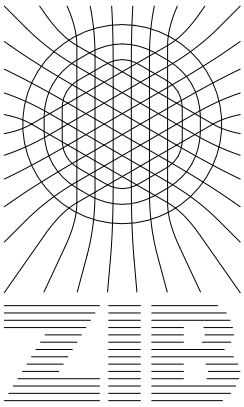
Konrad-Zuse-Zentrum für Informationstechnik Berlin



Bodo Erdmann Jens Lang Rainer Roitzsch

KASKADE Manual

Version 2.0

FEM for 2 and 3 Space Dimensions

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, irgendwen 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, I/2, 98

TR 93–5 (December 1993)

Contents

List of Figures

List of Tables

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".

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.

`real`	floating point numbers	`float` or `double`
`ptr`	pointers	`char*`
`proc`	procedure	`int(*)()`
`realproc`	real procedure	`real(*)()`
`ptrproc`	address procedure	`ptr(*)()`

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.

nil	nil pointer
true	true
false	false
ZERO	0.0
ONE	1.0
HALF	0.5
REALPI	3.14159265358979323846
REALPI2	1.57079632679489661923
SQRT2	1.41421356237309514547

Table 2: KASKADE standard constants

`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(.......)  ⟹  sprintf(globBuf,......);
                    ZIBStdOut(globBuf);
```

The buffer `globBuf` is externally defined and should be of sufficient size for standard text operations (actually 256 bytes).

`void ZIBOutFile(char *s)`

can be used to redirect standard output via `ZIBStdOut` to a file with name `s`.

`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 is in some implementations 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`.

MAX_FIXED_LISTS	100	maximum numbers of lists
BAD_FIXED_LISTS_ID	-1	wrong list id number
MAX_SMALL_SIZE	161	greatest small list size
SMALL_BUCKET_SIZE	100	minimal bucket size

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.

EFS_SECOND	-1	second call of init routine
EFS_TOO_SHORT	-2	bucket size to small
EFS_BUCKETSIZE	-3	impossible bucket size
EFS_LENGTH	-4	list length not allowed
EFS_TOO_MANY	-5	too many lists
EFS_BAD_INDEX	-6	undefined list id

Table 4: Error constants for fixed size list management

`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 `maxList` 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`.

`MAX_EVENTS`	100	maximum of events
`MAX_EVENT_TYPES`	100	maximum of event types
`MAX_EVENT_PROCS`	10	maximum of event procedures
`POS_BEFORE`	-1	insert procedure before
`POS_SUBSTITUTE`	0	substitute procedure
`POS_AFTER`	1	insert procedure after

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

EAE_SECOND	-1	second try to initialize
EAE_UNKNOWN_TYPE	-2	unknown event type
EAE_ALREADY_DEFINED_TYPE	-3	
EAE_TOO_MANY_TYPES	-4	too many event types
EAE_TOO_MANY_PROCS	-5	too many procs at event type
TAE_TRACE_SEND	1	trace of SendAppEvent
TAE_TRACE_GET	2	trace of GetAppEvent
TAE_TRACE_DEFINE	3	trace of DefineEventType
TAE_TRACE_PROC	4	trace of SetEventProc

Table 6: Error constants for application event management

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.

```
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.

`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. Parameters 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 allocation 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`.

`MAX_ACCESS_LIST`	100	maximal number of access lists

Table 7: Constants for access right management

`ACC_SECOND`	-1	second call of init routine
`ACC_LNG_NEG`	-2	list length negative
`ACC_ALL_LISTS_USED`	-3	all lists in use
`ACC_BAD_INDEX`	-4	bad access list index
`ACC_NO_INDEX`	-5	list not defined
`ACC_NO_FIELD`	-6	field not in use
`ACC_MEMORY`	-7	not enough memory
`ACC_FULL`	-8	request for space failed

Table 8: Error constants for access right management

`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_ACCESS_LIST`). 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`.

`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.

`void ReturnAccessList(int listId)`

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

`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.

`int`	`no`	command number
`int`	`noOfPars`	number of parameters
`char**`	`pars`	array of parameters
`char**`	`names`	predefined parameter names
`char*`	`results`	return string (used in tcl embedding)
`int`	`verbose`	level of verbosity (0=silent)

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.

