Introduction to ALBERTA

Design of Adaptive Finite Element Software
The Finite Element Toolbox ALBERTA
Springer Series: Lecture Notes in Computational Science and Engineering, Vol. 42
Schmidt, Alfred, Siebert, Kunibert G.
2005, XII, 315 p., Hardcover

www.alberta-fem.de

(Old ALBERT) Manual: `www.alberta-fem.de/Downloads/Doc/albert_doc_000331.ps.gz`

Example: Heat Equation

$$\begin{aligned} u_t - \Delta u &= f && \text{in } \Omega \subset \mathbb{R}^d \times (0,T), \\ u &= g && \text{on } \partial\Omega \times (0,T), \\ u &= u_0 && \text{on } \Omega \times \{0\}. \end{aligned}$$

Weak formulation

Find $u \in L^2(0,T; g + H_0^1(\Omega)) \cap H^1(0,T; H^{-1}(\Omega))$ s.t. for a.a. $t \in (0,T)$ and for all $\eta \in H_0^1(\Omega)$

$$\langle u_t, \eta \rangle + (\nabla u, \nabla \eta) = (f, \eta)$$
$$u(\cdot, 0) = u_0$$

Finite Element Approximation

For $n = 1, \ldots, N$, find $u_n^h \in X_n^h(g_h)$ s.t. for all $\varphi^h \in X_n^h(0)$

$$\left(\frac{u_n^h - I_n^h[u_{n-1}^h]}{\tau_n}, \varphi^h \right) + (\nabla u_n^h, \nabla \varphi^h) = (f(\cdot, t_n), \varphi^h),$$

where $u_0^h = I_0^h[u_0]$.

Matrix notation

$$(\frac{1}{\tau_n}\,\mathcal{M} + \mathcal{A})\,U_n = \frac{1}{\tau_n}\,\mathcal{M}\,U_{n-1} + F_n$$

- $\mathcal{M}$ mass matrix, $\mathcal{M}_{ij} = (\varphi_i, \varphi_j)$.

- $\mathcal{A}$ stiffness matrix, $\mathcal{A}_{ij} = (\nabla\varphi_i, \nabla\varphi_j)$.

These matrices need to be assembled at every time step.

ALBERTA has predefined assemblage routines for any second order operator of the form

$$Lu = -\nabla\,.\,A(x)\,\nabla u + b(x)\,.\,\nabla u + c(x)\,u$$

Matrix assembly

Assembly is done element-wise. On each element, functions and coefficients are evaluated in **barycentric** coordinates on the standard reference element.

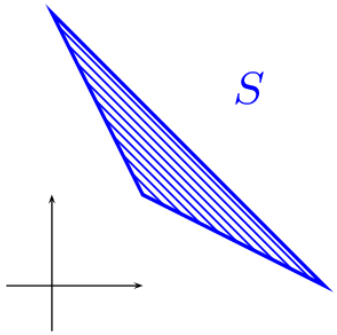
Given an element $S = \text{conv hull } \{p_0, \ldots, p_d\}$, we define the standard reference element $\widehat{S} := \text{conv hull } \{\hat{p}_0 = 0, \hat{p}_1 = e_1, \ldots, \hat{p}_d = e_d\}$ and the reference simplex $\bar{S} := \{(\lambda_0, \ldots, \lambda_d) : \lambda_k \geq 0, \sum_{k=1}^{d} \lambda_k = 1\}$.

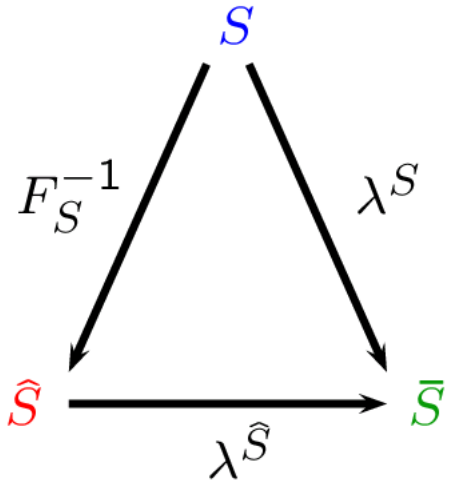
Then the following mappings are defined.

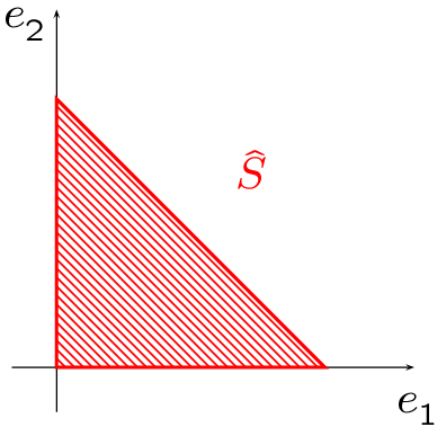
- The diffeomorphic parameterization $F_S : \widehat{S} \to S$, s.t. $F_S(\hat{p}_k) = p_k$, $k = 0, \ldots, d$.
- The barycentric coordinates $\lambda^S : S \to \bar{S}$, s.t. $\sum_{k=0}^{d} \lambda_k^S(x)\, p_k = x$, $\sum_{k=0}^{d} \lambda_k^S(x) = 1$.
- The barycentric coordinates $\lambda^{\widehat{S}} : \widehat{S} \to \bar{S}$.

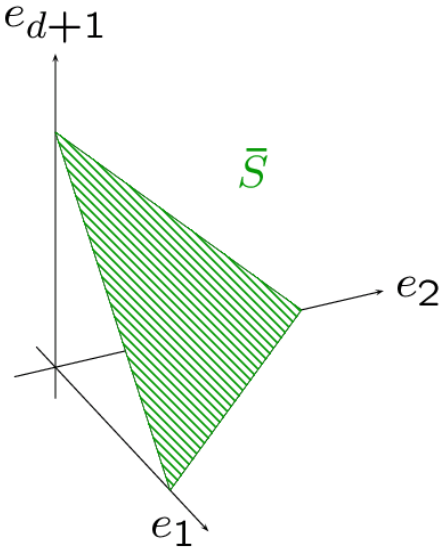
Note that given barycentric coordinates $\lambda \in \bar{S}$, the "world coordinates" are defined by $x^S(\lambda) := \sum_{k=0}^{d} \lambda_k\, p_k \in S$.

