Code sketch

A finite element code for linear elliptic problems can be sketched as follows:

1. Input data $\Omega, f, \ldots$
2. Build a mesh $\mathcal{M}$ of the domain Ω
3. Define the discrete space $\mathcal{S}$
4. Assemble marix M and right hand side b
5. Solve linear system $MU = b$
6. Visualize the computed solution

Finite element functions

Let $\Omega \subset \mathbb{R}^d$,
$\mathcal{M}$ be a mesh s.t. $\overline{\Omega} = \cup_{E \in \mathcal{M}} E$

Let $\mathcal{S}$ be the finite element space:

$$\mathcal{S} = \{v \in C^0(\overline{\Omega}) \mid v_{|E} \in \mathbb{P}_q(E),\ \forall E \in \mathcal{M}\}$$

Let $\{\varphi_1, \ldots, \varphi_N\}$ (global) basis of $\mathcal{S}$

Global representation:

$$v \in \mathcal{S} \qquad v(x) = \sum_{j=1}^{N} V_j \varphi_j(x) \qquad x \in \Omega$$

Let $E \in \mathcal{M}$, $K_d^b = \text{conv}(e_1, \ldots, e_{d+1})$

$$\begin{tikzcd} E \arrow[r, shift left, "\lambda_E"] & K_d^b \arrow[l, shift left, "x_E"] \end{tikzcd}$$

$\{\phi_1, \ldots, \phi_m\}$ (local) basis of $\mathbb{P}_q(K_d^b)$ identify
$\{\phi_1^E, \ldots, \phi_m^E\}$ (local) basis of $\mathbb{P}_q(E)$ s.t.

$$\phi_i^E = \phi_i \circ \lambda_E \qquad \text{on } \; E \qquad i = 1, \ldots m$$

Let $J_E = \{j \in \{1, \ldots N\} \mid \text{supp}(\varphi_j) \supset E\}$

$$\begin{tikzcd} J_E \arrow[r, shift left, "i_E"] & \{1, \ldots, m\} \arrow[l, shift left, "j_E"] \end{tikzcd}$$

$$\varphi_{j|E} (= \phi^E_{i_E(j)}) = \phi_{i_E(j)} \circ \lambda_E \qquad \forall j \in J_E$$

Local representation of $v \in \mathcal{S}$

in cartesian coordinates:

$$v(x) = \sum_{i=1}^{m} V_i^E \phi_i(\lambda_E(x)) \qquad x \in E$$

in barycentric coordinates

$$v(x_E(\lambda)) = \sum_{i=1}^{m} V_i^E \phi_i(\lambda) \qquad \lambda \in K_d^b$$

where

$$(V_1^E, \cdots, V_m^E) = (V_{j_E(1)}, \cdots, V_{j_E(m)})$$

Degrees of freedom give connection between local and global finite element functions.

$\mathcal{S}$ is defined by:

1. A mesh $\mathcal{M}$ of the domain

2. The set of local basis functions $\{\phi_1, \ldots, \phi_m\}$ of $\mathbb{P}_q(K_d^b)$ that identify a basis of $\mathbb{P}_q(E)$, $\forall E \in \mathcal{M}$ and thus a global basis $\{\varphi_1, \ldots \varphi_N\}$ of S

3. The degrees of freedom and the relation between local and global ones.

In Alberta all information about the underlying mesh, the local basis functions and the degrees of freedom are collected in the following data structure which defines one single finite element space:

```
typedef struct fe_space          FE_SPACE;

struct fe_space
{
  const char        *name;

  const DOF_ADMIN   *admin;
  const BAS_FCTS    *bas_fcts;
  MESH              *mesh;

};
```

1. Degrees of freedom

In ALBERTA every abstract degree of freedom DOFs is realized as an integer index into vectors

```
typedef signed int                  DOF;
```

These indices and the corresponding indicized vectors or matrices are amministrated via the DOF_ADMIN data structure and some routines.