`$Prompt`	'string'	command prompt
`$MaxPar`	number	maximal number of parameters
`$Escape`	character	escape character
`$Comment`	character	comment character
`$Quote`	character	additional quote character
`$CmdDelim`	character	additional command delimiting character
`$ParDelim`	character	additional parameter delimiting character

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`.

`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 command number in `cmd`.

MAX_COM	1000	maximum number of commands
MAX_PAR	100	maximum number of parameters
endClass	127	character class: file end
commandClass	10	character class: command end
parameterClass	5	character class: parameter end
nameClass	5	character class: name end
quoteClass	1	character class: string end

Table 11: Constants for the command language module

`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 `command` 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.

`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`.

MAX_PARAMS_LISTS	100	maximum number of parameter list
T_INT	0	constant denoting type `int`
T_REAL	1	constant denoting type `real`
T_BOOL	2	constant denoting type `bool`

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 InitParams(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.

`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.

`name`	`char*`	name of the triangulation
`fileName`	`char*`	name of the input file
`noOfPoints`	`int`	number of points
`noOfEdges`	`int`	number of edges
`noOfTriangles`	`int`	number of triangles
`refLevel`	`int`	number of refinement steps
`maxDepth`	`int`	maximal number refinements of one triangle

Table 14: Fields of the triangulation data structure `TRIANGULATION`, for more fields see Table 20

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.

`vec`	`char*`	associated storage
`boundP`	`char`	boundary condition type, see Table 16
`mark`	`char`	marking field
`classA`	`char`	additional classification
`indexP`	`int`	number of point
`x`	`real`	x–coordinate
`y`	`real`	y–coordinate
`next`	`PT*`	pointer to the next point
`last`	`PT*`	pointer to the previous point

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.

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

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.

T_WHITE	0
T_GREEN	1
T_RED	2
T_BLUE	3
T_NOTHING	4

Table 18: Values of `type` fields

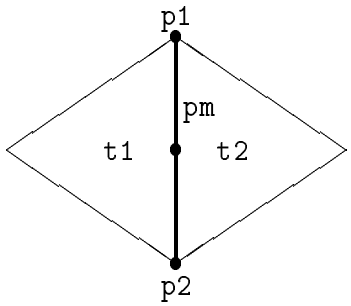


Figure 1: Field of an edge

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

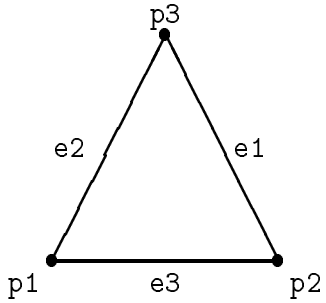


Figure 2: Position in a triangle

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 `depthFirstEdges` and `depthFirstTriangles` fields. All sublists for edges (and triangles) are linked to one list respectively. The beginning and end is stored in the `firstEdge`, `lastEdge`, `firstTriangle`, and `lastTriangle` fields.

`maxDepthFirsts`	`int`	max depth
`noOfInitPoints`	`int`	number of initial points
`noOfInitEdges`	`int`	number of initial edges
`noOfInitTriangles`	`int`	number of initial triangles
`firstPoint`	`PT*`	first point
`lastPoint`	`PT*`	last point
`initPoints`	`PT[]`	array of initial points
`firstEdge`	`EDG*`	first edge
`lastEdge`	`EDG*`	last edge
`initEdges`	`EDG[]`	array of initial edges
`depthFirstEdges`	`EDG**`	array of sublists
`firstTriangle`	`TR*`	first triangle
`lastTriangle`	`TR*`	last triangle
`initTriangles`	`TR[]`	array of initial triangles
`depthFirstTriangles`	`TR**`	array of sublists

Table 20: Additional fields of the `TRIANGULATION` data structure

3.4 Global operations on a triangulation

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

`TRIANGULATION *CrTri(char *name)`

creates a new `TRIANGULATION` data structure and initializes the fields. Additionally a list of triangulations is maintained.