Coordinate systems









Matrix assembly

That yields e.g. for the second order term in L:

$$\begin{aligned}\mathcal{A}_{ij} &= \int_S \nabla\varphi_i(x)\,.\,A(x)\,\nabla\varphi_j(x)\;\mathrm{d}x = \int_S \nabla(\bar{\varphi}_{i_S(i)}\circ\lambda)(x)\,.\,A(x)\,\nabla(\bar{\varphi}_{i_S(j)}\circ\lambda)(x)\;\mathrm{d}x \\ &= \int_S \nabla_\lambda\bar{\varphi}_{i_S(i)}(\lambda(x))\,.\,\Lambda(x)\,A(x)\,\Lambda^T(x)\,\nabla_\lambda\bar{\varphi}_{i_S(j)}(\lambda(x))\;\mathrm{d}x,\end{aligned}$$

where Λ is the Jacobian of the barycentric coordinates λ on S.

On transforming the integral to the reference simplex $\widehat{S}$, we have that

$$\mathcal{A}_{ij} = \int_{\widehat{S}} \nabla_\lambda\bar{\varphi}_{i_S(i)}(\lambda(\hat{x}))\,.\,\bar{A}(\lambda(\hat{x}))\,\nabla_\lambda\bar{\varphi}_{i_S(j)}(\lambda(\hat{x}))\;\mathrm{d}\hat{x},$$

where $\bar{A}(\lambda) := |\det DF_S(\hat{x}(\lambda))|\,\Lambda(x(\lambda))\,A(x(\lambda))\,\Lambda^T(x(\lambda))$.

All we have to implement in ALBERTA *is* $\bar{A}(\lambda)$!

Where $|\det DF_S|$ and Λ are given by ALBERTA.

Matrix assembly

In our example we have $Lu = -\Delta u$.

Hence $A(x) \equiv I$ and $\bar{A}(\lambda) := |\det DF_S(\hat{x}(\lambda))| \Lambda(x(\lambda)) \Lambda^T(x(\lambda))$.

```
struct op_info {
  REAL_D  Lambda[DIM+1];     /*  the gradient of the barycentric coordinates */
  REAL    det;               /*  |det D F_S|                                 */
  REAL    tau_1;
};

/* LALt():          implementation of -Lambda id Lambda^t for -Delta u,      */

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

Quadrature

In practice, quadratures are used for integration on elements.

In ALBERTA, all quadratures are defined in terms of the reference element $\widehat{S}$. In particular, a quadrature $\hat{Q}$ on $\widehat{S}$ is given by a set $\{(w_k, \lambda_k) \in \mathbb{R} \times \mathbb{R}^{d+1} : k = 1, \ldots, n_Q\}$ of weights w_k and quadrature points $\lambda_k \in \bar{S}$ such that

$$\int_{\widehat{S}} f(\hat{x}) \, \mathrm{d}\hat{x} \approx \hat{Q}(f) := \sum_{k=1}^{n_Q} w_k \, f(\hat{x}(\lambda_k))$$

ALBERTA provides numerical quadrature formulas that are exact up to order 19 ($d = 1$), 17 ($d = 2$) and 7 ($d = 3$), respectively.

Quadrature

Hence, in practice we have

$$\mathcal{A}_{ij} = \hat{Q}_2 \left([\nabla_\lambda \bar{\varphi}_{i_S(i)} \cdot \bar{A} \nabla_\lambda \bar{\varphi}_{i_S(j)}] \circ \lambda \right),$$

and it is sufficient (and necessary) to provide the values of $\bar{A}$ at all the quadratures points λ_k on S.

In our example, the definition of $\bar{A}$ does not depend on the quadrature point `int iq`.

```
/* LALt():          implementation of -Lambda id Lambda^t for -Delta u,      */

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

Matrix assembly

Now we need to give ALBERTA information on the matrix $\mathcal{A}$ that we want to assemble. Note the reference to `LALt()` previously defined, and the definition of the quadrature rule `quad[2]` for the second order term `LALt` of Lu.

```
  quad = get_quadrature(DIM, 2*u_h->fe_space->bas_fcts->degree);

/*--------------------------------------------------------------------------*/
/*  init functions for matrix assembling                                    */
/*--------------------------------------------------------------------------*/
    OPERATOR_INFO           o_info2 = {nil}, o_info0 = {nil};
    const EL_MATRIX_INFO    *matrix_info;

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

    o_info2.row_fe_space = o_info2.col_fe_space = u_h->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;
  }
```

Where OPERATOR_INFO is a structure with the following members.

```
/*--------------------------------------------------------------------*/
/*---  data structure about the differential operator for matrix assemblage */
/*--------------------------------------------------------------------*/
typedef struct operator_info  OPERATOR_INFO;
struct operator_info
{
  const FE_SPACE *row_fe_space;
  const FE_SPACE *col_fe_space;
  const QUAD *quad[3];

  void        (*init_element)(const EL_INFO *, const QUAD *[3], void *);
  const REAL (*(*LALt)(const EL_INFO *, const QUAD *, int, void *))[DIM+1];
  int         LALt_pw_const;
  int         LALt_symmetric;
  const REAL *(*Lb0)(const EL_INFO *, const QUAD *, int, void *);
  int         Lb0_pw_const;
  const REAL *(*Lb1)(const EL_INFO *, const QUAD *, int, void *);
  int         Lb1_pw_const;
  int         Lb0_Lb1_anti_symmetric;
  REAL        (*c)(const EL_INFO *, const QUAD *, int, void *);
  int         c_pw_const;

  int         use_get_bound;
  void        *user_data;
  FLAGS       fill_flag;
};
```

Assembly

```
/*--------------------------------------------------------------------------*/
/*  and now assemble the matrix and right hand side                         */
/*--------------------------------------------------------------------------*/
  clear_dof_matrix(matrix); dof_set(0.0, fh);

  fill_flag = CALL_LEAF_EL|FILL_COORDS|FILL_BOUND;
  el_info = traverse_first(stack, u_h->fe_space->mesh, -1, fill_flag);
  while (el_info) {
    const REAL   *u_old_loc = (*get_u_loc)(el_info->el, u_old, nil);
    const DOF    *dof       = (*get_dof)(el_info->el, admin, nil);
    const S_CHAR *bound     = (*get_bound)(el_info, nil);
/*--------------------------------------------------------------------------*/
/* initialization of values used by LALt and c                              */
/*--------------------------------------------------------------------------*/
    op_info->det    = el_grd_lambda(el_info, op_info->Lambda);

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

    add_element_matrix(matrix, 1.0, n, n, dof, dof, a_mat, bound);
    add_element_matrix(matrix, 1.0, n, n, dof, dof, c_mat, bound);
/*--------------------------------------------------------------------------*/
/*  f += 1/tau*m(u_old,psi_i)                                               */
/*--------------------------------------------------------------------------*/
    for (i = 0; i < n; i++) {
      if (bound[i] < DIRICHLET) {
        REAL val = 0.0;
        for (j = 0; j < n; j++) val += c_mat[i][j]*u_old_loc[j];
        fh->vec[dof[i]] += val;
      }
    }
    el_info = traverse_next(stack, el_info);
  }
  free_traverse_stack(stack);

  L2scp_fct_bas(f, quad, fh);           // add (f(,t), psi_i) to fh, i=1,...,N
  dirichlet_bound(g, fh, u_h, nil);     // set Dirichlet vertices of u_h
```

