Konrad-Zuse-Zentrum für Informationstechnik Berlin

Rainer Roitzsch

KASKADE Programmer's Manual

Version 1.0

Technical Report TR 89–5 (August 1989)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
1000 Berlin 31
Verantwortlich: Dr. Klaus André
Umschlagsatz und Druck: Rabe KG Buch- und Offsetdruck Berlin

ISSN 0933-789X

Contents

1 Introduction

The code version KASKADE to be presented here is written in the language C. The modules of KASKADE are designed to allow reuse and extensions. Well known software engineering principles like

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

are used. Thus it is hoped to support rapid prototyping of non trivial applications.
The functionality of each module is specified by

1. a set of data types and
2. a collection of procedures.

If necessary the module interface is extended by

1. files to predefine the initial "state" of a module, e.g. a list of texts,
2. global actions (commands), or
3. internal/external test features.

KASKADE consists of a collection of modules (a library) and a prototype application (main program and files to initialize the modules).
The programmer's manual includes some recipes to change and extend KASKADE and a "complete" interface description of the modules. It tries to hide information which might change in the near future. The program is still under development, there will be changes.
To use the program the programmer is refered to the User's Manual which contains the description of the KASKADE command language.
Errors, problem reports, or any comments should be forwarded to the author at the Konrad–Zuse–Zentrum (ZIB), for an e–mail address see the Installation Guide (`README`).

1.1 C basics

The reader should know the language C, at least he should be familiar with the concept of pointers (addresses). Reading the KASKADE sources is recommended.
In the source code some naming conventions should be used:

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 (in `kask.h`), see Table 1.

`REAL`	floating point numbers	`float`
`PTR`	pointers	`char*`
`PROC`	integer functions	`int (*PROC)()`
`REALPROC`	real functions	`REAL (*REALPROC)()`
`PTRPROC`	functions returning a pointer	`PTR (*PTRPROC)()`
`VOIDPROC`	procedures	`void (*VOIDPROC)()`

Table 1: KASKADE standard data types

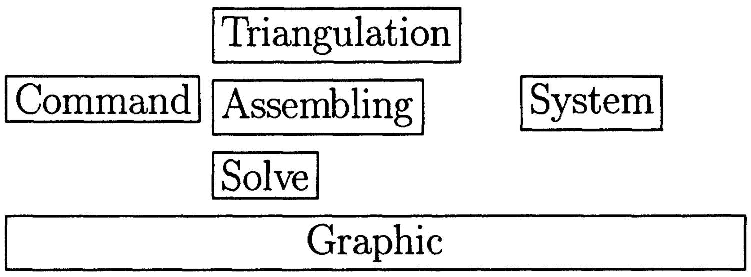
They should be used if possible like the "popular" constants in Table 2.

`nil`	nil pointer
`true`	true
`false`	false
`ZERO`	`0.0`
`ONE`	`1.0`

Table 2: KASKADE standard constants

1.2 Structure of the KASKADE program

KASKADE consists of the following modules:



The module "**Command**" implements the command language (and in the future the menu interface). It should be initialized by the main program. It reads the list of command names, numbers and their parameter names from a file. These lists have to be updated by the main program to include the real addresses of the functions which are called by "**Command**" if the user enters a command.

"**Command**" maintains a stack of command strings. It includes a procedure to stack a string of commands, extract the next command, and execute a command.

Routines to print text, which is read from a file are available too.

The modules "**Triangulation**" , "**Assembling**" , and "**Solve**" contain the code to store triangulations, to use existing triangulations, to assemble stiffness matrices, and to control the solution process. Each of these modules uses special data structures to handle more than one triangulation, more than one problem, more than one method of assembling, and more than one solution process.

They are the framework for an adaptive (2–dimensional) finite element program. The procedures for the linear elliptic PDE's are an example for their usage.

The module "**Graphic**" could be used to draw triangulations and level lines to an existing solution. There are some basic plot routines available at the command language level to generate pictures for reports. The procedures which make the graphic are using a driver through a special "primitive" interface. The PostScript driver is an example for a driver implementing this interface.

The interface to the real system (at the moment Mac II and SUN OS4) is specified in the module "**System**" . This module contains only few (trivial) procedures which are not described in this manual.

1.3 Building your own version

Just call `make`, if necessary after adding your own sources to `Makefile`.

2 Module documentation

2.1 "Command" module

"**Command**" is used to implement the interface between a user (at a terminal or writing a `do`-command, see the User's Manual [4]) and the procedures of the applications. The interface uses the `COMMAND`-data type to pass the information to application procedures.

The list of possible commands is read from a command definition file. This file consists of information for setting some modes of "**Command**" (i.e the prompt), command names followed by the command number and short description, and lists of parameters of commands.

After initialization (reading the command definition file) the linking between the unique command number and the address of the application procedure has to be done.

Command strings are stacked in "**Command**" and single commands will be extracted in the form of the `COMMAND`-data type on request. The actual processing is done by a further procedure.

Some commands are implemented as part of "**Command**" itself (i.e `do`, `quit`, `cmdinf`), see the User's Manual [4].

Furthermore procedures to read files with predefined strings and procedures to print these messages are part of "**Command**" .

2.1.1 Data types and procedures

The `COMMAND`-data type is used to pass a representation of a command to an application command procedure.

`no`	`int`	unique procedure number
`noOfPars`	`int`	number of parameters
`pars`	`char**`	array of addresses of strings which represent the parameters of the command
`keywords`	`char**`	array of addresses of strings which define the predefined keywords read from the command definition file

The "first" parameter `cmd.pars[0]` is always the actual command name, the parameter lists ends with an additional `nil` pointer. Therefore the length `cmd.pars` is `cmd.noOfPars+2`.

Some information on the state of "**Command**" is stored in a variable of the `CMDMODES`–data type .

`ignoreCases`	`int`	`true` if upper and lower case characters should be identified
`maxPars`	`int`	maximal number of parameters of a command
`prompt`	`char*`	string to a terminal prompt
`names`	`char**`	list of command names
`index`	`int*`	list of the corresponding command number
`procs`	`CMDPROCS*`	command procedure addresses
`shortDes`	`char**`	list of short command descriptions
`keywords`	`char**`	list of keywords of command
`escape`	`char`	escape character
`comment`	`char`	comment character

Furthermore information on the command stack and a character class array are stored in this data structure.