`int SelTri(char *name)`

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

`int CloseTri(TRIANGULATION **trgul)`

deletes the triangulation `trgul`, updates `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):

```
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

`all`	select all
`allBackward`	select all, backwards
`allHist`	select all, include fathers
`allHistBackward`	select all, backwards, include fathers
`initial`	select initial
`dirichlet`	select objects with Dirichlet condition
`notDirichlet`	select objects with no Dirichlet condition
`initialNotDirichlet`	select initial objects with no Dirichlet condition
`boundInit`	select all initial boundary
`allReds`	select all red refined triangles

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.

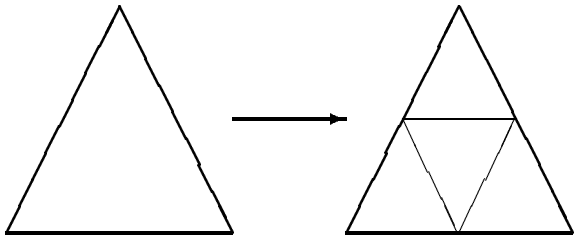


Figure 3: Red refinement

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 `isBlueDiagonal` denoting this inherits this attribute in case of (red and blue) refinement.

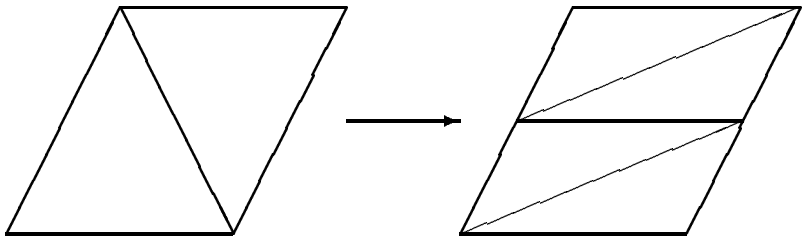


Figure 4: Blue refinement

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.

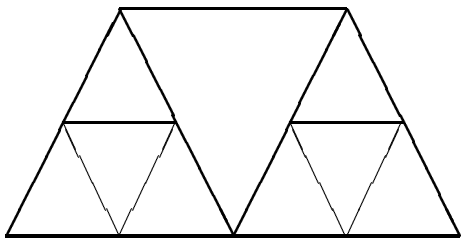


Figure 5: Middle triangle will be red refined

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.

```
int OpenRef()
```

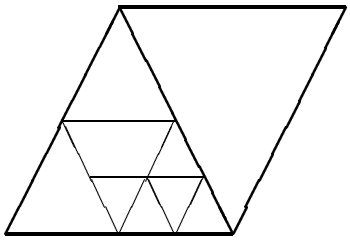


Figure 6: Right triangle will be refined red

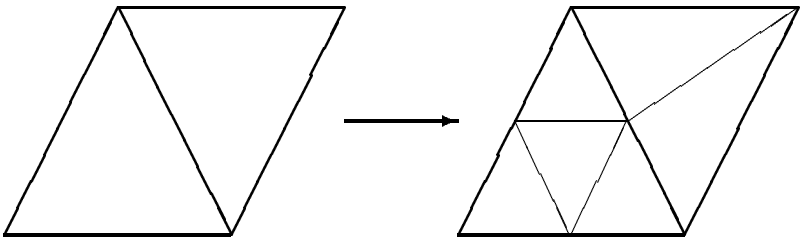


Figure 7: Green closure

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

`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).

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

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.

`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.

`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

```
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);
```

ptVec	storage id for point vector
edgVec	storage id for edge vector
trVec	storage id for triangle vector
ptWS	length of associated point vector
edgWS	length of associated edge vector
trWS	length of associated triangle vector
accPoint	access id for point vector
accEdge	access id for edge vector
accTriangle	access id for triangle vector
accNode	access id for node storage
bytesOnNode	length of node storage
noOfPointNodes	number of nodes at point
noOfEdgeNodes	number of nodes at edge
noOfTriangleNodes	number of nodes at triangle
noOfEquations	number of equations
startPointNodes	array of first byte of nodes at point
startEdgeNodes	array of first byte of nodes at edge
startTriangleNodes	array of first byte of nodes at triangle
eqnPointNodes	array of equation no of node
eqnEdgeNodes	array of equation no of node
eqnTriangleNodes	array equation no of node
iIndex	access identifier for index at nodes
cBoundP	access identifier for boundary type at nodes
cClassA	access identifier for class at nodes
rX	access x–coordinate
rY	access y–coordinate
rSol	access identifier for solution at nodes
rRhs	access identifier for right–hand side at nodes
rDiag	access identifier for diagonal at nodes
stiff	access identifier for sparse matrix
compNormP	ignore Dirichlet boundary conditions

Table 24: Fields of the `nodes` data structure (all type `int`)