For each set of DOFs one DOF_ADMIN structure is created. It contains all data about this set of DOFs. It includes linked lists of vectors and matrices of different data types indicized on DOFs. These vectors and matrices will be automatically resized during mesh changes.

```
typedef struct dof_admin        DOF_ADMIN;

struct dof_admin
{
  MESH          *mesh;
  const char    *name;
  ...

  DOF  size_used;
  ....

  DOF_INT_VEC     *dof_int_vec;
  DOF_DOF_VEC     *dof_dof_vec;
  DOF_DOF_VEC     *int_dof_vec;
  DOF_UCHAR_VEC   *dof_uchar_vec;
  DOF_SCHAR_VEC   *dof_schar_vec;
  DOF_REAL_VEC    *dof_real_vec;
  DOF_REAL_D_VEC  *dof_real_d_vec;
  DOF_MATRIX      *dof_matrix;
   ...
};
```

To create linked list of vectors indicized on DOFs, Alberta provides the following structures for vectors of different data types indicized on DOFs. For reals:

```
typedef struct dof_real_vec     DOF_REAL_VEC;

struct dof_real_vec
{
  DOF_REAL_VEC   *next;
  const FE_SPACE *fe_space;

  const char     *name;
  DOF            size;
  REAL           *vec;

  ...
};
```

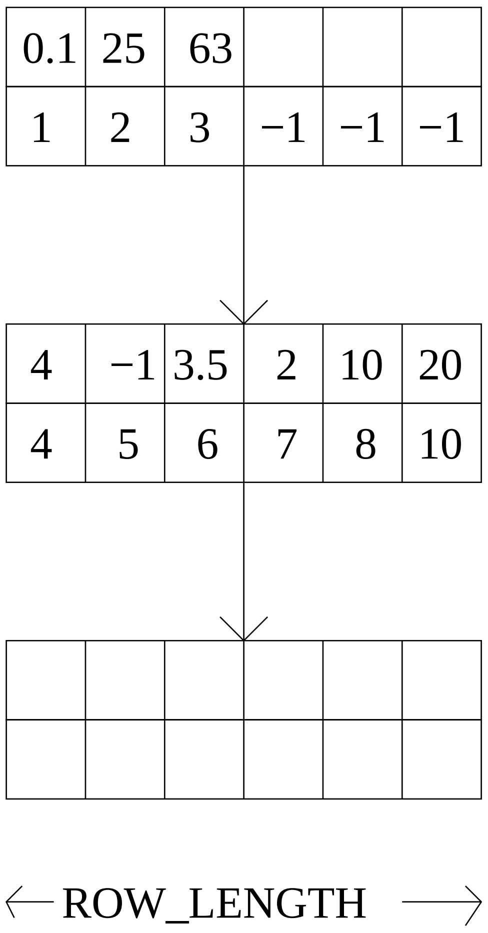
Similarly for other data types

```
typedef struct dof_int_vec      DOF_INT_VEC;
typedef struct dof_real_d_vec   DOF_REAL_D_VEC;
typedef struct dof_uchar_vec    DOF_UCHAR_VEC;
...
```

ALBERTA provides as well a data type for matrices indicized on `DOFs`

```
typedef struct dof_matrix        DOF_MATRIX;
```

these matrices are usually sparse and are stored in a way that reflects their sparseness:



Functions to handle vectors indicized on DOFs

To allocate memory for a new `DOF_?_VEC`:

```
DOF_INT_VEC    *get_dof_int_vec(const char *name,
                                 const FE_SPACE *);
DOF_REAL_VEC   *get_dof_real_vec(const char *name,
                                  const FE_SPACE *);
DOF_REAL_D_VEC *get_dof_real_d_vec(const char *name,
                                    const FE_SPACE *);
...
```

To fee memory used by a `DOF_?_VEC`:

```
void free_dof_int_vec(DOF_INT_VEC *vec);
void free_dof_real_vec(DOF_REAL_VEC *vec);
void free_dof_real_d_vec(DOF_REAL_D_VEC *vec);
...
```

To print all components of a `DOF_?_VEC`:

```
void print_dof_int_vec(const DOF_INT_VEC *div);
void print_dof_real_vec(const DOF_REAL_VEC *drv);
void print_dof_real_d_vec(const DOF_REAL_D_VEC *drdv);
...
```

To access the entry `vec` of a `DOF_?_VEC * dof_vec`:

```
GET_DOF_VEC(ptr, dof_vec)
```

it assign `dof_vec->vec` to `ptr` if `dof_vec` and `dof_vec->vec` are not `nil`

Functions to handle matrices indicized on DOFs

Similarly for matrices indicized on DOFs:

```
DOF_MATRIX  *get_dof_matrix(const char *name,
                            const FE_SPACE *);

void free_dof_matrix(DOF_MATRIX *mat);

void print_dof_matrix(const DOF_MATRIX *matrix);
```

Basic linear algebra routines for vectors and matrices indicized on DOFs

`REAL dof_nrm2(const DOF_REAL_VEC *x);`	$(\sum x_i^2)^{1/2}$
`REAL dof_asum(const DOF_REAL_VEC *x);`	$\sum \lvert x_i \rvert$
`REAL dof_min(const DOF_REAL_VEC *x);`	$\min x_i$
`REAL dof_max(const DOF_REAL_VEC *x);`	$\max x_i$
`void dof_set(REAL alpha, DOF_REAL_VEC *x);`	$x_i = \alpha$
`void dof_copy(const DOF_REAL_VEC *x, DOF_REAL_VEC *y);`	$y = x$
`void dof_scal(REAL alpha, DOF_RE_VEC *x);`	$x_i = \alpha x_i$
`REAL dof_dot(const DOF_REAL_VEC *x, const DOF_REAL_VEC *y);`	$\sum x_i y_i$

...