The procedures of "**Command**" are:

`int InitCommand(path,name)`
: reads the file `/path/name`. The format of the file is described in the next section. `InitCommand` returns `true` if the command file is successfully read.

`int DoCommand(line,proc)`
: puts the string `line` on the command stack. `proc` is called after the last command from `line` is executed. (Can be used to return the storage for `line`.)

`int ExecCommand(cmd)`
: executes a command by calling the procedure identified by the command number in `cmd`.

`COMMAND *GetNext()`
: extracts the next command from the command stack. This procedure does the syntax analysis under control of the `CMDMODES`–data type object `stdCmdModes`.

`void SetCommand(no,proc)`
: links the procedure address `proc` to the command number `no`.

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

`int CheckName(ptAdr,keywords,charClass`
: tests if a string is in a keyword list.

`int InitMsg(maxList)`
: initializes the procedure to read strings from `maxList` number of message files.

`int MsgList(path,name)`
: reads a list of strings from the message file `path/name`. The list may be identified by the returned list number. A failure is signaled by a return code `-1`.

`int Msg(id,no,p1,p2,p3,p4)`
: prints string number `no` from list `id` by using `printf` with parameters `p1`, `p2`, `p3`, and `p4`. A newline character is appended.

`int MsgNoNL(id,no,p1,p2,p3,p4)`
: does the same as `Msg` without an appended newline character.

2.1.2 File formats

A command definition file consists of two parts separated by a blank line. In the first part some modes of "**Command**" may be set. Each line starts with `'$'` and the mode name followed by the new value. The mode names are:

`$Prompt`	'string'	command prompt
`$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

The second part defines the command names "**Command**" 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. The next lines define the keywords. For a complete example see 2.1.4.

The message files are just simple text files, each line can be addressed separately by the procedures `Msg` and `MsgNoNL`.

2.1.3 Files

The files of "**Command**" are:

`kaskcmd.h`	header file, includes the definition of data types and externals
`command.msg`	message file, including all texts produced by "**Command**"
`kaskcmd.def`	command definition file; the definition of the KASKADE command language
`initcmd.c`	source; initialization of "**Command**"
`commands.c`	source; command interpreter
`proccmd.c`	source; some commands
`msg.c`	source; string printing procedures

2.1.4 Examples

Parts of the files `kaskcmd.def` and `command.msg` together with some extracts from KASKADE sources are reproduced to show the usage of "**Command**"

.

The command definition file `kaskcmd.def` begins with

```
$Prompt 'Kaskade: '
$MaxPars 10

quit        0 'quit Kaskade'
end         0 ''
cmdprint    1 'print command interpreter settings'
do          2 'execute command file'
msgprint    4 'print lists of messages'
```

later on keywords for `cmdprint` are defined:

```
$1
names settings
```

The message file `command.msg` includes:

```
Cmd: too many active do file(s): %d
Cmd: no code for '%s'(%d)
Cmd: nothing to do
Cmd: all done
Cmd: unknown command '%s'
Cmd: current settings of the command interpreter
   Prompt:'%s'
   Maximal number of parameters: %d
   Cases are ignored
   Escape symbol: '%c'
   Comment symbol: '%c'
Cmd: Can't read line from 'stdin'
```

In the main program of KASKADE the initialization of "**Command**" is done.

```
    if (!InitMsg(20)) { ZIBPrintf("Stoehn\n"); return; }
    msgId = MsgList(kaskPath,msgFileName);
    if (msgId<0) { ZIBPrintf("O Weh\n"); return; }
    if (!InitCommands(kaskPath, cmdDefFileName))
      { ZIBPrintf("Jammer\n"); return; }
    SetCommand( 0, CmdQuit);
    SetCommand( 1, CmdPrint);
    SetCommand( 2, CmdDo);
    SetCommand( 4, MsgPrint);
```

The loop reading lines from `stdin` and executing the commands may be:

```
while (goon) /* loop reading lines and executing commands */
  {
    if(gets(line)==nil) { Msg(cmdMsg, 3); break; }
    DoCommand(line,nil);
    while (goon) /* loop executing commands from one line */
      {
        cmd = GetNextCommand();
        if (cmd==nil) break;
        if (!ExecCommand(cmd)) cmdError = true;
        }
  }
```

The message 3 from the list `cmdMsg` is defined in the line 3 of file `commands.msg`.

```
Cmd: nothing to do
```

2.2 "Triangulation" module

The triangulation module of KASKADE handles all basic operations concerning the data structures for a triangulation. It includes methods to operate sets of points, edges or triangles too. These sets are internally defined by the module (like "all triangles of the nodal base") 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 at a time. 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 data structures are developed by P. Leinen.

2.2.1 Data Structures

The following basic data types are available:

`PT` points, including at least the (x,y)-coordinates, a boundary type descriptor, some internal data, and an array of real numbers. Direct access to the following components of the data type is available:

vec	REAL*	associated real array
x,y	REAL	x, y–coordinate
boundP	int	boundary type descriptor
indexP	int	number of the point
level	int	number of the refinement which generates this point
next	PT*	pointer to next point
last	PT*	pointer to previous point

EDG edges, including at least two pointers to the end points, a boundary type descriptor, refinement type descriptor, data to describe curved edges, some internal data, and an array of real numbers (see Figure 1). Direct access to the following components of the data type is available:

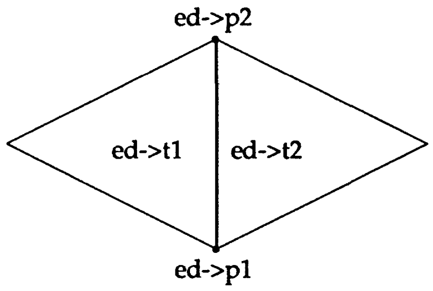


Figure 1: Information directly derivable from an edge ed

vec	REAL*	associated real array
p1,p2	PT*	pointers to end points
pm	PT*	pointer to midpoint
t1,t2	TR*	pointers to neighbor triangles
boundP	int	boundary type descriptor
MidPoint	PROC	procedure to compute the midpoint of an edge
type	int	refinement type descriptor
next	EDG*	pointer to next edge
last	EDG*	pointer to previous edge

TR triangles, including at least three pointers to the edges, refinement type descriptor, two integer numbers to specify the (KASKADE) refinement level and the refinement depth, some internal data, and an array of real numbers (see Figure 2). Direct access to the following components of the data type is available:

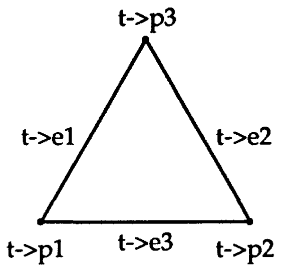


Figure 2: Information directly derivable from an triangle `t`

`vec`	`REAL*`	associated real array
`e1,e2,e3`	`EDG*`	pointers to the edges
`p1,p2,p3`	`PT*`	pointers to the points
`type`	`int`	refinement type descriptor
`level`	`int`	KASKADE refinement level
`depth`	`int`	refinement depth
`next`	`TR*`	pointer to next triangle
`last`	`TR*`	pointer to previous triangle
`son`	`TR*`	pointer to first son triangle
`father`	`TR*`	pointer to father triangle

Some assumptions on these objects are implicitly made. Their validity is maintained throughout the refinement process.