```
...
...  /* Freeing access right after use */
...
   ReturnAccess(nodesState->accTriangle,triArea);
   triArea = -1;
```

`CP(p,id)`	access byte `id` from point `p`
`IP(p,id)`	access integer `id` from point `p`
`RP(p,id)`	access real `id` from point `p`
`AP(p,id)`	access address `id` from point `p`
`CE(ed,id)`	access byte `id` from edge `ed`
`IE(ed,id)`	access integer `id` from edge `ed`
`RE(ed,id)`	access real `id` from edge `ed`
`AE(ed,id)`	access address `id` from edge `ed`
`CT(t,id)`	access byte `id` from triangle `t`
`IT(t,id)`	access integer `id` from triangle `t`
`RT(t,id)`	access real `id` from triangle `t`
`AT(t,id)`	access address `id` from triangle `t`

Table 25: Access macros, first letter: data type, second letter: object

The following routine can be used to change the default storage sizes.

`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.

`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.

`void PrintPointValues(char *text, int maxVals)`
: prints data of the associated array for the first `maxVals` points.

`void PrintEdgeValues(char *text, int maxVals)`
: prints data of the associated array for the first `maxVals` edges.

`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.

`CV(vec,id)`	access byte `id` from node `vec`
`IV(vec,id)`	access integer `id` from node `vec`
`RV(vec,id)`	access real `id` from node `vec`
`AV(vec,id)`	access address `id` from node `vec`

Table 26: Access macros for node storage

access id	macro	explanation
`iIndex`	`IV(vec,iIndex)`	index of node `vec`
`cClassA`	`CV(vec,cClassA)`	index of node `vec`
`cBoundP`	`CV(vec,cBoundP)`	index of node `vec`
`rX`	`RV(vec,rX)`	x of node `vec`
`rY`	`RV(vec,rY)`	y of node `vec`
`rSol`	`RV(vec,rSol)`	solution at node `vec`

Table 27: Preset fields of nodes

```
int NodeSetUp(int nodeWorkSpace, int nodesAtPoint,
    int nodesAtEdge, int nodesAtTriangle, int noOfEquations)
```

defines the internal data structure of working space at nodes. The size `nodeWorkSpace` is reserved for each equation through the access management `nodesAtPoint` times at points, `nodesAtEdge` times at edges,

`NodesAtTriangle` times at triangles. Figure 8 shows the local storage setup at points and edges after a call of `NodeSetUp(60,1,1,0,2)`.

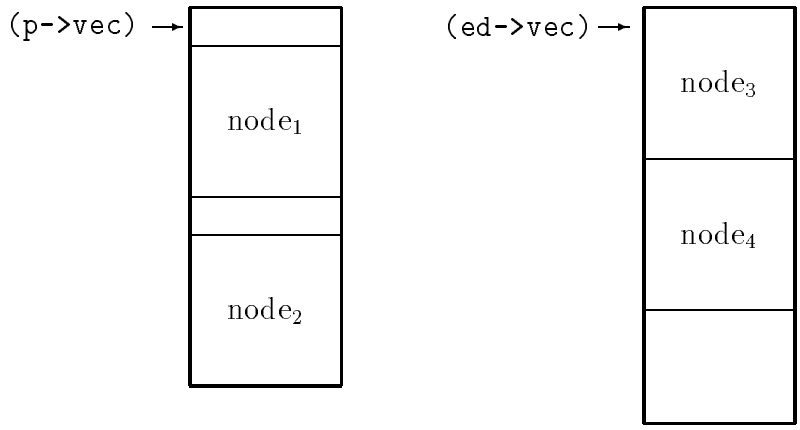


Figure 8: Node storage at points and edges

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

`void ReturnNodes()`

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

`int ApplyNode(int (*UserProc)(char*),int sel)`

applies `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 `NodeXxxx` routines. The parameter to `UserProc` is a pointer to the node storage. The selection code `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).

`NO_OF_NODE_NEIGHBORS(vec)`	retrieve the number of neighbors from `vec`
`NODE_NEIGHBOR_VEC(vec)`	retrieve the array of neighbor nodes from `vec`
`NODE_SPARSE_VEC(vec,index)`	retrieve from `vec` the line of sparse matrix `index`

Table 28: Macros to access sparse matrices

The algorithm to generate the sparse matrix now implemented is straight forward:

- 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`.

`name`	`char*`	name of the method
`rhs`	`int`	identifier right–hand side
`stiff`	`int`	identifier sparse matrix
`InitMatMul`	`int(*)(int,real**(*)(TR*), int,real*(*)(TR*))`	initialize method
`MatMul`	`int(*)(int,int)`	multiply stiffness matrix
`CloseMatMul`	`int(*)()`	close method

Table 29: The `matMulMethod` data structure

`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*)
```

