Projekt A:Mittlere Krümmung

Martha Ludwig

Martha.Ludwig@ruhr-uni-bochum.de

16.09.2016

'Implementierung adaptiver Finite Element Verfahren',

SS 2016,RUB,bei Prof. Dr. Christian Kreuzer

Inhaltsverzeichnis

Inhaltsverzeichnis **2**

1 Einleitung **3**

2 Benötigte Dateien **4**
2.1 Makefile.in 5
2.2 Makefile_1d.in 6
2.3 Makefile_2d.in 8
2.4 Makefile_3d.in 10
2.5 nonlin.h 12
2.6 macro-big_1d.amc 13
2.7 macro_1d.amc 14
2.8 macro-big_2d.amc 15
2.9 macro_2d.amc 16
2.10 macro-big_3d.amc 17
2.11 macro_3d.amc 18
2.12 nlprob.c 19
2.13 nlsolve.c 21
2.14 graphics.c 27
2.15 nonlin_1d.dat 30
2.16 nonlin_2d.dat 31
2.17 nonlin_3d.dat 32
2.18 nonlin.c 33

3 Ausgabe **37**

I Literaturverzeichnis **38**

Index **39**

1 Einleitung

Thema des Projektes sind Hyperflächen M in $\mathbb{R}^{d+1}$ mit mittlerer Krümmung H/d,die als Graph einer Funktion $u : \Omega \to \mathbb{R}$ beschreiben werden können,
d.h.$M = \{(x, u(x)) \mid x \in \Omega\}$,
genügen der quasi-linearen Differentialgleichung

$$-\text{div}\left(\frac{\nabla u}{\sqrt{1+|\nabla u|^2}}\right) = H \text{ in } \Omega,$$
$$u = g \text{ auf } \Omega.$$

Folgende Probleme sind zu lösen:

0. Testen Sie das Programm mit der exakten Lösung
   $u(x) = \sqrt{R^2 - |x|^2} \quad x \in \Omega = B_r(0) \subset \mathbb{R}^d$,
   für unterschiedliche $r < R$ und $d = 2, 3$.

1. Berechnen Sie die minimale Oberfläche gegeben durch
   $u(x) = -\ln(|x| - \sqrt{|x|^2 - 1}) \quad x \in \Omega = (a, a+5)^2$,
   wobei $\alpha > \frac{1}{\sqrt{2}}$ gewählt ist.

2. Berechnen Sie die minimale Oberfläche M für die Randdaten g in der Polarkoordinatendarstellung
   $g(r, \varphi) = \sin(k * \varphi)$
   auf $B_1(0) \subset \mathbb{R}^2$ für unterschiedliche k.

2 Benötigte Dateien

Es werden folgende Dateien benötigt:

- Makefile.in. 2.1
- Makefile_1d.in. 2.2
- Makefile_2d.in. 2.3
- Makefile_3d.in. 2.4
- nonlin.h. 2.5
- macro-big_1d.amc. 2.6
- macro_1d.amc. 2.7
- macro-big_2d.amc. 2.8
- macro_2d.amc. 2.9
- macro-big_3d.amc. 2.10
- macro_3d.amc. 2.11
- nlprob.c. 2.12
- nlsolve.c. 2.13
- graphics.c. 2.14
- nonlin_1d.dat. 2.15
- nonlin_2d.dat. 2.16
- nonlin_3d.dat. 2.17
- nonlin.c. 2.18

```
SHELL = /bin/sh
.PHONY: clean tgz tar backup

default: tgz

DATA =  */*.[ch] *d/Makefile *d/Macro/*.amc *d/INIT/*.dat Makefile

##############################################################################
#  target for removing object and ~ files                                    #
##############################################################################

clean:
	find . \( -name \*~ -o -name \*.o -o -name \*.lo \) -exec /bin/rm -f {} \;
	find . \( -name ellipt -o -name heat -o -name nonlin \) -exec /bin/rm -f {} \;

tgz:
	tar cvf - $(DATA) | gzip -9v > Projekt_A.tgz
	@ls -l Projekt_A.tgz

tar:
	tar cvf Projekt_A.tar $(DATA)
	@ls -l Projekt_A.tar

backup:
	tar cvf - $(DATA) | gzip -9v > Projekt_A-`date '+%y%m%d.tgz'`
	@ls -l Projekt_A-`date '+%y%m%d.tgz'`
```

```
###############################################################################
#   Sample ALBERTA Makefile for 1d                                            #
###############################################################################
.PHONY: all

all: nonlin

DEFAULT = nonlin

prefix = @prefix@

# delete line, if the paths are environment variables
ALBERTA_INCLUDE_PATH = @includedir@
ALBERTA_LIB_PATH = @libdir@

# compile flags
#
# run make with "make DEBUG=1" to get debuggable code
#
ifeq ($(DEBUG),1)
CFLAGS = @ALBERTA_DEBUG_CFLAGS@ @CFLAGS@
else
CFLAGS = @ALBERTA_OPTIMIZE_CFLAGS@ @CFLAGS@
endif
FFLAGS = @FFLAGS@

# link flags (use "-shared" or "-static" to specify ALBERTA library type)
# Default is "-shared" if configuration permits it.
# Use "-all-static" to create a standalone, truly static binary.

LDFLAGS = @LDFLAGS@

# uncomment line for using the debug library
#DEBUG = 1

# if configured with --enable-el_index, this variable may be set to 1
EL_INDEX = 0

include $(ALBERTA_INCLUDE_PATH)/Makefile.alberta

###############################################################################
#   DFLAGS: DIM, DIM_OF_WORLD                                                 #
###############################################################################

DIM = 1
DIM_OF_WORLD = 1

###############################################################################
#   set virtual path                                                          #
###############################################################################
VPATH = ./:../Common

###############################################################################
#   and now the user's files                                                  #
###############################################################################

NONLIN_OFILES = nonlin.o nlprob.o nlsolve.o graphics.o
$(NONLIN_OFILES): nonlin.h

nonlin: $(NONLIN_OFILES)
	$(LINK) $(NONLIN_OFILES) $(LIBS)
```

```
.PHONY: clean realclean new

clean: albertaclean
	-rm -f nonlin

realclean: albertarealclean

new: albertanew
```

```
#!gmake
###############################################################################
#   Sample ALBERTA Makefile for 2d                                            #
###############################################################################
.PHONY: all

all: nonlin

DEFAULT = nonlin

prefix = @prefix@

# delete line, if the paths are environment variables
ALBERTA_INCLUDE_PATH = @includedir@
ALBERTA_LIB_PATH = @libdir@

# compile flags
#
# run make with "make DEBUG=1" to get debuggable code
#
ifeq ($(DEBUG),1)
CFLAGS = @ALBERTA_DEBUG_CFLAGS@ @CFLAGS@
else
CFLAGS = @ALBERTA_OPTIMIZE_CFLAGS@ @CFLAGS@
endif
FFLAGS = @FFLAGS@

# default link flags (use "-shared" or "-static" to specify ALBERTA library type)
# Default is "-shared", if configuration permits it.
# Use "-all-static" to create a standalone, truly static binary.

LDFLAGS = @LDFLAGS@

# uncomment line for using the debug library
#DEBUG = 1

# if configured with --enable-el_index, this variable may be set to 1
EL_INDEX = 0

include $(ALBERTA_INCLUDE_PATH)/Makefile.alberta

###############################################################################
#   DFLAGS: DIM, DIM_OF_WORLD                                                 #
###############################################################################

DIM = 2
DIM_OF_WORLD = 2

###############################################################################
#   set virtual path                                                          #
###############################################################################
VPATH = ./:../Common

###############################################################################
#   and now the user's files                                                  #
###############################################################################

NONLIN_OFILES = nonlin.o nlprob.o nlsolve.o graphics.o
$(NONLIN_OFILES): nonlin.h

nonlin: $(NONLIN_OFILES)
	$(LINK) $(NONLIN_OFILES) $(LIBS)
```

```
.PHONY: clean realclean new

clean: albertaclean
	-rm -f nonlin

realclean: albertarealclean

new: albertanew
```

```
#!gmake
###############################################################################
#   Sample ALBERTA Makefile for 3d                                            #
###############################################################################
.PHONY: all

all: nonlin

DEFAULT = nonlin

prefix = @prefix@

# delete line, if the paths are environment variables
ALBERTA_INCLUDE_PATH = @includedir@
ALBERTA_LIB_PATH = @libdir@

# compile flags
#
# run make with "make DEBUG=1" to get debuggable code
#
ifeq ($(DEBUG),1)
CFLAGS = @ALBERTA_DEBUG_CFLAGS@ @CFLAGS@
else
CFLAGS = @ALBERTA_OPTIMIZE_CFLAGS@ @CFLAGS@
endif
FFLAGS = @FFLAGS@

# link flags (use "-shared" or "-static" to specify ALBERTA library type)
# Default type is "-shared", if the configuration permits it.
# Use "-all-static" to create a standalone, truly static binary.

LDFLAGS = @LDFLAGS@

# uncomment line for using the debug library
#DEBUG = 1

# if configured with --enable-el_index, this variable may be set to 1
EL_INDEX = 0

include $(ALBERTA_INCLUDE_PATH)/Makefile.alberta