- Triangles are oriented in the mathematical positive sense. The edge i is opposite to the point i.
- midpoints transmit the boundary type of their edge.
- sons of edges have the same orientation, that is `(ed->firstSon)->p2 == ed->pm`.

The `TRIANGULATION`–data type contains all global data for a triangulation.

`name`	`char*`	name of the triangulation
`fileName`	`char*`	name of the file from which the triangulation was read
`noOfPoints`	`int`	number of points
`noOfEdges`	`int`	number of edges
`noOfTriangles`	`int`	number of triangles
`noOfInitPoints`	`int`	number of points of the initial triangulation
`noOfInitEdges`	`int`	number of edges of the initial triangulation
`noOfInitTriangles`	`int`	number of triangles of the initial triangulation
`refLevel`	`int`	number of refinements
`maxDepth`	`int`	max depth of refinement of a triangle
`problem`	PTR	pointer to the data type describing the problem
`assemble`	PTR	pointer to the data type describing the method of assembling
`solve`	PTR	pointer to the data type describing the solution process
`plot`	PTR	pointer to the data type describing current graphics settings
`firstPoint`	PT*	pointer to the first point
`lastPoint`	PT*	pointer to the last point
`initPoints`	PT*	pointer to the array of initial points
`lastDirectPoint`	PT*	pointer to the last point for which a direct solution is available
`firstEdge`	EDG*	pointer to the first edge
`lastEdge`	EDG*	pointer to the last edge
`initEdges`	EDG*	pointer to the array of initial edges
`firstTriangle`	TR*	pointer to the first triangle
`lastTriangle`	TR*	pointer to the last triangle
`initTriangles`	TR*	pointer to the array of initial triangles

In the initial triangulation the objects (points, edges, triangles) are stored consecutively in arrays. These arrays are accessibly by `actTriang->initPoints`, `actTriang->initEdges`, and `actTriang->initTriangles` respectively, i.e the first point of edge number 10 is `(actTriang->initEdges)[10].p1`. Furthermore all points and all edges are connected by a double linked list (fields `next` and `last`. New points and edges are always appended at the end of this list.

For triangles, only the objects of the actual triangulation (the nodal representation) are linked this way. New triangles always substitute at least one old one.

It is not possible to delete objects of the initial triangulation.

2.2.2 Constants

The predefined constants for boundary types (with respect to points and edges) are:

`INTERIOR`	inner point or edge
`DIRICHLET`	point/edge with Dirichlet boundary condition
`NEUMANN`	point/edge with Neumann boundary condition

The predefined constants for refinement types are:

`T_WHITE`	edge: from initial triangulation or as refined edge, triangle: not refined
`T_RED`	edge: generated by red refinement, inner edge, triangle: red refined
`T_GREEN`	edge: generated by green refinement (closure), inner edge, triangle: green refined
`T_BLUE`	edge: generated by blue refinement, inner edge, triangle: blue refined

2.2.3 Macros

The working of some operations on the basic data types should be hidden to the programmer by the use of macros (defined in `kasktri.h`). This might not only reduce the amount of typing but allows the introduction of future changes (induced by performance problems for example).

The following macros are defined:

`RA(ptr,ind)`	access to the `ind` $^{\text{th}}$–component of the `REAL` array associated to `ptr`. `ptr` might be a pointer to a point, edge, or triangle
`RD(obj,ind)`	the same for a point, edge, or triangle
`GREENEDGE(t)`	the inner edge of a green refined triangle
`INNERTRIANGLE(t)`	the inner triangle of a red refined triangle
`NEIGHBOR1(t)`	the neighbor triangle of `t` with respect to `t->e1`
`NEIGHBOR2(t)`	the neighbor triangle of `t` with respect to `t->e2`
`NEIGHBOR3(t)`	the neighbor triangle of `t` with respect to `t->e3`
`TRED1SON(t)`	the first son of a red refined triangle
`TRED2SON(t)`	the second son of a red refined triangle
`TRED3SON(t)`	the third son of a red refined triangle
`TRED4SON(t)`	the fourth son of a red refined triangle (the inner triangle)

2.2.4 Operations on triangulations

The basic idea in using the information stored in a triangulation is to apply a user supplied function `proc` to a set of points, edges or triangles in a predefined order. `proc` may stop this process by returning `false`. All apply procedures return `true` if proc returns `true` for all selected objects, else `false`.

`int ApplyP(proc,selection)`

The user supplied routine `proc` is called for each point defined by `selection` with a pointer to the point as a parameter. `selection` is one of the following predefined selections .

`all`	all points of the triangulation
`initial`	all points of the initial triangulation
`boundary`	all boundary points of the triangulation
`nonBound`	all non boundary points of the triangulation
`boundInit`	all boundary points of the initial triangulation
`dirichlet`	all dirichlet boundary points of the triangulation
`direct`	all points for which a direct solution exists
`nonDirect`	all points for which no direct solution exists

`int ApplyE(proc,selection)`

The user supplied routine `proc` is called for each edge defined by `selection` with a pointer to the edge as a parameter. `selection` is one of the following predefined selections.

`all`	all edges of the triangulation (hierarchical representation)
`allBackward`	all edges of the triangulation in reverse order (hierarchical representation)
`nodal`	all edges of the triangulation (nodal representation)
`initial`	all edges of the initial triangulation
`boundHier`	all boundary edges of the triangulation (hierarchical representation)
`boundNodal`	all boundary edges of the triangulation (nodal representation)
`boundInit`	all boundary edges of the initial triangulation
`dirichlet`	all dirichlet boundary edges of the triangulation

`int ApplyT(proc,selection)`

The user supplied routine `proc` is called for each triangle defined by `selection` with a pointer to the triangle as a parameter. `selection` is one of the following predefined selections.