prepares 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 `compNormP` in `nodesState` is set.

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

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

assembles the stiffness matrix `index` and the right–hand side `rhs`. The routines `LocAssA` and `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 `index` or `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.

`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 `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.

`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_i^k be defined

$$D_i^k u^i := -\operatorname{div}(\mathbf{P}^{ik} \operatorname{grad} u^i) + \beta_1^{ik} u_x^i + \beta_2^{ik} u_y^i + q^{ik} u^i \quad \text{with } \mathbf{P}^{ik} = \begin{pmatrix} p_{11}^{ik} & p_{12}^{ik} \\ p_{21}^{ik} & p_{22}^{ik} \end{pmatrix}$$

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

$$\begin{aligned} \sum_{i=0}^{n-1} D_i^k u^i &= g^k \quad \text{in } \Omega \quad i = 0, \ldots, n-1 \\ u^i &= \gamma_i \quad \text{on } \Gamma_0 \subset \partial\Omega \\ (\mathbf{P}^{ik} \operatorname{grad} u^i) \cdot \mathbf{n} + \eta^i u^i &= \xi^i \quad \text{on } \Gamma_1 \subset \partial\Omega \\ (\mathbf{P}^{ik} \operatorname{grad} u^i) \cdot \mathbf{n} &= 0 \quad \text{on } \partial\Omega \backslash (\Gamma_0 \cup \Gamma_1) \ . \end{aligned} \tag{1}$$

p_{jl}^{ik}, q^{ik}, β_1^{ik}, β_2^{ik}, g^k, γ^i, η^i and ξ^i are piecewise continuous real valued functions on Ω, 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

```
int InitProblem(char *name,int noOfProblems,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.

`char*`	`name`	name of the problem
`char**`	`varNames`	names of variables
`IntProc`	`Laplace`	local Laplace terms p_{jl}^{ik}
`IntProc`	`Convection`	local convection terms β_1^{ik}, β_2^{ik}
`IntProc`	`Helmholtz`	local Helmholtz term q^{ik}
`IntProc`	`Source`	local load term g^k
`IntProc`	`Obstacle`	local obstacles
`IntProc`	`Cauchy`	local Cauchy boundary functions η^i, ξ^i
`IntProc`	`Dirichlet`	local Dirichlet function γ^i
`IntProc`	`Sol`	local solution (if known)
`Real**Proc`	`LocAss`	local assembling on a triangle
`Real*Proc`	`LocAssB`	local right–hand side on a triangle

Table 31: Fields of the `problem` data structure

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`.

```
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.

```
void SetProblemAddresses()
```

