKADE/MiniGraphik context can be a window or a file to which postscript output is written. Ports are identified by a driver identification and an additional driver specific number. All other data of a port are collected in the graphic data structure, see Table 50.

The global variable graphStreams holds an array of pointers to all graphical ports. The length is MAX_GRAPH. The following routines can be used to create and delete graphical ports.

```
graphic *NewKaskGraph()

    creates a new graphic data structure. The parameter lists (see Subsection 2.8) selectxx, colorsxx, and clippingxx are created to raise the information of the graphic data structure to the command language level. (xx is an index into the graphStreams array.)
```

```
void UpdateKaskGraphs()

    updates the information in all graphic data structures, i.e. computes the xmin etc. newly.
```

```
void CloseGraphic(int port)

    closes port graphStreams[port] and returns the graphic data structures. All parameter lists are returned. If port<0 all ports are closed.
```

9.2 Drawing

The following high level graphic subroutines are available.

```
void DrawFrame(graphic *g)

    draws a frame using graphic->xmin etc.
```
void DrawBound(graphic *g)
    draws the boundary of the current triangulation.

void DrawTri(graphic *g)
    draws the current triangulation.

void DrawIndex(graphic *g)
    draws the point indices (p->indexP).

void DrawSol(graphic *g)
    draws level lines. If graphic->levelsAt==nil, graphic->levels level lines are drawn, otherwise level lines at graphic->levelsAt[] are used.

void DrawTemp(graphic *g)
    shows the solution by coloring the triangles.

void Draw3D(graphic *g)
    shows the solution 3D.

void DrawMatStruct(graphic *g, int stiff, int rhs)
    shows the structure of the sparse matrix stiff. Zero elements in the structure are shown in red.

void DrawDistr(graphic *distrGraph, int *distr, int 1ng)
    draws a distribution.

9.3 Command language interface

The commands window, graphic, show, and todraw are defined by calling the following routine.

void SetGraphAddresses()
    defines the graphic commands.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int</td>
<td>MiniGraphic driver identification</td>
</tr>
<tr>
<td>no</td>
<td>int</td>
<td>MiniGraphic port identification</td>
</tr>
<tr>
<td>noClear</td>
<td>int</td>
<td>used to suppress first zibcl call</td>
</tr>
<tr>
<td>boundary</td>
<td>int</td>
<td>draw boundary</td>
</tr>
<tr>
<td>level</td>
<td>int</td>
<td>draw level lines</td>
</tr>
<tr>
<td>triangulation</td>
<td>int</td>
<td>draw triangulation</td>
</tr>
<tr>
<td>index</td>
<td>int</td>
<td>draw point indices</td>
</tr>
<tr>
<td>temperature</td>
<td>int</td>
<td>draw (color) temperatures</td>
</tr>
<tr>
<td>ddd</td>
<td>int</td>
<td>draw solution 3D</td>
</tr>
<tr>
<td>noOfLevels</td>
<td>int</td>
<td>draw noOfLevels levels</td>
</tr>
<tr>
<td>percentage</td>
<td>int</td>
<td>start level line at zero</td>
</tr>
<tr>
<td>toDraw</td>
<td>int</td>
<td>selected variable</td>
</tr>
<tr>
<td>automatic</td>
<td>int</td>
<td>update at certain events</td>
</tr>
<tr>
<td>sparseMat</td>
<td>int</td>
<td>show sparse matrix</td>
</tr>
<tr>
<td>distr</td>
<td>int</td>
<td>show distribution</td>
</tr>
<tr>
<td>backCol</td>
<td>int</td>
<td>background color</td>
</tr>
<tr>
<td>triCoarseCol</td>
<td>int</td>
<td>used to draw coarse triangulation</td>
</tr>
<tr>
<td>triFineCol</td>
<td>int</td>
<td>used to draw refined triangulation</td>
</tr>
<tr>
<td>dirichletCol</td>
<td>int</td>
<td>used to draw Dirichlet boundary</td>
</tr>
<tr>
<td>cauchyCol</td>
<td>int</td>
<td>used to draw Cauchy boundary</td>
</tr>
<tr>
<td>neumannCol</td>
<td>int</td>
<td>used to draw Neumann boundary</td>
</tr>
<tr>
<td>levelCol</td>
<td>int</td>
<td>used to draw level lines</td>
</tr>
<tr>
<td>xMin</td>
<td>real</td>
<td>minimal x-coordinate of triangulation</td>
</tr>
<tr>
<td>xMax</td>
<td>real</td>
<td>maximal x-coordinate of triangulation</td>
</tr>
<tr>
<td>yMin</td>
<td>real</td>
<td>minimal y-coordinate of triangulation</td>
</tr>
<tr>
<td>yMax</td>
<td>real</td>
<td>maximal y-coordinate of triangulation</td>
</tr>
<tr>
<td>xSelMin</td>
<td>real</td>
<td>selected minimal x-coordinate</td>
</tr>
<tr>
<td>xSelMax</td>
<td>real</td>
<td>selected maximal x-coordinate</td>
</tr>
<tr>
<td>ySelMin</td>
<td>real</td>
<td>selected minimal y-coordinate</td>
</tr>
<tr>
<td>ySelMax</td>
<td>real</td>
<td>selected maximal y-coordinate</td>
</tr>
<tr>
<td>rotX</td>
<td>real</td>
<td>viewing angle</td>
</tr>
<tr>
<td>rotY</td>
<td>real</td>
<td>for</td>
</tr>
<tr>
<td>rotZ</td>
<td>real</td>
<td>3D graphic</td>
</tr>
<tr>
<td>levelsAt</td>
<td>real*</td>
<td>array of values for level lines</td>
</tr>
</tbody>
</table>