`all`	all triangles of the current triangulation (nodal representation)
`initial`	all triangles of the initial triangulation

2.2.5 Procedures (Refinement/Deletion)

The standard way of refinement of a triangulation is the "red" (uniform) refinement, see Figure 3.

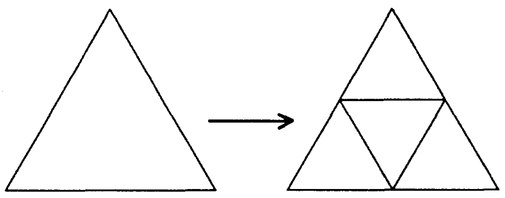


Figure 3: Red refinement

The completion (closure) is done by "green" refinement, see Figure 4.

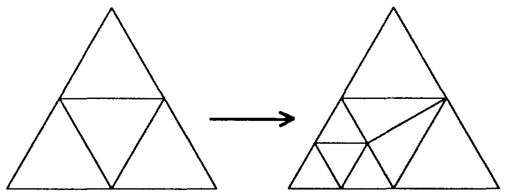


Figure 4: Green closure

Refinement is done by calling `OpenRef()`, which deletes the green closure of the triangulation. Then triangles are marked for red refinement by the procedures `RefRed(t)`. The actual refinement is invoked by `CloseRef()` which adds the green closure.

Calls to `DelTr(t)` after `OpenDel()` mark triangles do be deleted. The actual deletion is performed when `CloseDel()` is called. All brother triangles are automatically deleted too.

2.2.6 Procedures (Handling the midpoint of edges)

Two procedures are available to handle the refinement of edges. They are called by the triangulation module to compute the midpoint of an edge, interpolating a solution value (`vec[0]` of the new point), and setting the boundary value (tt vec[1]) according to the boundary type. At the moment only line or arc edges are implemented. The necessary information is stored in the data structure for an edge (`ed.MidPoint`).

2.2.7 Procedures (Creating/Deleting a triangulation)

A new triangulation is created by calling `CrTri` which returns a pointer to a new `TRIANGULATION` and makes this the current one. `CloseTri` deletes a triangulation and returns all space used for this triangulation. The current triangulation may be changed by `SelTri`.

These procedures are called by the commands to read and maintain a triangulation.

2.2.8 Global modes and variables

Some modes may be changed by setting variables of the triangulation module. A call to `VecSpace(ptDim,edgDim,trDim)` defines the length of the `REAL`–array associated with the points, edges and triangles. A minimum of `ptDim=2` and `edgDim=2` is required. These elements are used for the solution and right–side of the system.

The global variable `actTriang` of type `TRIANGULATION` gives access to the current triangulation, the one which is selected last.

2.2.9 Files

The files of "**Triangulation**" are

`kasktri.h`	header file, includes the definition of constants, data types, macros, and externals
`triang.msg`	message file, including all texts produced by "**Triangulation**"
`geointer.c`	source; apply procedures
`geobasic.c`	source; dynamic memory management of data types
`refine.c`	source; refinement
`closure.c`	source; generating and deleting green closure
`delete.c`	source; deletion
`tricmd.c`	source; commands (`inftri`, etc.)
`readtr.c`	source; command for reading a triangulation

2.2.10 Examples

Refine all triangles of a triangulation:

```
OpenRef();
  ApplyT(RefTr,all);
CloseRef();
```

A procedure to set vec[index] to zero on all points with Dirichlet boundary conditions.

```
static int gIndex;

static int SetZero(p)
  PT *p;
  {
    RA(p,gIndex) = ZERO;
    return OK;
  }
void SetBoundZero(index)
  int index;
  {
    gIndex = index;
    ApplyP(SetZero,dirichlet);
  }
```

2.3 "Assembling" module

"**Assembling**" can be used to assemble the stiffness matrix and right–hand side for a triangulation. A procedure AssTriang which computes the local stiffness matrix for a triangle is used as the central interface for the procedures which assemble the complete matrix or do a multiplication of a vector and the stiffness matrix.

"**Assembling**" contains an example for a procedure (NumAss) which uses numerical integration on linear and quadratic finite element base functions over a triangle. The information about the problem to solve is given by a set of procedures to compute the functions p_1, p_2, q, g, and the boundary values γ for the equation

$$\begin{aligned} -(p_1 u_x)_x - (p_2 u_y)_y + qu &= g \quad \mathit{in}\ \Omega \\ u &= \gamma \quad \mathit{on}\ \Gamma_0 \\ u_\nu &= 0 \quad \mathit{on}\ \partial\Omega\backslash\Gamma_0\ . \end{aligned}$$

or the corresponding variational formulation: Find a function that minimizes

$$f(u) = \frac{1}{2}a(u,u) - G(u), \quad u \in H^1_\gamma\Omega$$

with

$$\begin{aligned} a(u,v) &= \int_\Omega \left[p_1 u_x v_x + p_2 u_y v_y + quv\right] d(x,y) \\ G(v) &= \int_\Omega gvd(x,y) \end{aligned}$$

(The notation of [1] is used.) The procedure addresses for p_1, p_2, q, g, and γ are stored in the PROBLEM–data type .

The INTEGDATA–data type controls the numerical integration. The set of integration points and weights may be changed. The values of the shape functions at the integration points of the standard triangle can be precomputed.

On top of the AssTriang procedure routines to assemble the full matrix and routines to do vector operations like multiplication with the stiffness matrix are available.

2.3.1 The AssTriang interface

The AssTriang procedure is called with an address to a TR–data type . All other control is passed by the following global data. (The external declarations are part of kaskass.h.)

partP	int	P_STIFF: compute only the local stiffness matrix, P_RHS: compute only the local right–hand side, or P_ALL: compute both
symP	int	SYMMETRIC: compute only the lower triangle of the local stiffness matrix, FULL: compute the full local stiffness matrix, or DIAGONAL: compute the diagonal of the local stiffness matrix
iFrom, iTo, kFrom, kTo	int	variables to select the block of the local stiffness matrix to be computed
assA	REAL**	local stiffness matrix
assB	REAL*	local right–hand side

AssTriang should compute the matrix $A_{ik}^{(T)}$ to a given triangle T corresponding to the shape functions ϕ_i.

$$A_{ik}^{(T)} = \int_T \Big(p_1 \frac{\partial}{\partial x}\phi_k^{(T)} \frac{\partial}{\partial x}\phi_i^{(T)} + p_2 \frac{\partial}{\partial y}\phi_k^{(T)} \frac{\partial}{\partial y}\phi_i^{(T)} + q\phi_k^{(T)}\phi_i^{(T)}\Big) d(x,y)$$

For an example of a set of ϕ_i see the Section 2.3.3. The right–hand side B_i is computed in a similar fashion

$$B_i^{(T)} = \int_T g\phi_i d(x,y)$$

AssTriang itself is a global variable of type PROC.