###############################################################################
#   DFLAGS: DIM, DIM_OF_WORLD                                                 #
###############################################################################

DIM = 3
DIM_OF_WORLD = 3

###############################################################################
#   set virtual path                                                          #
###############################################################################
VPATH = ./:../Common

###############################################################################
#   and now the user's files                                                  #
###############################################################################

NONLIN_OFILES = nonlin.o nlprob.o nlsolve.o graphics.o
$(NONLIN_OFILES): nonlin.h

nonlin: $(NONLIN_OFILES)
	$(LINK) $(NONLIN_OFILES) $(LIBS)
```

```
.PHONY: clean realclean new

clean: albertaclean
	-rm -f nonlin

realclean: albertarealclean

new: albertanew
```

```
/*----------------------------------------------------------------------*/
/*                                                                      */
/* file:     nonlin.h                                                   */
/*                                                                      */
/*----------------------------------------------------------------------*/

#include <alberta.h>

/*----------------------------------------------------------------------*/
/*  structure for collecting data of the problem:                       */
/*    g:     pointer to a function for evaluating boundary values      */
/*    f:     pointer to a function for evaluating the right-hand side  */
/*    u0:  if not nil, pointer to a function for evaluating an initial */
/*           guess for the discrete solution on the                     */
/*           macro triangulation                                        */
/*    u:     if not nil, pointer to a function for evaluating the true */
/*           solution (only for test purpose)                           */
/*    grd_u: if not nil, pointer to a function for evaluating  the      */
/*              gradient of the true solution (only for test purpose)   */
/*----------------------------------------------------------------------*/

typedef struct prob_data PROB_DATA;
struct prob_data
{
  REAL        (*g)(const REAL_D x);
  REAL        (*f)(const REAL_D x);

  REAL        (*u0)(const REAL_D x);

  REAL        (*u)(const REAL_D x);
  const REAL    *(*grd_u)(const REAL_D x);
};

/*--- file nlprob.c ----------------------------------------------------*/
const PROB_DATA *init_problem(MESH *mesh);

/*--- file nlsolve.c ---------------------------------------------------*/
int nlsolve(DOF_REAL_VEC *, REAL, REAL, REAL (*)(const REAL_D));

/*--- file graphics.c --------------------------------------------------*/
void graphics(MESH *, DOF_REAL_VEC *, REAL (*)(EL *));
```

```
DIM: 1
DIM_OF_WORLD: 1

number of vertices: 2
number of elements: 1

vertex coordinates:
-1.0
 1.0

element vertices:
0 1

element boundaries:
1 1
```

```
DIM: 1
DIM_OF_WORLD: 1

number of elements: 1
number of vertices: 2

element vertices:
0 1

element boundaries:
1 1

vertex coordinates:
 0.0
 1.0
```

```
DIM: 2
DIM_OF_WORLD: 2

number of vertices: 5
number of elements: 4

vertex coordinates:
-1.0 -1.0
 1.0 -1.0
 1.0  1.0
-1.0  1.0
 0.0  0.0

element vertices:
0 1 4
1 2 4
2 3 4
3 0 4

element boundaries:
0 0 1
0 0 1
0 0 1
0 0 1
```

```
DIM: 2
DIM_OF_WORLD: 2

number of elements: 4
number of vertices: 5

element vertices:
0 1 4
1 2 4
2 3 4
3 0 4

element boundaries:

0 0 1
0 0 1
0 0 1
0 0 1


vertex coordinates:
 0.0 0.0
 1.0 0.0
 1.0 1.0
 0.0 1.0
 0.5 0.5

element neighbours:
1 3 -1
2 0 -1
3 1 -1
0 2 -1
```

```
DIM:           3
DIM_OF_WORLD: 3

number of vertices: 8
number of elements: 6

vertex coordinates:
 -1.0 -1.0 -1.0
  1.0 -1.0 -1.0
 -1.0 -1.0  1.0
  1.0 -1.0  1.0
  1.0  1.0 -1.0
  1.0  1.0  1.0
 -1.0  1.0 -1.0
 -1.0  1.0  1.0

element vertices:
  0    5    4    1
  0    5    3    1
  0    5    3    2
  0    5    4    6
  0    5    7    6
  0    5    7    2

element boundaries:
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0

element neighbours:
 -1   -1    1    3
 -1   -1    0    2
 -1   -1    5    1
 -1   -1    4    0
 -1   -1    3    5
 -1   -1    2    4
```

```
DIM:           3
DIM_OF_WORLD: 3

number of vertices: 8
number of elements: 6

vertex coordinates:
  0.0  0.0  0.0
  1.0  0.0  0.0
  0.0  0.0  1.0
  1.0  0.0  1.0
  1.0  1.0  0.0
  1.0  1.0  1.0
  0.0  1.0  0.0
  0.0  1.0  1.0

element vertices:
  0    5    4    1
  0    5    3    1
  0    5    3    2
  0    5    4    6
  0    5    7    6
  0    5    7    2

element boundaries:
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0
  1    1    0    0

element neighbours:
 -1   -1    1    3
 -1   -1    0    2
 -1   -1    5    1
 -1   -1    4    0
 -1   -1    3    5
 -1   -1    2    4
```

```
/*----------------------------------------------------------------------*/
/*                                                                      */
/* file:     nlprob.c                                                   */
/*                                                                      */
/*----------------------------------------------------------------------*/

#include "nonlin.h"

/*----------------------------------------------------------------------*/
/*  Problem 0:  Test mit der exakten Lösung                             */
/*----------------------------------------------------------------------*/

static REAL u_0(const REAL_D x)
{
  REAL   x2 = SCP_DOW(x,x);
  return(SQR(R * R - x2));
}

static const REAL *grd_u_0(const REAL_D x)
{
  static REAL_D grd;
  REAL          ux = SQR(R * R - SCP_DOW(x,x));
  int           n;

  for (n = 0;  n < DIM_OF_WORLD; n++)
    grd[n] = -x[n] / SQR(R * R - ux);

  return(grd);
}

static REAL f_0(const REAL_D x)
{
  return(DIM_OF_WORLD / R);
}

/*----------------------------------------------------------------------*/
/*  Problem 1:  Test für die minimale Oberffläche                       */
/*----------------------------------------------------------------------*/
static REAL g_1(const REAL_D x)
{
  REAL          x2 = SCP_DOW(x,x);

  return(-ln(SQR(x2) - SQR(x2-1)));
}

static REAL f_1(const REAL_D x)
{
  REAL          x2 = SCP_DOW(x,x);