Table 50: The **graphic** data structure
10 KASKADE applications

10.1 ELLKASK 2D

The ELLKASK application solves a linear scalar, second-order, elliptic equation in two dimensions:

\[-(p_{11}u_x)_x - (p_{12}u_x)_y - (p_{21}u_y)_x - (p_{22}u_y)_y + qu = g \text{ in } \Omega\]
\[u = \gamma \text{ on } \Gamma_0 \subset \partial \Omega\]
\[p_{11}u_x n_1 + p_{12}u_y n_1 + p_{21}u_x n_2 + p_{22}u_y n_2 + \eta u = \xi \text{ on } \Gamma_1.\]

with \(\Gamma_0 \cup \Gamma_1 = \partial \Omega\) and \(q(x,y) \geq 0\) and \(0 \leq \eta(x,y)\), \(P = p_{ik}(x,y)\) positive definite. Here, \(\Omega\) denotes a polygonal domain in \(\mathbb{R}^2\) and \(\Gamma_0\) is composed of edges of \(\partial \Omega\). Furthermore \(n = (n_1, n_2)\) denotes the normal vector associated with \(\Gamma_1\).

Table 5.1 shows the numerical methods included.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholfull</td>
<td>direct solver full matrix Cholesky decomposition</td>
</tr>
<tr>
<td>cholenv</td>
<td>direct solver sparse/skyline Cholesky decomposition</td>
</tr>
<tr>
<td>pcg</td>
<td>iterative solver preconditioned conjugated gradients</td>
</tr>
<tr>
<td>ssor</td>
<td>iterative solver symmetric Gauss–Seidel</td>
</tr>
<tr>
<td>pbicgstab</td>
<td>iterative solver preconditioned bicgstab</td>
</tr>
<tr>
<td>none</td>
<td>preconditioner no preconditioner</td>
</tr>
<tr>
<td>diag</td>
<td>preconditioner diagonal</td>
</tr>
<tr>
<td>hb</td>
<td>preconditioner hierarchical bases</td>
</tr>
<tr>
<td>bpx</td>
<td>preconditioner bpx</td>
</tr>
<tr>
<td>none</td>
<td>estimator no estimator</td>
</tr>
<tr>
<td>dly</td>
<td>estimator see [6]</td>
</tr>
<tr>
<td>all</td>
<td>refinement refine all</td>
</tr>
<tr>
<td>meanval</td>
<td>refinement refine with mean value threshold</td>
</tr>
<tr>
<td>extrapol</td>
<td>refinement refine with extrapolated threshold</td>
</tr>
</tbody>
</table>

Table 5.1: Numerical methods in ELLKASK

To give the user a nice start some examples and command files are included.

10.2 ELLKASK 3D

This ELLKASK version uses a modified KASKADE toolbox to handle three-dimensional problems [5].
10.3 KASTIO

KASTIO solves a parabolic equation in two space dimensions [4].

10.4 KARDOS

The program KARDOS solves a system of semilinear, parabolic initial boundary value problems in one space dimension [10]. It uses a modified KASKADE toolbox to handle one-dimensional problems too.