2.3.2 The PROBLEM–data type and related procedures

The PROBLEM–data type is used to define a set of model problems. It is used by the numerical integration procedure. All REAL–functions whose addresses are mentioned in this context get the (x, y)–coordinates and optionally the enclosing triangle as parameters. The fields are:

name	char*	name of the problem
pX, pY, q, g, BValue	REALPROC	the functions p_1, p_2, q, g, and γ
pXConst	REAL	constant returned by StdpX, preset with 1
pYConst	REAL	constant returned by StdpY, preset with 1
qConst	REAL	constant returned by Stdq, preset with 0
gConst	REAL	constant returned by Stdg, preset with -1
BValueConst	REAL	constant returned by StdBValue, preset with 0

The variable actProblem is preset with the standard functions defining

$$\Delta = -1, \quad \gamma = 0$$

as the problem. Furthermore there are functions for some other model problems, see the User's Manual [4].

2.3.3 Numerical integration

The numerical integration of

$$I = \int_T F(x,y) d(x,y)$$

is done by summing up

$$I_N = \sum_{m=0}^{n} w_m F(x^{(m)}, y^{(m)})$$

The procedure NumAss transforms the actual triangle to a reference triangle first (see Figure 5) and than applies the integration formula.

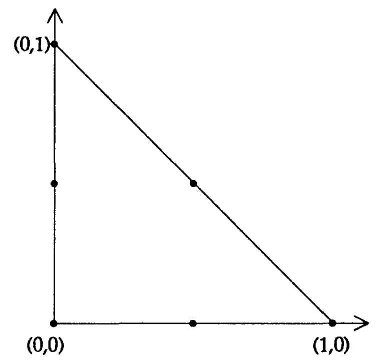


Figure 5: Reference triangle

For this triangle the values of the shape functions at the integration points are precomputed and stored in the INTEGDATA–data type , which owns the following fields

noOfIPoints	int	number of integration points n
integPointX, integPointY	REAL*	vector of integration points $x^{(m)}$, $y^{(m)}$
integWeight	REAL*	vector of weights $w^{(m)}$
noOfShapeFunc	int	number of shape functions
shape, shapeX, shapeY	REAL**	matrices with precomputed values $\phi_i(x^{(m)}, y^{(m)})$, $\frac{\partial}{\partial x}\phi_i(x^{(m)}, y^{(m)})$, and $\frac{\partial}{\partial y}\phi_i(x^{(m)}, y^{(m)})$

Two standard sets of shape functions are available. The first one is the linear set extended by 3 quadratic elements. This one is used by the KASKADE error estimator.

$$\begin{array}{lllllllll} \phi_0 & = & 1-x-y & \phi_1 & = & x & \phi_2 & = & y \\ \phi_3 & = & 4xy & \phi_4 & = & 4y(1-x-y) & \phi_5 & = & 4x(1-x-y) \end{array}$$

The quadratic shape functions are

$$\begin{array}{lll} \phi_0^{(Q)} = 2(1-x-y) & \phi_1^{(Q)} = x(2x-1) & \phi_2^{(Q)} = y(2y-1) \\ \phi_3^{(Q)} = 4xy & \phi_4^{(Q)} = 4y(1-x-y) & \phi_5^{(Q)} = 4x(1-x-y) \end{array}$$

The following procedures select integration parameters and precompute values.

`int NumAss(type)`
presets global values and precomputes values if necessary. `type` may have one of the values:

`N_STD`	select $\{\phi_i\}$, $i = 0, 1, 2$
`N_QUAD`	select $\{\phi_i^{(Q)}\}$, $i = 0, \ldots, 5$
`N_LQ`	select $\{\phi_i\}$, $i = 0, \ldots, 5$, `iFrom=3, iTo=5, kFrom=0, kTo=2`
`N_QQ`	select $\{\phi_i\}$, $i = 0, \ldots, 5$, `iFrom=3, iTo=5, kFrom=3, kTo=5, symP=DIAGONAL`

`N_LQ` and `N_QQ` are needed for the KASKADE error estimator. (In the notation of [3] A_{QL} and D_{QQ}.)

`void StdShape(x,y,k,f,fx,fy,fxx,fyy)`
computes the values of $\{\phi_k\}$ and its derivates at $(x^{(m)}, y^{(m)})$.

`void StdQShape(x,y,no,f,fx,fy,fxx,fyy)`
computes the values of $\{\phi_k^{(Q)}\}$ and its derivates at $(x^{(m)}, y^{(m)})$.

`int CompShapeVals(iData,ShapeF)`
allocates and computes the `shape`, `shapeX`, and `shapeY` matrices from iDate of type `INTEGDATA` with the values return by ShapeF, for the parameters see `StdShape`.

`INTEGDATA *NewIData(iFormula,SF,ShapeF,symP)`
allocates a new instance of an `INTEGDATA` and presets all values. `iFormula` may have the following values:

`BANKIP`	set of integration points from PLTMG [2]
`LINIP`	set of integration points for linear elements [1]
`QUADIP`	set of integration points for quadratic elements [1]
`USERIP`	set of user defined integration points[2]

`int UpdateIData(iData,iFormula,SF,ShapeF,symP)`
update `iData` as requested by `iFormula`

2.3.4 Assembling of the complete stiffness matrix

At present symmetric full matrices are handled only (no sparse packing). A matrix set of the `MATRIXSET`–data type consists of the matrices `stiff` and `decomp` and the vectors `rhs`, `diagonal`, and `solution`. This structure is used by the direct solver and later on for preconditioning.

The procedures to handle this are

`MATRIXSET *NewMatSet(length)`
: generates an element of the `MATRIXSET`–data type and allocates the memory for the matrices and vectors.

`int Assemble(mset,dim)`
: expands storage if necessary and assembles the global stiffness matrix to `mset->stiff` and the right–hand side to `mset->rhs`.