Mass matrix

Here we also assembled the stiffness matrix $\frac{1}{\tau_n}\mathcal{M}$, that is defined similarly to $\mathcal{A}$. Note that now we only need a quadrature for the zero order term c of Lu.

```
/* c():              implementation of 1/tau*m(,)                    */

static REAL c(const EL_INFO *el_info, const QUAD *quad, int iq, void *ud) {
  struct op_info *info = ud;
  return(info->tau_1*info->det);
}

// <snip>

    OPERATOR_INFO           o_info2 = {nil}, o_info0 = {nil};
    const EL_MATRIX_INFO    *matrix_info;

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

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

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

Solving the system

$$(\frac{1}{\tau_n}\mathcal{M} + \mathcal{A})\,U_n = \frac{1}{\tau_n}\mathcal{M}\,U_{n-1} + F_n =: \hat{F}_n$$

Once these matrices are assembled, we can let ALBERTA solve that system.

```
/*--------------------------------------------------------------------------*/
/*  solve(): solve the linear system, called by adapt_method_stat()         */
/*--------------------------------------------------------------------------*/
static void solve(MESH *mesh) {
  FUNCNAME("solve");
  static REAL        tol = 1.e-8;
  static int         max_iter = 1000, info = 2, icon = 1, restart = 0;
  static OEM_SOLVER solver = NoSolver;

  if (solver == NoSolver) {
    tol = 1.e-8;
    GET_PARAMETER(1, "solver", "%d", &solver);
    GET_PARAMETER(1, "solver tolerance", "%f", &tol);
    GET_PARAMETER(1, "solver precon", "%d", &icon);
    GET_PARAMETER(1, "solver max iteration", "%d", &max_iter);
    GET_PARAMETER(1, "solver info", "%d", &info);
    if (solver == GMRes) GET_PARAMETER(1, "solver restart", "%d", &restart);
  }
  oem_solve_s(matrix, f_h, u_h, solver, tol, icon, restart, max_iter, info);
  return;
}
```

Recap

```
static DOF_REAL_VEC    *u_h = nil;
static DOF_REAL_VEC    *u_old = nil;
static DOF_REAL_VEC    *f_h = nil;
static DOF_MATRIX      *matrix = nil;
```

So far we have described how, given a certain mesh, we can have ALBERTA

- assemble the system's stiffness matrix `'matrix'`
- assemble the right hand side vector `'f_h'`
- solve the resulting (linear) system `'matrix u_h = f_h'`

Next we need to know how to obtain adapted meshes and how to drive their refinement.

Mesh refinement

In ALBERTA a triangulation is a hierarchical mesh, where the *root* elements are given by a **macro triangulation** and the *leaf* elements make up the actual mesh.

Starting from a macro triangulation, we can uniformly refine the mesh a certain number of times.

```
/*--------------------------------------------------------------------*/
/*  get a mesh, and read the macro triangulation from file            */
/*--------------------------------------------------------------------*/
  GET_PARAMETER(1, "macro file name", "%s", filename);
  GET_PARAMETER(1, "global refinements", "%d", &n_refine);

  mesh = GET_MESH("ALBERTA mesh", init_dof_admin, init_leaf_data);
  read_macro(mesh, filename, nil);
  global_refine(mesh, n_refine*DIM);
```

Here, `global_refine(mesh, mark)` sets all leaf element markers and then calls `refine(mesh)` which results in each element to be refined `mark` times.

Mesh refinement

In general, the refinement of a mesh is done by the function `refine(mesh)`.

It refines all leaf elements of `mesh` with a positive element marker `mark` times. The routine loops over all leaf elements and refines elements with a positive marker until there is no element left with a positive marker.

Similarly, the coarsening of a mesh is done by the function `coarsen(mesh)`.

It tries to coarsen all leaf elements with a negative element marker `-mark` times. Note that coarsening of an element will only be performed, if no hanging nodes are created. I.e. all the involved elements must be marked for coarsening and they must be of finest level locally.

It suffices to set `mark` on each element in order to drive the mesh refinement.

Marking routines

ALBERTA offers a standard marking routine `marking()`, that will be used whenever no user defined marking routine is provided.

Assume we are given an a posteriori error estimate

$$\|u - u_h\| \leq \eta(u_h) = \left(\sum_{S \in \mathcal{S}} \eta_S(u_h)^p \right)^{\frac{1}{p}}.$$

Then the standard marking routine offers four different marking strategies, in order to achieve $\eta(u_h) \leq tol$ with repeated refinement of $\mathcal{S}$, solution of u_h on $\mathcal{S}$, marking of elements etc.

1: Global refinement: mesh is refined globally, no coarsening is performed.

2: Maximum strategy: refine S with $\eta_S > \gamma \max_{S' \in \mathcal{S}} \eta_{S'}$, $\quad \gamma \in (0,1)$.

3: Equidistribution strategy: refine S with $\eta_S > \theta \frac{tol}{|\mathcal{S}|^{\frac{1}{p}}}$, $\quad \theta \in (0,1)$, $\theta \approx 1$.

4: Guaranteed error reduction strategy: See Dörfler (96) and Morin, Nochetto, Siebert (00).

Marking routines – coarsening

Given error estimate

$$\|u - u_h\| \leq \eta(u_h) = \left(\sum_{S \in \mathcal{S}} \eta_S(u_h)^p \right)^{\frac{1}{p}}.$$

Furthermore, assume that we have an a posteriori estimate $\eta_{c,S}$ for the additional error introduced by the coarsening of element S. Then the following coarsening strategies are available.

1: Global refinement: no coarsening is performed.

2: Maximum strategy: try to coarsen S with $\eta_S^p + \eta_{c,S}^p \leq \gamma_c \max_{S' \in \mathcal{S}} \eta_{S'}^p$, $\gamma_c \in (0, \gamma)$.

3: Equidistribution strategy: try to coarsen S with $\eta_S + \eta_{c,S} \leq \theta_c \frac{tol}{|\mathcal{S}|^{\frac{1}{p}}}$, $\theta_c \in (0, \theta)$.

4: Guaranteed error reduction strategy: See Dörfler (96) and Morin, Nochetto, Siebert (00).

Estimators – Elliptic Problems

ALBERTA provides residual type error estimators for quasi–linear elliptic problems of the type

$$\begin{aligned} -\nabla \,.\, A\,\nabla u(x) + r(x, u(x), \nabla u(x)) &= 0 \qquad \text{in } \Omega, \\ u(x) &= 0 \qquad \text{on } \Gamma_D, \\ \nu \,.\, A\,\nabla u(x) &= 0 \qquad \text{on } \Gamma_N. \end{aligned}$$

They are of the form

$$\begin{aligned} \|u - u_h\|^2_{H^1(\Omega)} &\leq \eta^1(u_h, \nabla u_h, A, r), \\ \|u - u_h\|^2_{L^2(\Omega)} &\leq \eta^0(u_h, \nabla u_h, A, r), \end{aligned}$$

see Verfürth (94) and Bänsch & Siebert (95).

They are implemented in the ALBERTA function `ellipt_est()`.

Estimators – Parabolic Problems

ALBERTA also provides error estimators for quasi–linear parabolic problems of the type

$$\begin{aligned} u_t - \nabla \,.\, A\,\nabla u(x,t) + r(x,t,u(x,t),\nabla u(x,t)) &= 0 && \text{in } \Omega_T, \\ u(x,t) &= 0 && \text{on } \Gamma_D, \\ \nu \,.\, A\,\nabla u(x,t) &= 0 && \text{on } \Gamma_N, \\ u(x,0) &= u_0 && \text{in } \Omega. \end{aligned}$$

They are of the form

$$\|u - u_h\|^2_{L^2(\Omega_T)} \le \eta_0 + \eta_h + \eta_c + \eta_\tau,$$

and implemented in `heat_est()`, where the right hand side is made up of

- terms estimating the initial error,
- terms estimating the error from discretization in space,
- terms estimating the error from mesh coarsening between time steps,
- terms estimating the error from discretization in time.

Estimator in `heat.c`