  return((2 - DIM_OF_WORLD) / x2);
}

/*----------------------------------------------------------------------*/
/*   Problem 2:  Test minimale Oberffläche M für Randdaten              */
/*----------------------------------------------------------------------*/


static REAL g_2(const REAL_D x)
{
  return(sin(k * phi));
}

static REAL f_2(const REAL_D x)
{
```

```
  REAL          cos2  = cos(k * phi) * cos(k * phi);
  REAL      Nenner  = 1 + k * cos2;
  REAL     Nenner3 = Nenner * Nenner * Nenner;
  return(k * k * sin(k * phi) / SQR(Nenner3));
}

/*--------------------------------------------------------------------------*/
/*  init_problem:                                                           */
/*    adjust function pointers and the read macro                           */
/*    triangulation                                                         */
/*--------------------------------------------------------------------------*/

const PROB_DATA *init_problem(MESH *mesh)
{
  FUNCNAME("init_problem");
  static PROB_DATA prob_data = {0};
  int             pn = 2, n_refine = 0;

  GET_PARAMETER(1, "problem number", "%d", &pn);
  switch (pn)
  {
  case 0:   /*--- Test mit der exakten Lösung  ------------------------*/
    prob_data.g = u_0;
    prob_data.f = f_0;

    prob_data.u = u_0;
    prob_data.grd_u = grd_u_0;

    read_macro(mesh, "Macro/macro-big.amc", nil);
    break;
  case 1:   /*---  Test für die minimale Oberffläche  ----------------*/
    prob_data.g = g_1;
    prob_data.f = f_1;

    prob_data.u0 = g_1;
    GET_PARAMETER(1, "U0", "%f", &U0);

    read_macro(mesh, "Macro/macro.amc", nil);
    break;
  case 2:   /*---  Test minimale Oberffläche M für Randdaten  ---*/
    prob_data.g = g_2;
    prob_data.f = f_2;
    read_macro(mesh, "Macro/macro-big.amc", nil);
    break;
  default:
    ERROR_EXIT("no problem defined with problem no. %d\n", pn);
  }
  GET_PARAMETER(1, "global refinements", "%d", &n_refine);
  global_refine(mesh, n_refine*DIM);

  return(&prob_data);
}
```

```
/*------------------------------------------------------------------------*/
/*                                                                        */
/* file:     nlsolve.c                                                    */
/*                                                                        */
/*------------------------------------------------------------------------*/

#include "nonlin.h"

/*------------------------------------------------------------------------*/
/*  data structure for the assembling of the linearized system     */
/*------------------------------------------------------------------------*/
struct op_info
{
  REAL_D  Lambda[DIM+1];    /*  the gradient of the barycentric coordinates */
  REAL    det;              /*  |det D F_S|                                 */

  const QUAD_FAST *quad_fast;  /*  quad_fast for the zero order term        */
  const REAL      *v_qp;       /*  v at all quadrature nodes of quad_fast   */
};

/*------------------------------------------------------------------------*/
/*  data structure for passing information from the                       */
/*  newton solver to the function for assembling,                         */
/*  solving and norm calculation                                          */
/*------------------------------------------------------------------------*/
typedef struct newton_data NEWTON_DATA;
struct newton_data
{
  const FE_SPACE   *fe_space;     /* used finite element space             */

  REAL       (*f)(const REAL_D);  /* for evaluation H     */

  DOF_MATRIX *DF;                 /* pointer to system matrix              */

/*---  parameters for the linear solver  ----------------------------------*/
  OEM_SOLVER solver;              /* used solver: CG (v >= 0) else BiCGStab */
  REAL       tolerance;
  int        max_iter;
  int        icon;
  int        restart;
  int        info;
};

static const REAL (*LALt(const EL_INFO *el_info, const QUAD *quad,
                         int iq, void *ud))[DIM+1]
{
  struct op_info *info = ud;
  REAL           fac = info->k*info->det;
  int            i, j, k;
  static REAL    LALt[DIM+1][DIM+1];

  for (i = 0; i <= DIM; i++)
  {
    for (j = i; j <= DIM; j++)
    {
      for (LALt[i][j] = k = 0; k < DIM_OF_WORLD; k++)
            LALt[i][j] += info->Lambda[i][k]*info->Lambda[j][k];
      LALt[i][j] *= fac;
      LALt[j][i] = LALt[i][j];
    }
  }
  return((const REAL (*)[DIM+1]) LALt);
}
```

```
static REAL c(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud)
{
  struct op_info *info = ud;
  REAL  v3;

  TEST_EXIT(info->quad_fast->quad == quad)("quads differ\n");

  v3 = info->v_qp[iq]*info->v_qp[iq]*info->v_qp[iq];

  return(info->info->det*v3);
}

static void update(void *ud, int dim, const REAL *v, int up_DF, REAL *F)
{
  FUNCNAME("update");
  static struct op_info *op_info = nil;
  static const REAL     **(*fill_a)(const EL_INFO *, void *) = nil;
  static void           *a_info = nil;
  static const REAL     **(*fill_c)(const EL_INFO *, void *) = nil;
  static void           *c_info = nil;

  static const DOF_ADMIN *admin = nil;
  static int             n_phi;
  static const REAL   *(*get_v_loc)(const EL *, const DOF_REAL_VEC *, REAL *);
  static const DOF    *(*get_dof)(const EL *, const DOF_ADMIN *, DOF *);
  static const S_CHAR *(*get_bound)(const EL_INFO *, S_CHAR *);

  NEWTON_DATA    *data = ud;
  TRAVERSE_STACK *stack = get_traverse_stack();
  const EL_INFO  *el_info;
  FLAGS          fill_flag;
  DOF_REAL_VEC   dof_v = {nil, nil, "v"};
  DOF_REAL_VEC   dof_F = {nil, nil, "F(v)"};

/*--------------------------------------------------------------------------*/
/*  init functions for assembling DF(v) and F(v)                            */
/*--------------------------------------------------------------------------*/

  if (admin != data->fe_space->admin)
  {
    OPERATOR_INFO          o_info2 = {nil}, o_info0 = {nil};
    const EL_MATRIX_INFO   *matrix_info;
    const BAS_FCTS         *bas_fcts = data->fe_space->bas_fcts;
    const QUAD             *quad = get_quadrature(DIM, 2*bas_fcts->degree-2);

    admin = data->fe_space->admin;
    n_phi      = bas_fcts->n_bas_fcts;
    get_dof    = bas_fcts->get_dof_indices;
    get_bound  = bas_fcts->get_bound;
    get_v_loc = bas_fcts->get_real_vec;;