Furthermore low level routines to expand, set to zero, or print matrices and vectors are available.

2.3.5 Vector and matrix operations

The iterative solver of KASKADE uses the preconditioned conjugate gradient method. The solver needs no explicit representation of the stiffness matrix, but a procedure to multiply a vector by the matrix. The preconditioning matrix is also available as a procedure doing the multiplication.

A vector is stored in the array associated with the `PT`–data type , or in the case of quadratic elements with the `EDG`–data type . Vectors are identified by their index. Some vectors have a predefined meaning:

R_SOL	the solution, or the start values for the iterative solver
R_RHS	the right–hand side of the system
R_DIAG	the diagonal of the stiffness matrix

The most important procedures are

`int axMul(x,y)`
: multiplies the vector `x` (an integer offset to the associated arrays) by the stiffness matrix and stores the result to `y`.

`int AssRSide()`
: assembles the right–hand side to vector `R_RHS`.

`void pcxMul(x,y)`
multiplies **y** by the preconditioner $S_i^{-T}\bar{D}_i S_i^{-1}$, see [3], and stores the result to **x**.

`void SetBound(x)`
applies Dirichlet boundary conditions on vector **x**.

Furthermore low level routines to set to zero, to add vectors, to compute the inner product, to print etc. are available.

2.3.6 Files

The files of "**Assembling**" are:

`kaskass.h`	header file, includes the definition of constants, data types and externals
`assemble.msg`	message file, including all texts produced by "**Command**"
`problem.c`	source; definition of functions, the `problem`–command
`numinteg.c`	source; numerical integration
`shape.c`	source; shape functions, precomputation
`assemble.c`	source; assembling of the global stiffness matrix
`assfull.c`	source; procedures for symmetric full matrix handling
`assmul.c`	source; multiplication with the stiffness matrix, preconditioning, utilities
`asscmd.c`	source; some commands

2.4 "Solve" module

"**Solve**" is a simple frame to step through a solution process and it is the collection of the KASKADE direct solver, error estimator and iterative solver. The solution process is controlled by the `SOLVE`–data type . It is possible to "single step" through the process or to solve until certain conditions hold. Those conditions may be

1. a certain number of steps is reached,
2. a maximal number of points is reached,
3. a global error level is achieved, or
4. a failure of one step arose.

The direct solver should generate a suitable decomposition of the stiffness matrix which can be used by the preconditioner. The KASKADE direct solver implements a Cholesky decomposition.

The error estimator computes an estimation of the local errors and the global error. This values may be used by the refinement process. The KASKADE error estimator and refinement strategy is described in [3].

The iterative solver iterates until a given accuracy is achieved. KASKADE uses the preconditioned conjugate gradient method.

2.4.1 Controlling the solution process

Most of the relevant information is stored in **actSolve**, a variable of the SOLVE type, which owns the following fields:

dirSolP	int	true, if any direct solution is available
dirLevel	int	level, on which a direct solution is available
estiP	int	true, if a error estimation on the current triangulation is available, used for interpolation by the midpoint routines
dirFail	int	true, if direct solver failed
estiFail	int	true, if the error estimator failed
refineFail	int	true, if the refinement process failed
iteFail	int	true, if the iterative solver failed
maxIteSteps	int	maximum of iteration steps
iteSteps	int	number of iteration steps from last call to Iterate
lastStep	int	START: at the beginning, DIRECT: last step was direct solution, ESTIMATE: last step was error estimation, ITERATE: last step was iterative solution,
iteEps	REAL	last iterative error
globEps	REAL	last global error
breakDim	int	maximum number of points
breakNo	int	maximum number of steps
breakEps	REAL	global error level to be reached
Direct	VOIDPROC	procedure address of the direct solver
Estimate	VOIDPROC	procedure address of the error estimator
Refine	VOIDPROC	procedure address of the refinement process
Iterate	VOIDPROC	procedure address of the iterative solver

The solving process is the following calling sequence:

```
Direct();
```

```
while (!BreakCond())
  { Estimate(); Refine(); Iterate(); }
```

Setting break conditions or single step mode are implemented as command procedures, see the User's Manual [4].

2.4.2 The direct solver

The KASKADE procedure for the direct solution of the problem is called by the `Solve` or `Direct` command with parameter `verbose` set to `false` or `true` respectively. The procedure should (if successful)

- store the solution in `RA (p, R_SOL)`,
- set `actSolve->dirSOLP` to `true`,
- set `actSolve->dirLevel` to `actTriang->refLevel`,
- signal the existance of a new solution to the "**Graphic**"
- make a procedure `MulInvDirect` available, which multiplies a vector with the inverse of the stiffness matrix. This procedure may be called by the preconditioner.

If not successful if should set `actSolve->dirFail` to `true`.

The actual KASKADE direct solver assembles the complete stiffness matrix as a full symmetric matrix by a call of `Assemble`. In the next step the stiffness matrix is factorized

$$A = LDL^T$$

by Cholesky decomposition. The solution of

$$Ax = b$$

is computed by "forward" substitution

$$Ly = b$$

and "backward" substitution

$$DL^Tx = b\,.$$

Finally the result is stored in `RA(p,R_SOL)`.

2.4.3 The error estimator

The KASKADE procedure for the estimation of the global error is called by the `Estimate` or `Solve` command with parameter `verbose` set to `true` or `false` respectively.
The procedure should (if successful)

- set `actSolve->globEps` and
- compute the information which is necessary for the following refinement step.

If not successful it should set `actSolve->estiFail` to `true`.
The actual KASKADE error estimator uses the hierarchical basis for quadratic finite elements. The higher order system, whose solution should be compared with the linear solution (computed by the direct or iterative solver) is

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

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

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

and satisfies

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

To solve this system is too expensive, it is substituted by the simpler 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}. To measure the size of global error the energy norm is chosen and the error 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 newly computed. The sum is stored to `actSolve->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 `RA(ed,R_RES)`.

2.4.4 The refinement process

The KASKADE procedure refining the grid adaptively is called by the `Refine` or `Solve` command with parameter `verbose` set to `true` or `false` respectively.

The procedure should (if successful)

- refine the mesh and
- signal the existence of a new mesh to "**Graphic**"

If not successful it should set `actSolve->refineFail` to `true`.