```c
static REAL r(const EL_INFO *el_info, const QUAD *quad, int iq, REAL t,
              REAL uh_iq, const REAL_D grd_uh_iq) {
  REAL_D      x;
  coord_to_world(el_info, quad->lambda[iq], x);
  eval_time_f = t;
  return(-f(x));
}

static REAL estimate(MESH *mesh, ADAPT_STAT *adapt) {
  FUNCNAME("estimate");
  static int     degree;
  static REAL    C[4] = {1.0, 1.0, 1.0, 1.0};
  REAL_DD        A = {{0.0}};
  FLAGS          r_flag = 0;  /* = (INIT_UH|INIT_GRD_UH), if needed by r()  */
  int            n;
  REAL           space_est;

  for (n = 0; n < DIM_OF_WORLD; n++)
    A[n][n] = 1.0;   /* set diagonal of A; all other elements are zero      */

  eval_time_u = adapt_instat->time;
  degree = 2*u_h->fe_space->bas_fcts->degree;
  time_est = heat_est(u_h, adapt_instat, rw_el_est, rw_el_estc,
                      degree, C, u_old, (const REAL_D *) A, r, r_flag);

  space_est = adapt_instat->adapt_space->err_sum;
  err_L2 = L2_err(u, u_h, nil, 0, nil, nil);

  INFO(adapt_instat->info, 2)("time = %.4le with timestep = %.4le\n",
                              adapt_instat->time, adapt_instat->timestep);
  INFO(adapt_instat->info, 2)("estimate   = %.4le, max = %.4le\n", space_est,
                              sqrt(adapt_instat->adapt_space->err_max));
  INFO(adapt_instat->info, 2)("||u-uh||L2 = %.4le, ratio = %.2lf\n", err_L2,
                              err_L2/MAX(space_est,1.e-20));

  return(adapt_instat->adapt_space->err_sum);
}
```

Here the mesh is adapted to the solution in every time step using a posteriori error estimators or indicators.

Several strategies are possible and provided within ALBERTA:

- Explicit strategy: The current time step is solved once on the mesh from the previous time step. Based on error estimates for this solution, the mesh is adapted and will be used for the next time step.
- Semi–implicit strategy: Same as explicit strategy, except that solution for current time step is solved on the newly adapted mesh, before moving to next time step.
- Implicit strategy: Starting from the previous time step's mesh, a mesh is generated based on error estimates for a solution of the current time step on the current mesh. This solve–estimate–adapt process is iterated until a certain error tolerance is reached.

These spatial strategies can be combined with an adaptive time stepping strategy to give a time and space adaptive algorithm.

Adaptive method in `heat.c`

```
/*--------------------------------------------------------------*/
/*  init adapt structure from "INIT/heat.dat" and start adaptive method     */
/*--------------------------------------------------------------*/
  adapt_instat = get_adapt_instat("heat", "adapt", 2, adapt_instat);

// <snip>

  eval_time_u0 = adapt_instat->start_time;

  adapt_instat->adapt_initial->get_el_est = get_el_est;
  adapt_instat->adapt_initial->estimate = est_initial;   // true error for u_0
  adapt_instat->adapt_initial->solve = interpol_u0;      // solve for u_0

  adapt_instat->adapt_space->get_el_est   = get_el_est;
  adapt_instat->adapt_space->get_el_estc  = get_el_estc;
  adapt_instat->adapt_space->estimate = estimate;          // estimate()
  adapt_instat->adapt_space->build_after_coarsen = build; // calls assemble()
  adapt_instat->adapt_space->solve = solve;                // solve()

  adapt_instat->init_timestep  = init_timestep;          // set u_old = u_h
  adapt_instat->set_time       = set_time;
  adapt_instat->get_time_est   = get_time_est;
  adapt_instat->close_timestep = close_timestep;         // write u_h to file,
                                                         // graphics etc

  adapt_method_instat(mesh, adapt_instat);            // start adaptive method
```

Parameters read from `INIT/heat.dat`