    if (!op_info) op_info = MEM_ALLOC(1, struct op_info);

    o_info2.row_fe_space = o_info2.col_fe_space = data->fe_space;

    o_info2.quad[2]        = quad;
    o_info2.LALt           = LALt;
    o_info2.LALt_pw_const  = true;
    o_info2.LALt_symmetric = true;
    o_info2.user_data      = op_info;

    matrix_info = fill_matrix_info(&o_info2, nil);
```

```
    fill_a = matrix_info->el_matrix_fct;
    a_info = matrix_info->fill_info;

    o_info0.row_fe_space = o_info0.col_fe_space = data->fe_space;

    o_info0.quad[0]       = quad;
    o_info0.c             = c;
    o_info0.c_pw_const    = false;
    o_info0.user_data     = op_info;

    op_info->quad_fast = get_quad_fast(bas_fcts, quad, INIT_PHI);

    matrix_info = fill_matrix_info(&o_info0, nil);
    fill_c = matrix_info->el_matrix_fct;
    c_info = matrix_info->fill_info;
   }

/*--------------------------------------------------------------------------*/
/*  make a DOF vector from input vector v_vec                               */
/*--------------------------------------------------------------------------*/
   dof_v.fe_space = data->fe_space;
   dof_v.size     = dim;
   dof_v.vec      = (REAL *) v; /* cast needed; dof_v.vec isn't const REAL * */
                        /* nevertheless, values are NOT changed      */

/*--------------------------------------------------------------------------*/
/*  make a DOF vector from F, if not nil                                    */
/*--------------------------------------------------------------------------*/
   if (F)
   {
    dof_F.fe_space = data->fe_space;
    dof_F.size     = dim;
    dof_F.vec      = F;
   }

/*--------------------------------------------------------------------------*/
/*  and now assemble DF(v) and/or F(v)                                      */
/*--------------------------------------------------------------------------*/
   if (up_DF)
   {
/*--- if v_vec[i] >= 0 for all i => matrix is positive definite (p=1)  ----*/
    data->solver = dof_min(&dof_v) >= 0 ? CG : BiCGStab;
    clear_dof_matrix(data->DF);
   }

   if (F)
   {
    dof_set(0.0, &dof_F);
    L2scp_fct_bas(data->f, op_info->quad_fast->quad, &dof_F);
    dof_scal(-1.0, &dof_F);
   }

   fill_flag = CALL_LEAF_EL|FILL_COORDS|FILL_BOUND;
   el_info = traverse_first(stack, data->fe_space->mesh, -1, fill_flag);
   while (el_info)
   {
    const REAL   *v_loc = (*get_v_loc)(el_info->el, &dof_v, nil);
    const DOF    *dof   = (*get_dof)(el_info->el, admin, nil);
    const S_CHAR *bound = (*get_bound)(el_info, nil);
    const REAL   **a_mat, **c_mat;

/*--------------------------------------------------------------------------*/
/* initialization of values used by LALt and c                              */
```

```
/*--------------------------------------------------------------------------*/
   op_info->det   = el_grd_lambda(el_info, op_info->Lambda);
   op_info->v_qp = uh_at_qp(op_info->quad_fast, v_loc, nil);

   a_mat = fill_a(el_info, a_info);
   c_mat = fill_c(el_info, c_info);

   if (up_DF)  /*---  add element contribution to matrix DF(v)  ----------*/
   {
/*--------------------------------------------------------------------------*/
/*  add a(phi_i,phi_j) + 4*m(u^3*phi_i,phi_j) to matrix                     */
/*--------------------------------------------------------------------------*/
     add_element_matrix(data->DF, 1.0, n_phi, n_phi, dof, dof, a_mat, bound);
     add_element_matrix(data->DF, 4.0, n_phi, n_phi, dof, dof, c_mat, bound);
   }

   if (F)     /*---  add element contribution to F(v)  ---------------------*/
   {
     int   i, j;
/*--------------------------------------------------------------------------*/
/*  F(v) += a(v, phi_i) + m(v^4, phi_i)                                     */
/*--------------------------------------------------------------------------*/
     for (i = 0; i < n_phi; i++)
     {
           if (bound[i] < DIRICHLET)
           {
             REAL val = 0.0;
             for (j = 0; j < n_phi; j++)
               val += (a_mat[i][j] + c_mat[i][j])*v_loc[j];
             F[dof[i]] += val;
           }
           else
             F[dof[i]] = 0.0;  /*--- zero Dirichlet boundary conditions! ------*/
     }
   }

   el_info = traverse_next(stack, el_info);
  }

  free_traverse_stack(stack);

  return;
}

static int solve(void *ud, int dim, const REAL *F, REAL *d)
{
  NEWTON_DATA   *data = ud;
  int           iter;
  DOF_REAL_VEC  dof_F = {nil, nil, "F"};
  DOF_REAL_VEC  dof_d = {nil, nil, "d"};

/*--------------------------------------------------------------------------*/
/*  make DOF vectors from F and d                                           */
/*--------------------------------------------------------------------------*/
  dof_F.fe_space = dof_d.fe_space = data->fe_space;
  dof_F.size     = dof_d.size     = dim;
  dof_F.vec      = (REAL *) F; /* cast needed ...  */
  dof_d.vec      = d;

  iter = oem_solve_s(data->DF, &dof_F, &dof_d, data->solver,
                     data->tolerance, data->icon, data->restart,
                     data->max_iter, data->info);
```

```
  return(iter);
}

static REAL norm(void *ud, int dim, const REAL *v)
{
  NEWTON_DATA   *data = ud;
  DOF_REAL_VEC  dof_v = {nil, nil, "v"};

  dof_v.fe_space = data->fe_space;
  dof_v.size     = dim;
  dof_v.vec      = (REAL *) v;  /* cast needed ... */

  return(H1_norm_uh(nil, &dof_v));
}

#include <nls.h>

int nlsolve(DOF_REAL_VEC *u0, REAL (*f)(const REAL_D))
{
  FUNCNAME("nlsolve");
  static NEWTON_DATA  data = {nil, 0, 0, nil, nil, CG, 0, 500, 2, 10, 1};
  static NLS_DATA     nls_data = {nil};
  int                 iter, dim = u0->fe_space->admin->size_used;

  if (!data.fe_space)
  {
/*--------------------------------------------------------------------------*/
/*--  init parameters for newton  -----------------------------------------*/
/*--------------------------------------------------------------------------*/
    nls_data.update      = update;
    nls_data.update_data = &data;
    nls_data.solve       = solve;
    nls_data.solve_data  = &data;
    nls_data.norm        = norm;
    nls_data.norm_data   = &data;

    nls_data.tolerance = 1.e-4;
    GET_PARAMETER(1, "newton tolerance", "%e", &nls_data.tolerance);
    nls_data.max_iter = 50;
    GET_PARAMETER(1, "newton max. iter", "%d", &nls_data.max_iter);
    nls_data.info = 8;
    GET_PARAMETER(1, "newton info", "%d", &nls_data.info);