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

`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 $p_{jl}^{ik}(x,y)$ in the array elements of `fVals`, i.e. $p_{11}^{ik}(x,y)$ in `fVals[0]` etc. `equation`(index k) and `variables`(index i) are valid for systems of equations. `classA` gives additional information of the geometric subdomain 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 $\beta_1^{ik}(x,y)$ and $\beta_2^{ik}(x,y)$ 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 $q^{ik}(x,y)$ 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 `problemType` 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,
```

`real *fx,real *fy, real *fxx,real *fxy,real *fyy)`

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.

$$1 - x - y$$
$$x$$
$$y$$
$$4xy$$
$$4y(1 - x - y)$$
$$4x(1 - x - y)$$

`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.

$$2(1 - x - y)(1/2 - x - y)$$
$$x(2x - 1)$$
$$y(2y - 1)$$
$$4xy$$
$$4y(1 - x - y)$$
$$4x(1 - x - y)$$

`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.

$$(1 - x - y)(3(1 - x - y) - 1)(3(1 - x - y) - 2)/2$$
$$x(3x - 1)(3x - 2)/2$$
$$y(3y - 1)(3y - 2)/2$$
$$9xy(3x - 1)/2$$
$$9xy(3y - 1)/2$$
$$9y(1 - x - y)(3y - 1)/2$$
$$9(1 - x - y)y(3(1 - x - y) - 1)/2$$
$$9(1 - x - y)x(3(1 - x - y) - 1)/2$$
$$9x(1 - x - y)(3x - 1)/2$$
$$27xy(1 - x - y)$$

6.3 Local assembling

Two data structures `integData` and `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 `InitAss` routine. The second one (see Table 34) holds the transformation data of a triangle to the reference triangle $\{0,0),(0,1),(1,0)\}$.

`integPointX`	`real*`	x_k values of integration points
`integPointY`	`real*`	y_k values of integration points
`integWeight`	`real*`	weights at the integration points
`lineIPX`	`real**`	x_{ik} for edge i
`lineIY`	`real**`	y_{ik} for edge i
`lineIW`	`real**`	weights for line integration
`shape`	`real**`	values of shape functions ϕ^i
`shapeX`	`real**`	values of shape functions ϕ^i_x
`shapeY`	`real**`	values of shape functions ϕ^i_y
`lineVals`	`real***`	values v_{ikm} of ϕ^i_y on edge k
`noOfIPoints`	`int`	number of integration points
`noOfLineIP`	`int`	number of integration points
`noOfShapeFunc`	`int`	number of shape functions
`symP`	`int`	true if symmetric

Table 33: The `integData` data structure

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

```
integData *NewIData(int iFormula, int SF,
     void (*ShapeF)(real,real,int,real*,real*,real*,real*,
     real*,real*),
     real*,real*),int symP)
```

precomputes values of the shape functions and their derivatives at the integration points. The shape functions are defined by the routine `ShapeF`, examples are `StdShape` and `StdQShape`. The parameter `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 `userNoIP`, `userIPX`, `userIPY`, and `userIW`.

`x,y`	`real*`	x_k, y_k values in triangle
`p11,p12,p21,p22`	`real*`	$p_{jl}(x_k, y_k)$ values
`q`	`real*`	$q(x_k, y_k)$ values
`g`	`real*`	$g(x_k, y_k)$ values
`beta1,beta2`	`real*`	$\beta_j(x_k, y_k)$ values
`f11,f12,f21,f22`	`real`	transformation data
`area`	`real`	area of the triangle
`classA`	`int`	class identifier of the triangle
`equation`	`int`	number of equation
`variable`	`int`	number of variable
`t`	`TR*`	pointer to the triangle

Table 34: The `localData` data structure

`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.

name	#IPs	x of IPs	y of IPs	weights of IP
`BANKIP`	3	$1/6, 2/3, 1/6$	$1/6, 1/6, 2/3$	$1/6, 1/6, 1/6$
`LINIP`	1	$1/3$	$1/3$	$1/2$
`QUADIP`	3	$1/2, 1/2, 0$	$0, 1/2, 1/2$	$1/6, 1/6, 1/6$
`USERIP`	`userNoIP`	`userIPX`	`userIPY`	`userIW`

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)`

precomputes 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.

`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

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

with the shape functions ϕ_k and $x_1 = x, x_2 = y$. The Cauchy boundary conditions are applied if necessary:

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

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

$$A_{ik} = \int_T \Big(\beta_1 \phi_k \frac{\partial}{\partial x} \phi_i + \beta_2 \phi_k \frac{\partial}{\partial y} \phi_i \Big) d(x,y) \ .$$

(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.

$$B_k \mathrel{+}= \int_{\partial T \cap \Gamma_1} \xi \phi_k ds \ .$$

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).

`real **NumAss(TR *t)`

assembles the local matrix of triangle `t` for a linear system of equations.

`real *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.

`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*`.

`name`	`char*`	name of the solver
`verboseP`	`int`	level of verbosity
`totalTime`	`real`	time (seconds)
`assTime`	`real`	time for assembly
`dirTime`	`real`	time for direct solving
`iteTime`	`real`	time for iterative solving
`estiTime`	`real`	time for error estimation
`refTime`	`real`	time for refining
`drawTime`	`real`	time for drawing
`renumbTime`	`real`	time for renumbering
`eNorm`	`real`	energy norm
`Direct`	`int(*)(dirMethod*,int)`	solves direct
`Iterate`	`int(*)(iteMethod*)`	solves iterative
`Estimate`	`int(*)(estiMethod*)`	estimates error
`Refine`	`int(*)(refMethod*)`	refines
`Solve`	`int(*)()`	adaptive solve
`ansatz`	`femAnsatz`	see Table 37
`director`	`dirMethod*`	direct data structure
`itor`	`dirMethod*`	iterative data structure
`estor`	`dirMethod*`	error estimate data structure
`reftor`	`refMethod*`	refine data structure

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

`linear`	linear ansatz
`quadratic`	quadratic ansatz
`cubic`	cubic ansatz

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.

`NewSol`	solution computed
`NewErrEsti`	error estimated
`ProblemChanged`	the problem has changed

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(*)()`	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

$$Ly = b$$

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).

`full`	full matrix
`envelope`	envelope matrix
`sparse`	sparse matrix

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)(),
```

```
int (*Assemble)(), int (*Renumber)(),
int (*PrintLS)(printMode), int (*ReleaseAll)())
```