```
adapt->start_time:                0.0
adapt->end_time:                  0.1

adapt->tolerance:                 1.0e-4
adapt->timestep:                  1.0e-2
adapt->rel_initial_error:         0.5
adapt->rel_space_error:           0.5
adapt->rel_time_error:            0.5
adapt->strategy:                  1    % 0=explicit, 1=implicit
adapt->max_iteration:             10
adapt->info:                      2

adapt->initial->strategy:         2     % 0=none, 1=GR, 2=MS, 3=ES, 4=GERS
adapt->initial->MS_gamma:         0.5
adapt->initial->max_iteration:    10
adapt->initial->info:             2

adapt->space->strategy:           3     % 0=none, 1=GR, 2=MS, 3=ES, 4=GERS
adapt->space->ES_theta:           0.9
adapt->space->ES_theta_c:         0.2
adapt->space->max_iteration:      10
adapt->space->coarsen_allowed:    1    % 0|1
adapt->space->info:               2
```

Adaptive strategies

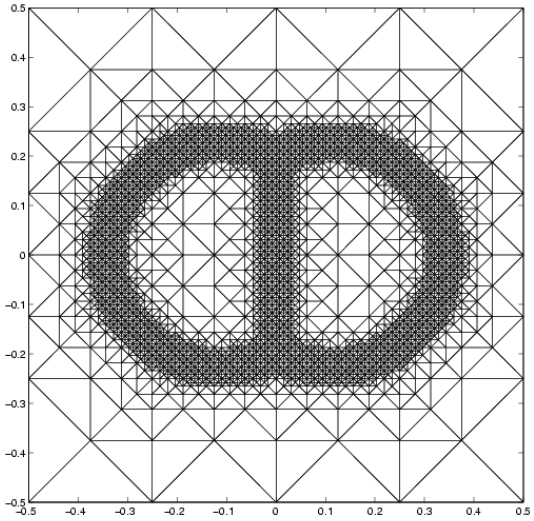
By default, ALBERTA provides her own marking routines and error estimators.

But there is great flexibility.

Often it can be desirable and necessary to implement one's own a posteriori error estimators. This can easily be done within the ALBERTA frame work.

Secondly, in situations where no estimators are available, user–defined marking routines can be implemented.

An example are phase field equations, where the refinement should be done in the interfacial region only. Here the marking is driven by the order parameter, without regard to error estimators.



... picking up the loose ends ...

Within our example `heat.c`, we are using e.g. piece wise linear elements.

```
static const FE_SPACE *fe_space;          /* initialized by init_dof_admin() */

/*--------------------------------------------------------------------------*/
/* init_dof_admin(): init DOFs for Lagrange elements of order               */
/*                   <polynomial degree>, called by GET_MESH()              */
/*--------------------------------------------------------------------------*/

static void init_dof_admin(MESH *mesh) {
  FUNCNAME("init_dof_admin");
  int             degree = 1;
  const BAS_FCTS  *lagrange;

  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;
}
```

where within `main()`

```
  mesh = GET_MESH("ALBERTA mesh", init_dof_admin, init_leaf_data);

  matrix = get_dof_matrix("A", fe_space);
  f_h    = get_dof_real_vec("f_h", fe_space);
  u_h    = get_dof_real_vec("u_h", fe_space);
```

Recall adaptive method in `heat.c`

```
/*--------------------------------------------------------------------*/
/*  init adapt structure from "INIT/heat.dat" and start adaptive method     */
/*--------------------------------------------------------------------*/
  adapt_instat = get_adapt_instat("heat", "adapt", 2, adapt_instat);

// <snip>

  eval_time_u0 = adapt_instat->start_time;

  adapt_instat->adapt_initial->get_el_est = get_el_est;
  adapt_instat->adapt_initial->estimate = est_initial;   // true error for u_0
  adapt_instat->adapt_initial->solve = interpol_u0;      // solve for u_0

  adapt_instat->adapt_space->get_el_est   = get_el_est;
  adapt_instat->adapt_space->get_el_estc  = get_el_estc;
  adapt_instat->adapt_space->estimate = estimate;          // estimate()
  adapt_instat->adapt_space->build_after_coarsen = build; // calls assemble()
  adapt_instat->adapt_space->solve = solve;                // solve()

  adapt_instat->init_timestep  = init_timestep;          // set u_old = u_h
  adapt_instat->set_time       = set_time;
  adapt_instat->get_time_est   = get_time_est;
  adapt_instat->close_timestep = close_timestep;         // write u_h to file,
                                                         // graphics etc

  adapt_method_instat(mesh, adapt_instat);            // start adaptive method
```

... where ...

```
/*---8<-------------------------------------------------------------*/
/*---   interpolation is solve on the initial grid                ---*/
/*------------------------------------------------------------>8---*/
static void interpol_u0(MESH *mesh) {
  dof_compress(mesh);
  interpol(u0, u_h);
  return;
}

static REAL est_initial(MESH *mesh, ADAPT_STAT *adapt) {
  err_L2 = adapt->err_sum = L2_err(u0, u_h, nil, 0, rw_el_est, &adapt->err_max)
  return(adapt->err_sum);
}
```

... and ...

```
static void init_timestep(MESH *mesh, ADAPT_INSTAT *adapt) {
  FUNCNAME("init_timestep");
  INFO(adapt_instat->info,1)
    ("---8<-------------------------------------------\n");
  INFO(adapt_instat->info, 1)("starting new timestep\n");
  dof_copy(u_h, u_old);
  return;
}

static void set_time(MESH *mesh, ADAPT_INSTAT *adapt) {
  FUNCNAME("set_time");
  INFO(adapt->info,1)
    ("---8<-------------------------------------------\n");
  if (adapt->time == adapt->start_time) {
    INFO(adapt->info, 1)("start time: %.4le\n", adapt->time);
  }
  else {
    INFO(adapt->info, 1)("timestep for (%.4le %.4le), tau = %.4le\n",
                         adapt->time-adapt->timestep, adapt->time,
                         adapt->timestep);
  }
  eval_time_f = adapt->time;
  eval_time_g = adapt->time;
  return;
}
```

... and ...