    nls_data.restart = 0;
    GET_PARAMETER(1, "newton restart", "%d", &nls_data.restart);

/*--------------------------------------------------------------------------*/
/*--  init data for update and solve  -------------------------------------*/
/*--------------------------------------------------------------------------*/

    data.fe_space = u0->fe_space;
    data.DF       = get_dof_matrix("DF(v)", u0->fe_space);

    data.tolerance = 1.e-2*nls_data.tolerance;
    GET_PARAMETER(1, "linear solver tolerance", "%f", &data.tolerance);
    GET_PARAMETER(1, "linear solver max iteration", "%d", &data.max_iter);
    GET_PARAMETER(1, "linear solver info", "%d", &data.info);
    GET_PARAMETER(1, "linear solver precon", "%d", &data.icon);
    GET_PARAMETER(1, "linear solver restart", "%d", &data.restart);
  }
  TEST_EXIT(data.fe_space == u0->fe_space)("can't change f.e. spaces\n");

/*--------------------------------------------------------------------------*/
/*--  init problem dependent parameters  ----------------------------------*/
```

```
/*-------------------------------------------------------------------------*/
  data.f     = f;

/*-------------------------------------------------------------------------*/
/*--  enlarge workspace used by newton(_fs),                             */
/*      and solve by Newton  ---------------------------------------------*/
/*-------------------------------------------------------------------------*/
  if (nls_data.restart)
  {
    nls_data.ws = REALLOC_WORKSPACE(nls_data.ws, 4*dim*sizeof(REAL));
    iter = nls_newton_fs(&nls_data, dim, u0->vec);
  }
  else
  {
    nls_data.ws = REALLOC_WORKSPACE(nls_data.ws, 2*dim*sizeof(REAL));
    iter = nls_newton(&nls_data, dim, u0->vec);
  }

  return(iter);
}
```

```
/*----------------------------------------------------------------------*/
/*                                                                      */
/* file: graphics.c                                                     */
/*                                                                      */
/*----------------------------------------------------------------------*/

#include <alberta.h>


#define USE_GLTOOLS (HAVE_LIBGLTOOLS  && true)

#if !USE_GLTOOLS
/*---8<-----------------------------------------------------------------*/
/*---   simple GL graphics ...                                     ---*/
/*---   nothing will be done in 3d                                 ---*/
/*----------------------------------------------------------------->8---*/

void graphics(MESH *mesh, DOF_REAL_VEC *u_h, REAL (*get_est)(EL *el))
{
  FUNCNAME("graphics");
  static int first = true;
  static GRAPH_WINDOW win_est = nil, win_val = nil, win_mesh = nil;
  static REAL  min=0.0, max=-1.0;
  int    refine;

#if DIM < 3
  if (first)
  {
    int size[3] = {0, 0, 0};
    char   geom[128] = "500x500+0+0";
    GET_PARAMETER(1, "graphic windows", "%d %d %d", size, size+1, size+2);
    GET_PARAMETER(1, "graphic range", "%e %e", &min, &max);

    if (size[0] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[0], size[0], 0, 0);
      win_val = graph_open_window("ALBERTA values", geom, nil, mesh);
    }
    if (size[1] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[0], size[0], size[0], 0);
      win_est = graph_open_window("ALBERTA estimate", geom, nil, mesh);
    }
    if (size[2] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[2], size[2], size[0]+size[1], 0);
      win_mesh = graph_open_window("ALBERTA mesh", geom, nil, mesh);
    }
    first = false;
  }

  if (mesh && win_mesh)
  {
    graph_clear_window(win_mesh, rgb_white);
    graph_mesh(win_mesh, mesh, rgb_black, 0);
  }
  if (u_h && win_val)
  {
    graph_clear_window(win_val, rgb_white);
    refine = u_h->fe_space->bas_fcts->degree;
    if (refine<2) refine=0;
    graph_drv(win_val, u_h, min, max, refine);
  }
```

```
  if (get_est && win_est)
  {
    graph_clear_window(win_est, rgb_white);
    graph_el_est(win_est, mesh, get_est, 0.0, 0.0);
  }
#endif
  WAIT;
  return;
}
#else

void graphics(MESH *mesh, DOF_REAL_VEC *u_h, REAL (*get_est)(EL *el))
{
  FUNCNAME("graphics");
  static int first = true;
  static GLTOOLS_WINDOW   win_est = nil, win_val = nil, win_mesh = nil;
  static REAL  min=0.0, max=-1.0;

  if (first)
  {
    int size[3] = {0, 0, 0};
    char   geom[128] = "500x500+0+0";
    GET_PARAMETER(1, "graphic windows", "%d %d %d", size, size+1, size+2);

    GET_PARAMETER(1, "graphic range", "%e %e", &min, &max);

    if (size[0] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[0], size[0], 0, 0);
      win_val = open_gltools_window("ALBERTA values", geom, nil, mesh, true);
    }

    if (size[1] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[1], size[1], size[0], 0);
      win_est = open_gltools_window("ALBERTA estimate", geom, nil, mesh, true);
    }

    if (size[2] > 0)
    {
      sprintf(geom, "%dx%d+%d+%d", size[2], size[2], size[0]+size[1], 0);
      win_mesh = open_gltools_window("ALBERTA mesh", geom, nil, mesh, true);
    }
    first = false;
    if (!(u_h && win_val) && !(get_est && win_est) && !(mesh &&win_mesh))
            return;
  }


  if (mesh && win_mesh) {
    gltools_mesh(win_mesh, mesh, 0);
  }


  if (u_h && win_val) {
    gltools_drv(win_val, u_h, min, max);
  }

  if (win_est)
  {
    if (get_est)
      gltools_est(win_est, mesh, get_est, 0.0, -1.0);
```

```
/*     else if (mesh) */
/*       gltools_mesh(win_est, mesh, 0); */
  }


  if (!win_val && !win_est)  WAIT;

  return;
}
#endif
```

```
problem number:       1
global refinements:   0
polynomial degree:    3

U0:   -2.0            % height of initial guess for Problem 1

% graphic windows: solution, estimate, and mesh if size > 0
graphic windows:         400 400 0
% for graphics you can specify the range for the values of
% discrete solution for displaying:  min max
% automatical scaling by display routine if min >= max
graphic range:  1.0 0.0

newton tolerance: 1.e-6     % tolerance for Newton
newton max. iter: 50        % maximal number of iterations of Newton
newton info:      6         % information level of Newton
newton restart:   10        % number of iterations for step size control

linear solver max iteration:  1000
linear solver restart:        10  %  only used for GMRES
linear solver tolerance:      1.e-8
linear solver info:           0