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

7.1.3 Command language interface

`direct` is the command language interface. The actual direct solver can be changed with the `seldirect` command.

7.2 Interface for iterative solvers

Iterative solvers are called by the `Iterate` driver routine. This routine

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

The `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 `iteMethod`, see Table 42. The actual iterative method is accessed via `actItor`.

7.2.1 Iterative methods and their duties

`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 `preCondMethod` data structure contains a set of routines with some data, see Table 43.

`name`	`char*`	name of the preconditioner
`InitPreCond`	`int(*)()`	initialize
`PreCond`	`int(*)(int,int)`	preconditioning
`ClosePreCond`	`(*)()`	close

Table 43: The `preCondMethod` data structure

The routines of the method have the following duties.

`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.

```
int DefPreCondMethod(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.

`name`	`char*`	name of the error estimator
`verboseP`	`int`	level of verbosity
`ed_rq`	`int`	storage for $A_{QL}\tilde{U}_L$
`ed_diag`	`int`	storage for d_Q
`ed_rhs`	`int`	storage for b_Q
`ed_res`	`int`	storage for r_Q
`globError`	`real`	estimated global error
`mBar`	`real`	weighted residual $\bar{m}$
`sFactor`	`real`	need $s(2.0)$ times new points
`sigma`	`real`	safety factor $\sigma(0.95)$
`InitMethod`	`int(*)(iteMethod*)`	initialization method
`Method`	`int(*)(iteMethod*)`	estimation method
`CloseMethod`	`int(*)(iteMethod*)`	closing method

Table 44: The `estiMethod` data structure

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

`name`	`char*`	name of the refinement method
`verboseP`	`int`	level of verbosity
`newFactor`	`real`	$n_{\text{new}}/n_{\text{old}}$
`InitMethod`	`int(*)(iteMethod*)`	initialization method
`Method`	`int(*)(iteMethod*)`	refinement method
`CloseMethod`	`int(*)(iteMethod*)`	closing method

Table 45: The `refMethod` data structure

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.

`breakReason`	`int`	break reason, see Table 47
`maxSteps`	`int`	maximal number of steps
`curStep`	`int`	current step
`maxDepth`	`int`	maximal depth of triangulation
`maxPoints`	`int`	maximal number of points
`dirFail`	`int`	direct solver failed
`estiFail`	`int`	estimator failed
`refFail`	`int`	refinement failed
`iteFail`	`int`	iterative solve failed
`reqGlobError`	`real`	requested accuracy
`BreakCond`	`int(*)(action)`	check break condition
`PrintBreakCond`	`void(*)()`	print break reason
`lastOp`	`action`	see Table 48

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

`4`	global requested error reached
`3`	max. nodes reached
`2`	max. depth reached
`-1`	direct solver failed
`-2`	iterative solver failed
`-3`	error estimator failed
`-4`	refinement failed

Table 47: Values for the `breakReason` field

`opStart`	no last action
`opDir`	last action Direct
`opEsti`	last action Estimate
`opRef`	last action Refine