The actual strategy implemented in the KASKADE refinement process is to refine each triangle with an edge that satisfies

$$(D_{QQ}^{1/2}\tilde{d}_Q|_{\text{edge}})^2 \geq 0.95 * \bar{m}$$

If n the number of points of the new triangulation satisfies

$$n \geq s \times n_0\,, \quad s = 2.0$$

(n_0 the number of points of the old triangulation), the refinement process stops. If not the local error estimator is called again until enough new points are found.

2.4.5 The iterative solver

The KASKADE iterative solver is called by the `Iterate` or `Solve` command with parameter `verbose` set to `true` or `false` respectively.

The procedure should (if successful)

- solve the system with the requested accuracy and
- signal the existence of a new solution to "**Graphic**" .

If not successful it should set `actSolve->iteFail` to **true**.

The preconditioned conjugate gradient method used is the "untransformed" one (see [1], p. 29, 49) with

$$C = S^{-T}DS^{-1} \ (C^{-1} = SD^{-1}S^{T})$$

D^{-1} contains the information of the Cholesky decomposition A_0^{-1} and the $1/d_{kk}$ of the stiffness matrix for the other points. The residual used to check for the accuracy is

$$g^{T}C^{-1}g \text{ with } g = A(x - \hat{x}) \ .$$

which corresponds with the linear part of the global error

$$(D_{LL}^{-1}r_L, r_L)$$

used by the error estimator.

2.4.6 Files

The files of "**Solve**" are:

`kasksol.h`	header file, includes the definition of constants, data types and externals
`gsolve.msg`	message file, including all texts produced by "**Solve**"
`solve.c`	source; command language interface of "**Solve**" , includes the procedures `Direct`, `Estimate`, `Refine`, and `Iterate`,
`cholesky.c`	source; Cholesky decomposition
`cg.c`	source; conjugate gradient method

2.5 "Graphic" module

"**Graphic**" implements some basic capabilities like drawing the boundary of a triangulation, the triangulation itself, the level lines of a solution, indices of points. These features are available at the command language level (see the User's Manual [4]). The basic settings are collected in the `GRAPHIC`–data type . This code uses a special interface to a real graphic environment.

This interface consists of the `DRIVER`–data type and a set of basic procedure calls to draw lines, fill polygons etc.

2.5.1 The GRAPHIC–data type and procedures

All coordinates in this section are in the coordinate system used when reading the triangulation. In the variable **actGraph** of type **GRAPHIC** information concerning the coordinates of the triangulation is stored, first the rectangle surrounding the triangulation then the rectangle the user wanted to draw (a zoom). Furthermore requests what to draw, how many level lines to use, etc. can be stored. The fields are:

`maxBottom, maxLeft, maxTop, maxRight`	REAL	rectangle surrounding the actual triangulation, computed by **NewGraph**
`bottom, left, top, right`	REAL	rectangle to draw
`resolution`	REAL	pixel size in user coordinates
`fontSize`	REAL	font size in user coordinates
`lineWidth`	REAL	current line width in user coordinates
`boundary`	int	`true`, if the boundary is to be drawn
`level`	int	`true`, if level lines are to be drawn
`triangulation`	int	`true`, if the triangulation is to be drawn
`index`	int	`true`, if the point indices are to be drawn
`points`	int	`true`, if points are to be drawn
`levels`	int	number of level lines to be drawn
`caption`	char*	name of the picture

The procedures (besides the command procedures) are

`GRAPHIC *NewGraph()`
: allocates a new instance of the **GRAPHIC**–data type and presets the fields.

`void DrawFrame()`
: draws the enclosing rectangle of the triangulation.

`void DrawBound()`
: draws the boundary of the triangulation.

`void DrawTri()`
: draws the triangulation.

`void DrawPoint()`
draws the points of the triangulation.

`void DrawIndex()`
draws the indices of the points of the triangulation.

`void DrawSol()`
draws the level lines of a solution.

2.5.2 Driver interface

The `DRIVER`–data type is used to maintain global information on the state of a driver. Information about the selected driver is available in the variable `actDriver`. The fields are:

`Line`	`PROC`	procedure to draw a line
`Arc`	`PROC`	procedure to draw an arc
`String`	`PROC`	procedure to draw a string
`Fill`	`PROC`	procedure to fill a polygon
`Settings`	`PROC`	procedure to set pen size and font size
`NewPict`	`PROC`	procedure to initialize a new picture
`Show`	`PROC`	procedure to output a picture
`OpenPort`	`PROC`	opening a new port
`ClosePort`	`PROC`	closing a port
`maxBottom, maxLeft, maxTop, maxRight`	`REAL`	rectangle of the output frame (driver coordinates)
`bottom, left, top, right`	`REAL`	rectangle to draw in (driver coordinates)
`fillingP`	`int`	`true`, if filling is implemented
`clippingP`	`int`	`true`, if clipping is implemented
`colorsP`	`int`	`true`, if colors are available
`graysP`	`int`	`true`, if gray scales are available
`windowNo`	`int`	port identification
`graph`	`GRAPHIC*`	associated `GRAPHIC` data type
`fontName`	`char*`	name of the current font
`fileName`	`char*`	name of the current output file, if one exists

The procedures whose addresses are stored in the DRIVER–data type are called with REAL coordinates of the triangulation coordinate system always. The return codes are true or false.

`int NewPict()`
: recomputes the scaling and does some initialization.

`int Show()`
: finishes the picture, e.g. generating a newpage or actually drawing to a window.

`int Line(p1x,p1y,p2x,p2y)`
: draws a line from p1 to p2.

`int Arc(p1x,p1y,p2x,p2y,pmx,pmy)`
: draws an arc from p1 to p2 with midpoint pm.

`int String(px,py,s)`
: draws string s starting at point p.

`int Fill(xarray,yarray,n,color)`
: fills the polygon defined by the array of points (xarray[k],yarray[k]), k=0,...n-1 with color color. Predefined colors are BLACK, YELLOW, MAGENTA, RED, CYAN, GREEN, BLUE, and WHITE.

`int Settings(type,val)`
: sets the pen size (type=PENSIZE) or the font size (type=FONTSIZE) to val. val may have the value SMALLSIZE, MEDIUMSIZE, or BIGSIZE.

`int OpenPort ()`
: opens a new port. The procedure is only used by the window interface.

`int ClosePort ()`
: closes the actual drivers port. The procedure is only used by the window interface.

Furthermore for each driver a routine NameDriver should exist:

`DRIVER *NameDriver()`
: allocates and presets a new instance of the DRIVER–data type .