linear solver precon:         2   % 0: no precon 1: diag precon
                                  % 2: HB precon 3: BPX precon

error norm:           1   % 1: H1_NORM, 2: L2_NORM
estimator C0:         0.1 % constant of element residual
estimator C1:         0.1 % constant of jump residual
estimator C2:         0.0 % constant of coarsening estimate

adapt->strategy:        2   % 0: no adaption 1: GR 2: MS 3: ES 4:GERS
adapt->tolerance:       1.e-8
adapt->MS_gamma:        0.5
adapt->max_iteration:   15
adapt->info:            4

WAIT: 1
```

```
problem number:       2
global refinements:   1
polynomial degree:    1

U0:     -5.0          % height of initial guess for Problem 1

% graphic windows: solution, estimate, and mesh if size > 0
graphic windows:         500 500 0
% for graphics you can specify the range for the values of
% discrete solution for displaying:  min max
% automatical scaling by display routine if min >= max
graphic range:  1.0 0.0

newton tolerance: 1.e-6     % tolerance for Newton
newton max. iter: 50        % maximal number of iterations of Newton
newton info:      6         % information level of Newton
newton restart:   10        % number of iterations for step size control

linear solver max iteration:  1000
linear solver restart:        10  %  only used for GMRES
linear solver tolerance:      1.e-8
linear solver info:           0
linear solver precon:         2   % 0: no precon 1: diag precon
                                  % 2: HB precon 3: BPX precon

error norm:           1   % 1: H1_NORM, 2: L2_NORM
estimator C0:         0.1 % constant of element residual
estimator C1:         0.1 % constant of jump residual
estimator C2:         0.0 % constant of coarsening estimate

adapt->strategy:        2   % 0: no adaption 1: GR 2: MS 3: ES 4:GERS
adapt->tolerance:       1.e-1
adapt->MS_gamma:        0.5
adapt->max_iteration:   15
adapt->info:            4

WAIT: 1
```

```
problem number:       1
global refinements:   1
polynomial degree:    2

U0:   -2.0            % height of initial guess for Problem 1

% graphic windows: solution, estimate, and mesh if size > 0
graphic windows:         500 500 0
% for graphics you can specify the range for the values of
% discrete solution for displaying:  min max
% automatical scaling by display routine if min >= max
graphic range:  1.0 0.0

newton tolerance: 1.e-6     % tolerance for Newton
newton max. iter: 50        % maximal number of iterations of Newton
newton info:      6         % information level of Newton
newton restart:   10        % number of iterations for step size control

linear solver max iteration:  1000
linear solver restart:        10  %  only used for GMRES
linear solver tolerance:      1.e-8
linear solver info:           0
linear solver precon:         1   % 0: no precon 1: diag precon
                                  % 2: HB precon 3: BPX precon

error norm:           1   % 1: H1_NORM, 2: L2_NORM
estimator C0:         0.1 % constant of element residual
estimator C1:         0.1 % constant of jump residual
estimator C2:         0.0 % constant of coarsening estimate

adapt->strategy:        2   % 0: no adaption 1: GR 2: MS 3: ES 4:GERS
adapt->tolerance:       1.e-3
adapt->MS_gamma:        0.5
adapt->max_iteration:   10
adapt->info:            4

WAIT: 1
```

```
/*----------------------------------------------------------------------*/
/*                                                                      */
/* file:     nonlin.c                                                   */
/*                                                                      */
/*----------------------------------------------------------------------*/

#include "nonlin.h"

/*----------------------------------------------------------------------*/
/* global variables: finite element space, discrete solution           */
/*                   and information about problem data                 */
/*----------------------------------------------------------------------*/

static const FE_SPACE  *fe_space;        /* initialized by init_dof_admin() */
static DOF_REAL_VEC    *u_h = nil;       /* initialized by build()          */
static const PROB_DATA *prob_data = nil; /* initialized by main()           */

/*----------------------------------------------------------------------*/
/* init DOF_ADMIN: Lagrange elements of order                           */
/*  <polynomial degree>                                                 */
/* called by GET_MESH()                                                 */
/*----------------------------------------------------------------------*/

void init_dof_admin(MESH *mesh)
{
  FUNCNAME("init_dof_admin");
  int            degree;
  const BAS_FCTS  *lagrange;

  TEST_EXIT(mesh)("no MESH\n");

  degree = 1;
  GET_PARAMETER(1, "polynomial degree", "%d", &degree);
  lagrange = get_lagrange(degree);
  TEST_EXIT(lagrange)("no lagrange BAS_FCTS\n");

  fe_space = get_fe_space(mesh, lagrange->name, nil, lagrange);
  return;
}

/*----------------------------------------------------------------------*/
/*  init_leaf_data(): one REAL on each element for the                  */
/* error estimate                                                       */
/* called by GET_MESH()                                                 */
/*----------------------------------------------------------------------*/

typedef struct leaf_dat
{
  REAL est;
} LEAF_DAT;

void init_leaf_data(LEAF_DATA_INFO *leaf_data_info)
{
  FUNCNAME("init_leaf_data");
  TEST_EXIT(leaf_data_info)("no LEAF_DATA_INFO\n");

  leaf_data_info->leaf_data_size = sizeof(LEAF_DAT);
  leaf_data_info->coarsen_leaf_data = nil;
  leaf_data_info->refine_leaf_data = nil;
  return;
}

/*----------------------------------------------------------------------*/
```

```
/* rw_el_est(): read and write access to                                */
/* error estimates in LEAF_DATA                                         */
/* called by ellipt_est() in estimate()                                 */
/*----------------------------------------------------------------------*/
/* get_el_est(): reading of error estimates from LEAF_DATA   */
/* called by adapt_method_stat() and graph_el_est()                     */
/*----------------------------------------------------------------------*/

static REAL *rw_el_est(EL *el)
{
  FUNCNAME("rw_el_est");

  if (IS_LEAF_EL(el))