`opIte`	last action Iterate

Table 48: The `Action` data type

N	Number of nodes
ε_{tol}	Requested global error (user supplied)
ε_{est}	Estimated global error
$\varepsilon_{\text{est}}^{\text{new}}$	Predicted global error on the new mesh
ε_{req}	Required accuracy for the PCG iteration
s	Factor for the number of new points (parameter)
ρ	Iteration safety factor (parameter)

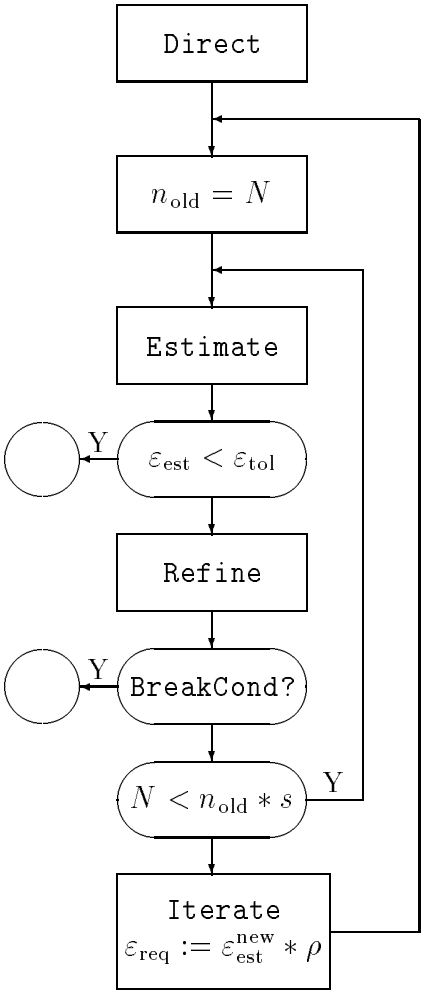


Figure 9: Main iteration loop of an adaptive solution

8 Numerical methods

8.1 Direct solvers

Available direct solvers are rational Cholesky decomposition[15] in full and envelope mode. They are defined by `DefFullChol()` respectively `DefEnvChol()` calls. The envelope solver resorts the nodes internally with the Reverse Cuthill/McKee algorithm[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 `DefCGMethod()`, `DefBiCGStabMethod()`, or `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 `DefEllErrMethod()`. The actual KASKADE error estimator [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

$$\begin{pmatrix} A_{LL} & A_{LQ} \\ A_{QL} & A_{QQ} \end{pmatrix} \begin{pmatrix} U_L^* \\ U_Q^* \end{pmatrix} = \begin{pmatrix} b_L \\ b_Q \end{pmatrix} \quad .$$

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

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

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

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

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

As an additional result a weighted residual $\bar{m} = (D_{QQ}^{-1}r_Q, r_Q)/n$ 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.

`none`	`DefNonePreCond()`	no preconditioner
`diag`	`DefDiagPreCond()`	the direct solution or $1/A_{kk}$
`hb`	`DefHBPreCond()`	hierarchical bases
`bpx`	`DefBPXPreCond()`	BPX preconditioner (see [4])

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 lng)`

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.

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

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:

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

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, Ω denotes a polygonal domain in $\mathbb{R}^2$ and Γ_0 is composed of edges of $\partial\Omega$. Furthermore $n = (n_1, n_2)$ denotes the normal vector associated with Γ_1.

Table 51 shows the numerical methods included.

`cholfull`	direct solver	full matrix Cholesky decomposition
`cholenv`	direct solver	sparse/skyline Cholesky decomposition
`pcg`	iterative solver	preconditioned conjugated gradients
`ssor`	iterative solver	symmetric Gauß–Seidel
`pbicgstab`	iterative solver	preconditioned bicgstab
`none`	preconditioner	no preconditioner
`diag`	preconditioner	diagonal
`hb`	preconditioner	hierarchical bases
`bpx`	preconditioner	bpx
`none`	estimator	no estimator
`dly`	estimator	see [6]
`all`	refinement	refine all
`meanval`	refinement	refine with mean value threshold
`extrapol`	refinement	refine with extrapolated threshold

Table 51: 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.