```
static void close_timestep(MESH *mesh, ADAPT_INSTAT *adapt) {
  FUNCNAME("close_timestep");
  static REAL err_max = 0.0;                          /* max space-time error    */
  static REAL est_max = 0.0;                          /* max space-time estimate */
  static int  write_fe_data = 0, write_stat_data = 0;
  static int  step = 0;
  static char path[256] = "./";
  REAL        space_est = adapt->adapt_space->err_sum;
  REAL        tolerance = adapt->rel_time_error*adapt->tolerance;

  err_max = MAX(err_max, err_L2);
  est_max = MAX(est_max, space_est + time_est);

  if (adapt->time == adapt->start_time) {
    tolerance = adapt->adapt_initial->tolerance;
    INFO(adapt->info,1)("start time: %.4le\n", adapt->time);
  }
  else {
    tolerance += adapt->adapt_space->tolerance;
    INFO(adapt->info,1)("timestep for (%.4le %.4le), tau = %.4le\n",
                        adapt->time-adapt->timestep, adapt->time,
                        adapt->timestep);
  }
  INFO(adapt->info,2)("max. est.  = %.4le, tolerance = %.4le\n",
                      est_max, tolerance);
  INFO(adapt->info,2)("max. error = %.4le, ratio = %.2lf\n",
                      err_max, err_max/MAX(est_max,1.0e-20));
```

```
  if (!step) {
    GET_PARAMETER(1, "write finite element data", "%d", &write_fe_data);
    GET_PARAMETER(1, "write statistical data", "%d", &write_stat_data);
    GET_PARAMETER(1, "data path", "%s", path);
  }

/*---8<---------------------------------------------------------------*/
/*---   write mesh and discrete solution to file for post-processing  ---*/
/*---------------------------------------------------------------->8---*/
  if (write_fe_data) {
    const char *fn;
    fn= generate_filename(path, "mesh", step);
    write_mesh_xdr(mesh, fn, adapt->time);
    fn= generate_filename(path, "u_h", step);
    write_dof_real_vec(u_h, fn);
  }
  step++;

/*---8<---------------------------------------------------------------*/
/*---  write data about estimate, error, time step size, etc.        ---*/
/*---------------------------------------------------------------->8---*/
  if (write_stat_data) {
    int n_dof = fe_space->admin->size_used;
    write_statistics(path, adapt, n_dof, space_est, time_est, err_L2);
  }
  graphics(mesh, u_h, get_el_est);
  return;
}
```

... and finally ...

```
/*--------------------------------------------------------------------*/
/*  build(): assemblage of the linear system: matrix, load vector,    */
/*           boundary values, called by adapt_method_stat()           */
/*           on the first call initialize u_h, f_h, matrix and information   */
/*           for assembling the system matrix                         */
/*--------------------------------------------------------------------*/

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

  dof_compress(mesh);
  INFO(adapt_instat->adapt_space->info, 2)
    ("%d DOFs for %s\n", fe_space->admin->size_used, fe_space->name);

  assemble(u_old, matrix, f_h, u_h, adapt_instat->timestep, f, g);
  return;
}
```

Then the whole process is automatically directed by

```
  adapt_method_instat(mesh, adapt_instat);          // start adaptive method
```

Macro triangulation

The initial triangulation of the domain Ω is given in a macro triangulation file. In our case, this file looks as follows.

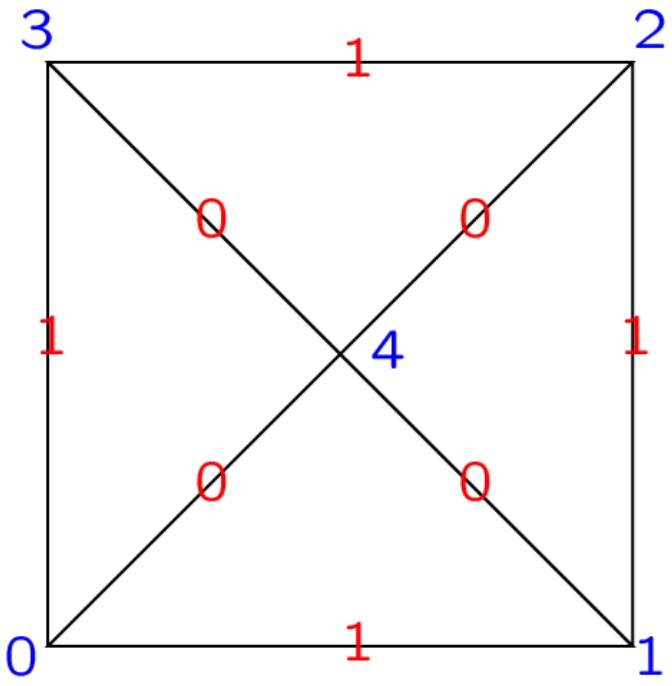
```
DIM: 2
DIM_OF_WORLD: 2

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
```



Different boundary conditions

ALBERTA distinguishes between vertices that are considered in the interior of the domain, are of Neumann boundary type, or of Dirichlet boundary type.

To this end, the following three macros are defined.

```
/*--------------------------------------------------------------------------*/
/*  boundary types                                                          */
/*--------------------------------------------------------------------------*/

#define INTERIOR        0
#define DIRICHLET       1
#define NEUMANN         -1
```

This can be used inside the macro triangulation file in order to specify e.g. Neumann boundary conditions.

More generally, any positive boundary type is considered Dirichlet, while any negative type is Neumann.

Remark: When solving `A u_h = f_h`, then the Dirichlet vertices of `u_h` must already have their correct value, and `f_h = u_h` at these vertices. See `heat.c` for an example.

Curved boundaries

Different parts of the boundary can be identified by different positive/negative constants. Depending on these constants, different boundary parameterizations can be invoked.

```
/*--------------------------------------------------------------------*/
/*  boundary descriptor                                               */
/*--------------------------------------------------------------------*/
/*  param_bound: pointer to a function to calculate a new point of a  */
/*               curved boundary                                      */
/*  type:        one of INTERIOR : DIRICHLET : NEUMANN                */
/*--------------------------------------------------------------------*/
typedef struct boundary          BOUNDARY;

struct boundary {
  void      (*param_bound)(REAL_D  );
  S_CHAR    bound;
};
```

When reading the macro triangulation, a function that assigns these different constants to different projections can be given. In this example this is `my_ibdry()`.