    return(&((LEAF_DAT *)LEAF_DATA(el))->est);
  else
  {
    ERROR("no leaf element\n");
    return(nil);
  }
}

static REAL get_el_est(EL *el)
{
  FUNCNAME("get_el_est");

  if (IS_LEAF_EL(el))
    return(((LEAF_DAT *)LEAF_DATA(el))->est);
  else
  {
    ERROR("no leaf element\n");
    return(0.0);
  }
}

/*----------------------------------------------------------------------*/
/* build(): compress the grid, access vector for discrete solution*/
/*              matrix and load vector are handled by nlsolve()        */
/* called by adapt_method_stat()                                        */
/*----------------------------------------------------------------------*/

static void build(MESH *mesh, U_CHAR flag)
{
  FUNCNAME("build");

  dof_compress(mesh);
  MSG("%d DOFs for %s\n", fe_space->admin->size_used, fe_space->name);

  if (!u_h)               /*  access and initialize discrete solution    */
  {
    u_h    = get_dof_real_vec("u_h", fe_space);
    u_h->refine_interpol = fe_space->bas_fcts->real_refine_inter;
    u_h->coarse_restrict = fe_space->bas_fcts->real_coarse_inter;
    if (prob_data->u0)
      interpol(prob_data->u0, u_h);
    else
      dof_set(0.0, u_h);
  }
  dirichlet_bound(prob_data->g, u_h, nil, nil); /* set boundary values     */

  return;
}

/*----------------------------------------------------------------------*/
```

```
/* solve(): solve the linear system                                        */
/* called by adapt_method_stat()                                           */
/*-------------------------------------------------------------------------*/

static void solve(MESH *mesh)
{
  nlsolve(u_h, prob_data->f);
  graphics(mesh, u_h, nil);
  return;
}

/*-------------------------------------------------------------------------*/
/* estimate(): calculates error estimate via ellipt_est()                  */
/*             calculates exact error also (only for test purpose)         */
/* called by adapt_method_stat()                                           */
/*-------------------------------------------------------------------------*/

static REAL r(const EL_INFO *el_info, const QUAD *quad, int iq, REAL uh_iq,
              const REAL_D grd_uh_iq)
{
  REAL_D      x;

  coord_to_world(el_info, quad->lambda[iq], x);
  return( - (*prob_data->f)(x));
}

#define EOC(e,eo) log(eo/MAX(e,1.0e-15))/M_LN2

static REAL estimate(MESH *mesh, ADAPT_STAT *adapt)
{
  FUNCNAME("estimate");
  static int     degree, norm = -1;
  static REAL    C[3] = {1.0, 1.0, 0.0};
  static REAL    est, est_old = -1.0, err = -1.0, err_old = -1.0;
  static REAL    r_flag = INIT_UH;
  REAL_DD        A = {{0.0}};
  int            n;

  for (n = 0; n < DIM_OF_WORLD; n++)
    A[n][n] = 1;  /* set diogonal of A; other elements are zero  */

  if (norm < 0)
  {
    norm = H1_NORM;
    GET_PARAMETER(1, "error norm", "%d", &norm);
    GET_PARAMETER(1, "estimator C0", "%f", C);
    GET_PARAMETER(1, "estimator C1", "%f", C+1);
    GET_PARAMETER(1, "estimator C2", "%f", C+2);
  }
  degree = 2*u_h->fe_space->bas_fcts->degree;
  est = ellipt_est(u_h, adapt, rw_el_est, nil, degree, norm, C,
                           (const REAL_D *) A, r, r_flag);

  MSG("estimate   = %.8le", est);
  if (est_old >= 0)
    print_msg(", EOC: %.2lf\n", EOC(est,est_old));
  else
    print_msg("\n");
  est_old = est;

  if (norm == L2_NORM  &&  prob_data->u)
    err = L2_err(prob_data->u, u_h, nil, 0, nil, nil);
  else if (norm == H1_NORM  &&  prob_data->grd_u)
```

```
    err = H1_err(prob_data->grd_u, u_h, nil, 0, nil, nil);

  if (err >= 0)
  {
    MSG("||u-uh||%s = %.8le", norm == L2_NORM ? "L2" : "H1", err);
    if (err_old >= 0)
      print_msg(", EOC: %.2lf\n", EOC(err,err_old));
    else
      print_msg("\n");
    err_old = err;
    MSG("||u-uh||%s/estimate = %.2lf\n", norm == L2_NORM ? "L2" : "H1",
            err/MAX(est,1.e-15));
  }
  graphics(mesh, nil, get_el_est);
  return(adapt->err_sum);
}

/*--------------------------------------------------------------------------*/
/* main program                                                             */
/*--------------------------------------------------------------------------*/

int main(int argc, char **argv)
{
  FUNCNAME("main");
  MESH      *mesh;
  ADAPT_STAT *adapt;
  int       k;

/*--------------------------------------------------------------------------*/
/*  first of all, init parameters from the init file and                    */
/*  command line                                                            */
/*--------------------------------------------------------------------------*/
  init_parameters(0, "INIT/nonlin.dat");
  for (k = 1; k+1 < argc; k += 2)
    ADD_PARAMETER(0, argv[k], argv[k+1]);

/*--------------------------------------------------------------------------*/
/*  get a mesh with DOFs and leaf data                                      */
/*--------------------------------------------------------------------------*/
  mesh = GET_MESH("Nonlinear problem mesh", init_dof_admin, init_leaf_data);

/*--------------------------------------------------------------------------*/
/*  init problem dependent data and read macro triangulation                */
/*--------------------------------------------------------------------------*/
  prob_data = init_problem(mesh);

/*--------------------------------------------------------------------------*/
/*  init adapt struture and start adaptive method                           */
/*--------------------------------------------------------------------------*/
  adapt = get_adapt_stat("nonlin", "adapt", 1, nil);
  adapt->estimate = estimate;
  adapt->get_el_est = get_el_est;
  adapt->build_after_coarsen = build;
  adapt->solve = solve;

  adapt_method_stat(mesh, adapt);

  WAIT_REALLY;
  return(0);
}
```

3 Ausgabe

Auf den nächsten Seiten sind die Ausgaben durch ALBERTA.

Teil I
Literaturverzeichnis

Kein Eintrag

Index

Dateien bnötigt, 4

PDF graphics.c 2.14, 4
PDF macro-big_1d.amc 2.6, 4
PDF macro-big_2d.amc 2.8, 4
PDF macro-big_3d.amc 2.10, 4
PDF macro_1d.amc 2.7, 4
PDF macro_2d.amc 2.9, 4
PDF macro_3d.amc 2.11, 4
PDF Makefile.in 2.1, 4
PDF Makefile_1d.in 2.2, 4
PDF Makefile_2d.in 2.3, 4
PDF Makefile_3d.in 2.4, 4
PDF nlprob.c 2.12, 4
PDF nlsolve.c 2.13, 4
PDF nonlin.c 2.18, 4
PDF nonlin.h 2.5, 4
PDF nonlin_1d.dat 2.15, 4
PDF nonlin_2d.dat 2.16, 4
PDF nonlin_3d.dat 2.17, 4