2.5.3 Window interface

The window interface of KASKADE consists of the `window` command and procedures to administrate multiple windows satisfying different graphical requests, e.g. one window to show the triangulation and one to show the solution.

The `Window` command is discussed in the User's Manual [4]. The most important procedures to handle windows are:

`int AddDriver(driv)`
: adds `DRIVER *driv` to the list of drivers, maximum is `MAXDRIVER`.

`DRIVER DelDriver(driv)`
: deletes `DRIVER *driv` and returns the address of another driver.

`void AutomaticRedraw(type)`
: selects all drivers from the drivers list which have a request for `type` in their associated `GRAPHIC` data type. This procedure is called by "**Solve**" to signal a new solution or triangulation.

2.5.4 Files

The files of "**Graphic**" are:

`kaskgraph.h`	header file, includes the definition of constants, data types and externals
`graphic.msg`	message file, including all texts produced by "**Graphic**"
`graphcmd.c`	source; command language interface of "**Command**"
`graphic.c`	source; KASKADE graphic procedures
`postscr.c`	source; PostScript driver
`window.c`	source; window interface
`macwindow.c`	source; window driver for the MacII
`sunwindow.c`	source; window driver for the Sun

2.5.5 Examples

The following example illustrates the use of the basic procedures:

```
static int DrEdge(ed)
  EDG *ed;
  {
    if ((ed->father)!=nil) return true;
    (actDriver->Line)((ed->p1)->x, (ed->p1)->y,
    return true;
  }

void DrawTri()
  {
    (actDriver->Settings)(PENCOLOR, RED);
    ApplyE(DrEdge, all);
    return;
  }
```

The next example shows the "lifting" of one of these basic procedures to the command level.

```
int DrawLine(cmd)
  Command *cmd;
  {
    REAL p1x, p1y, p2x, p2y;
    int rc;

    if (ParsCheck(cmd, 2, 2)) return false;

    if (actDriver==nil)
      { Msg(graphMsg, 10); return false; }

    rc = sscanf((cmd->pars)[1], "(%e,%e)", &p1x, &p1y);
    if (rc!=2)
      { Msg(graphMsg, 23, (cmd->pars)[1]); return false; }

    rc = sscanf((cmd->pars)[2], "(%e,%e)", &p2x, &p2y);
    if (rc!=2)
      { Msg(graphMsg, 23, (cmd->pars)[2]); return false; }

    (actDriver->Line)(p1x, p1y, p2x, p2y);
    return true;
  }
```

Acknowledgments

The author thanks P. Leinen for the helpful discussions during the development of the program and R. Kornhuber for being the first user with many new requirements.

References

[1] O. Axelsson, V.A. Barker: *Finite Element Solution of Boundary Value Problems: Theory and Computation.* New York: Academic Press (1984)

[2] R.E. Bank: *PLTMG Users' Guide, Edition 5.0.* Technical Report, Department of Mathematics, University of California at San Diego (1988)

[3] P. Deuflhard, P. Leinen, H. Yserentant: *Concept of an Adaptive Hierarchical Finite Element Code.* IMPACT of Computing in Science and Engineering, I, 3-35 (1989)

[4] R. Roitzsch: *Kaskade User's Manual.* Technical Report, ZIB TR 89–4, Berlin (1989)

Index

Page numbers refering to the User's Manual are in bold face.

Veröffentlichungen des Konrad-Zuse-Zentrum für Informationstechnik Berlin
Technical Reports August 1989

TR 86-1. H. J. Schuster. *Tätigkeitsbericht (vergriffen)*

TR 87-1. Hubert Busch; Uwe Pöhle; Wolfgang Stech. *CRAY-Handbuch. - Einführung in die Benutzung der CRAY.*
TR 87-2. Herbert Melenk; Winfried Neun. *Portable Standard LISP Implementation for CRAY X–MP Computers. Release of PSL 3.4 for COS.*
TR 87-3. Herbert Melenk; Winfried Neun. *Portable Common LISP Subset Implementation for CRAY X–MP Computers.*
TR 87-4. Herbert Melenk; Winfried Neun. *REDUCE Installation Guide for CRAY 1 / X-MP Systems Running COS Version 3.3*
TR 87-5. Herbert Melenk; Winfried Neun. *REDUCE Users Guide for the CRAY 1 / X-MP Series Running COS. Version 3.3*
TR 87-6. Rainer Buhtz; Jens Langendorf; Olaf Paetsch; Danuta Anna Buhtz. *ZUGRIFF - Eine vereinheitlichte Datenspezifikation für graphische Darstellungen und ihre graphische Aufbereitung.*
TR 87-7. J. Langendorf; O. Paetsch. *GRAZIL (Graphical ZIB Language).*

TR 88-1. Rainer Buhtz; Danuta Anna Buhtz. *TDLG 3.1 - Ein interaktives Programm zur Darstellung dreidimensionaler Modelle auf Rastergraphikgeräten.*
TR 88-2. Herbert Melenk; Winfried Neun. *REDUCE User's Guide for the CRAY 1 / CRAY X-MP Series Running UNICOS. Version 3.3.*
TR 88-3. Herbert Melenk; Winfried Neun. *REDUCE Installation Guide for CRAY 1 / CRAY X-MP Systems Running UNICOS. Version 3.3.*
TR 88-4. Danuta Anna Buhtz; Jens Langendorf; Olaf Paetsch. *GRAZIL-3D. Ein graphisches Anwendungsprogramm zur Darstellung von Kurven- und Funktionsverläufen im räumlichen Koordinatensystem.*
TR 88-5. Gerhard Maierhöfer; Georg Skorobohatyj. *Parallel-TRAPEX. Ein paralleler, adaptiver Algorithmus zur numerischen Integration ; seine Implementierung für SUPRENUM-artige Architekturen mit SUSI.*

TR 89-1. *CRAY-HANDBUCH. Einführung in die Benutzung der CRAY X-MP unter UNICOS.*
TR 89-2. Peter Deuflhard. *Numerik von Anfangswertmethoden für gewöhnliche Differential-*gleichungen.
TR 89-3. Artur Rudolf Walter. *Ein Finite-Element-Verfahren zur numerischen Lösung von Erhaltungsgleichungen.*
TR 89-4. Rainer Roitzsch. *Kascade User's Manual. Version 1.0*
TR 89-5. Rainer Roitzsch. *Kascade Programmer's Manual. Version 1.0*
TR 89-6. Herbert Melenk; Winfried Neun. *Implementation of Portable Standard LISP for the SPARC Processor.*