```
  mesh = GET_MESH("ALBERTA mesh", init_dof_admin, init_leaf_data);
  read_macro(mesh, filename, my_ibdry);
```

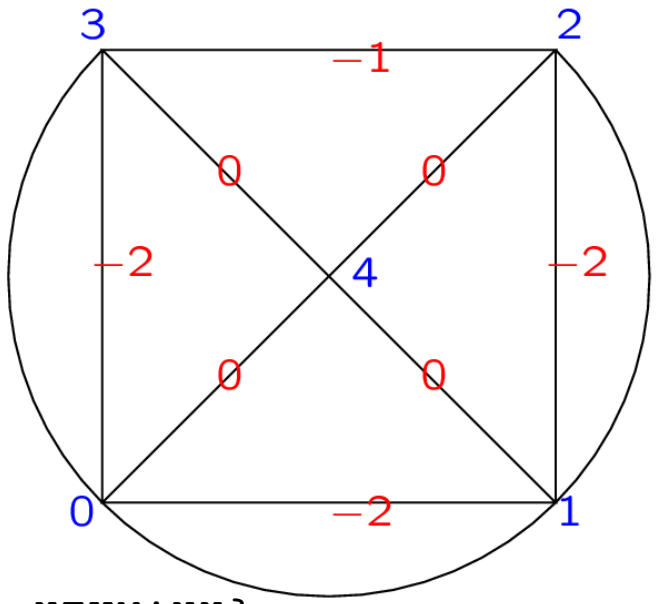
Parametric elements

In my_ibdry(), we decide which part of the boundary is parametric and which is not.

```
static void ball_project(REAL_D p) {
  FUNCNAME("ball_project");
  REAL norm = 0.0;
  int k;
  static REAL_D centre = {0.5, 0.5};
  norm = DIST_DOW(p, centre);
  norm = sqrt(0.5) / MAX(1e-10, sqrt(norm));
  for (k=0; k<DIM_OF_WORLD; k++) p[k] *= norm;
}

const BOUNDARY *my_ibdry(MESH *mesh, int bound) {
  FUNCNAME("my_ibdry");
  static const BOUNDARY ball_neumann = {ball_project, NEUMANN};
  static const BOUNDARY straight_neumann = {nil, NEUMANN};

  switch (bound) {
    case -2 : return &ball_neumann;
    case -1 : return &straight_neumann;
    default: ERROR_EXIT("no boundary type %d\n", bound);
  }
  return &ball_neumann;                    // so that compiler does not complain
}
```



Spatially dependent coefficients

Recall that in our example for the heat equation we had $Lu = -\Delta u$ and hence $A(x) \equiv I$ and $\bar{A}(\lambda) := |\det DF_S(\hat{x}(\lambda))|\, \Lambda(x(\lambda))\, \Lambda^T(x(\lambda))$.

In the more general case that $Lu = -\nabla . (b(x)\nabla u(x))$, we have to change the definition of `LALt()` to something like the following.

```
static const REAL (*LALt(const EL_INFO *el_info, const QUAD *quad,
                         int iq, void *ud))[DIM+1] {
  struct op_info *info = ud;
  int             i, j, k;
  static REAL     LALt[DIM+1][DIM+1], bx;
  REAL_D x;

  coord_to_world(el_info, quad->lambda[iq], x);
  bx = b(x);

  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] *= info->det * bx;
      LALt[j][i] = LALt[i][j];
    }
  return((const REAL (*)[DIM+1]) LALt);
}
```

Also note that `o_info2.LALt_pw_const = false;` should be set.

Solution dependent coefficients

Often it can be necessary to evaluate finite element functions *locally* on an element. An example is the assembling of the system matrix for a second order operator of e.g. the form $Lu = -\nabla . (b(u)\nabla u(x))$.

Then the ALBERTA functions

`get_real_vec` and `get_uh_at_qp`

have to be used.

They evaluate a given finite element function `u_h` at the local degrees of freedom and at the local quadrature points for a given quadrature, respectively.

An example can be seen on the next slides.

Solution dependent coefficients

First we need to adapt the definition of the structure op_info to also include the necessary values of u_h on a given element.

```
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 get_uh_at_qp                 */
  const REAL      *u_qp;       /*  u at all quadrature nodes of quad_fast     */
};
```

quad_fast has to be filled once before the assembly starts, and u_qp needs to be filled on each element before adding its contribution to the system matrix.

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

```
  op_info->u_qp  = uh_at_qp(op_info->quad_fast, u_old_loc, nil);
```

The assembly routine then looks as follows ...

Solution dependent coefficients

```
/*--------------------------------------------------------------------*/
/*  init functions for matrix assembling                              */
/*--------------------------------------------------------------------*/
    OPERATOR_INFO           o_info2 = {nil}, o_info0 = {nil};
    const EL_MATRIX_INFO    *matrix_info;
    const BAS_FCTS          *bas_fcts = u_h->fe_space->bas_fcts;

    if (!op_info) op_info = MEM_ALLOC(1, struct op_info);
    op_info->quad_fast = get_quad_fast(bas_fcts, quad, INIT_PHI);
    get_u_loc  = bas_fcts->get_real_vec;

// <snip>
  while (el_info) {
    const REAL   *u_old_loc = (*get_u_loc)(el_info->el, u_old, nil);
    const DOF    *dof       = (*get_dof)(el_info->el, admin, nil);
    const S_CHAR *bound     = (*get_bound)(el_info, nil);
/*--------------------------------------------------------------------*/
/* initialization of values used by LALt and c                        */
/*--------------------------------------------------------------------*/
    op_info->det   = el_grd_lambda(el_info, op_info->Lambda);
    op_info->u_qp  = uh_at_qp(op_info->quad_fast, u_old_loc, nil);

    a_mat = fill_a(el_info, a_info);
    c_mat = fill_c(el_info, c_info);
    add_element_matrix(matrix, 1.0, n, n, dof, dof, a_mat, bound);
    add_element_matrix(matrix, 1.0, n, n, dof, dof, c_mat, bound);
// <snip>
  }
```

Solution dependent coefficients

In order to assemble the contribution of $Lu = -\nabla.(b(u)\nabla u)$, we have to change the definition of `LALt()` to something like the following.

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

  bu = b(info->u_qp[iq]);

  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] *= info->det * bu;
      LALt[j][i] = LALt[i][j];
    }
  return((const REAL (*)[DIM+1]) LALt);
}
```

Also note that `o_info2.LALt_pw_const = false;` should be set.

From 2d to 3d

In ALBERTA most problems can be implement dimension independent. That means that a single source file yields numerical implementations in 1d, 2d and 3d.

Hence the user should try to write the program as general as possible. E.g.

- Use `DIM+1` for number of vertices of an element.
- Use `el_volume()` for length/area/volume of an element.
- Use `REAL_D, REAL_DD` for vectors in $\mathbb{R}^d$ and matrices in $\mathbb{R}^{d\times d}$, respectively.
- Use `N_NEIGH` for number of neighbours of an element.
- etc.

Multigrid

Apart from SOR and Krylov type solvers for general linear systems of the form

$$\texttt{matrix u\_h = f\_h},$$

that are implemented within `sor_s()`, `ssor_s()` and `oem_solve_s()`, ALBERTA also offers the possibility to solve the linear system with a multi grid solver.

The multi grid solver can be invoked by the ALBERTA function `mg_s()`.

```
mg_s(matrix, u_h, f_h, bound, tol, max_iter, info, prefix);
```

The internal structure `MULTI_GRID_INFO` can be used to adapt the implemented algorithm for degenerate or nonlinear problems.

Nonlinear solves

For the solution of nonlinear equations of the form

$$F(\mathtt{u\_h}) = 0$$

several Newton methods are provided within ALBERTA. They can be invoked by the function calls `nls_newton()`, `nls_newton_ds()` and `nls_newton_fs()`.

For further details see the manual and the ALBERTA example `nonlin.c`.

Directory structure

Default structure for ALBERTA projects is:

```
./src/1d/
        /data
        /INIT
        /Macro
./src/2d/
        /data
        /INIT
        /Macro
./src/3d/
        /data
        /INIT
        /Macro
./src/Common/
```

Each `/src/[1-3]d/` directory contains a `Makefile` to compile the project for the given dimension.

Makefile

```
#!gmake
###############################################################################
#    Sample ALBERTA Makefile for 2d                                           #
###############################################################################
DEFAULT = myheat
prefix = /usr/local

# delete line, if the paths are environment variables
ALBERTA_INCLUDE_PATH = ${prefix}/include
ALBERTA_LIB_PATH = ${exec_prefix}/lib

# compile flags
# run make with "make DEBUG=1" to get debuggable code
ifeq ($(DEBUG),1)
CFLAGS = -O0 -ggdb -fno-inline -fno-builtin
else
CFLAGS = -O3
endif
FFLAGS = -O3

# 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 =

# 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                                                 #
###############################################################################
MYHEAT_OFILES = myheat.o graphics.o

myheat: $(MYHEAT_OFILES)
        $(LINK) $(MYHEAT_OFILES) $(LIBS)

clean: albertaclean
        -rm -f myheat

realclean: albertarealclean
new: albertanew
```

GRAPE

If GRAPE is available on the system, then during installation of ALBERTA the binaries `alberta_grape22` and `alberta_grape33` will be installed.

E.g. in `/usr/local/bin/`.

These programs can be called with a mesh and a (scalar, vector) data file for visualization purposes. See `alberta_grape22 --help`.

Example: `alberta_grape22 -m mesh000020 -bs u_h000020`

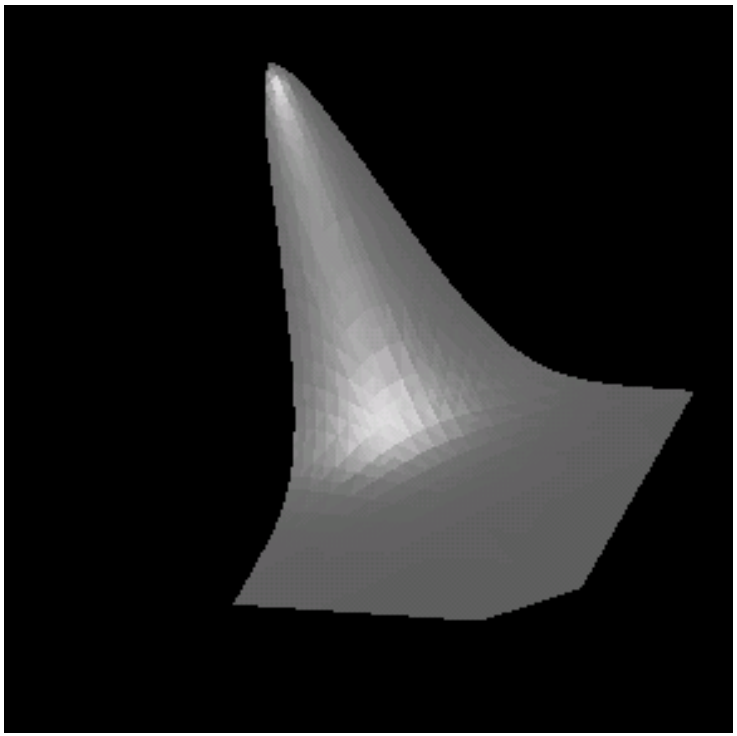
The usage of the GRAPE interface is not very intuitive. Here a quick guide on how to obtain a plot for the solution of the heat equation.

1. Select 'ALBERTA-Mesh'
2. Select Display-Method: 'function-graph'
3. Choose Options: Select `u_h` and choose 'graph height' 40
4. Select Transform: rotate x, y, z. (e.g. using the sphere)

In order to visualize the ALBERTA mesh, one can select as Display-Method 'inspect'.

GRAPE


light
surf
trans
manag
opts
exit
add project
del project
I/O
read st
write st
read gr
write gr
send method
display method
switches
I
E
local time
"u_h"
Scene
"ALBERTA mesh"
HMesh2d
mark
copy
hardcopy
expand
delete-one
delete-all
scene display method
scene dynamic method
scene-trans
scene-suprop
copy-trans
copy-suprop
main time
0.00000
play
<
>
steps
10
GRAPE
Version 5.4.2 Build 17
(C)1993-2004 SFB256 Grafiklabor Uni Bonn
?



Summary

- ALBERTA is an easy to use adaptive finite element toolbox (given knowledge of C).

- Example: `heat.c` $\approx$ 600 lines encode fully space and time adaptive approximation.

- Dimension independent coding provides program for 1d, 2d and 3d without any extra work.

- Visualization possible within X-window ($d < 3$), gltools and GRAPE.

- ALBERTA is very versatile and can be used, adapted and extended to cover any kind of PDE.

Practical

Possible Tasks:

1. Use GRAPE to visualize the solution of the heat equation on the unit square at time T.

2. Solve the heat equation on an L shaped domain.

3. Solve the heat equation on $\Omega = B_1(0) \setminus [0,1] \times [-1,0]$. Prescribe no-flux boundary conditions on $[(0,0),(1,0)] \cup [(0,0),(0,-1)]$ and Dirichlet conditions on the curved part of $\partial\Omega$.

4. Solve $u_t - \nabla . ((1 + |x|^2) \nabla u) = f$ on any of these domains.

5. Solve $u_t - \nabla . ((1 + |u|^2) \nabla u) = f$ on any of these domains.