For loops over all `DOFs` the following macro is defined

FOR_ALL_DOFS (admin,todo)

Example:

```
DOF_REAL_VEC *drv;

FOR_ALL_DOFS(drv->fe_space->admin, drv->vec[dof]=alpha);
```

The `dof_set()` routine is written this way.

During `todo`, the local variable `int dof` holds the current index of the used entry; it must not be alterated by `todo`

2. Local basis functions

```
typedef struct bas_fcts         BAS_FCTS;

struct bas_fcts
{
  char             *name;
  int              n_bas_fcts;
  int              degree;
  int              n_dof[DIM+1];
  ...

  BAS_FCT          **phi;
  GRD_BAS_FCT      **grd_phi;
  D2_BAS_FCT       **D2_phi;

  const DOF        *(*get_dof_indices)(const EL *,
                      const DOF_ADMIN *,  DOF *);
  ...
  const REAL       *(*interpol)(const EL_INFO *,
                                int, const int *,
                   REAL (*)(const REAL_D), REAL *);
  ...
  void             *bas_fcts_data;

  ...
};
```

`interpol(el_info,0,nil,f,nil)`

returns a pointer to a vector containing the interpolation coefficient of `f` for all local basis functions on `el_info->el`. Such a function needs vertex coordinate information, all routines using this function need the `FILL_COORD` flag during mesh traversal.

phi: $\texttt{(*phi[i])(lambda)} \longrightarrow \phi_i(\lambda)$,
$0 \le i \le \texttt{n\_bas\_fcts}$

grd_phi: $\nabla_\lambda \phi_i(\lambda) = \left(\frac{\partial \phi_i}{\partial \lambda_0}(\lambda), \ldots, \frac{\partial \phi_i}{\partial \lambda_m}(\lambda)\right)$

$$\texttt{(*grd\_phi[i])(lambda)[k]} = \frac{\partial \phi_i}{\partial \lambda_k}(\lambda)$$

D2_phi:

$$D^2_\lambda \phi_i(\lambda) = \begin{pmatrix} \frac{\partial^2 \phi_i}{\partial^2 \lambda_0}(\lambda), & \ldots, & \frac{\partial^2 \phi_i}{\partial \lambda_0 \partial \lambda_m}(\lambda) \\ & & \\ \frac{\partial^2 \phi_i}{\partial \lambda_m \partial \lambda_0}(\lambda), & \ldots, & \frac{\partial^2 \phi_i}{\partial^2 \lambda_m}(\lambda) \end{pmatrix}$$

$$\texttt{(*D2\_phi[i])(lambda)[k][l]} = \frac{\partial^2 \phi_i}{\partial \lambda_k \partial \lambda_l}(\lambda)$$

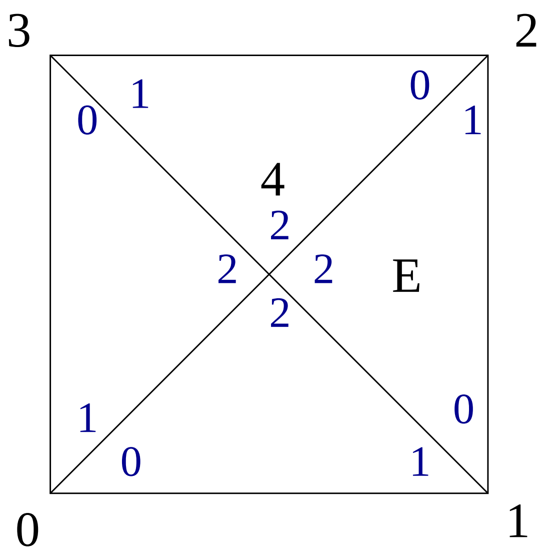
get_dof_indices: $j_E : \{1, \ldots m\} \longrightarrow J_E$

`get_dof_indices(el,admin,dof)` $\longrightarrow$ `dof[i]=`j_{el}`(i)`

`i=1,...,n_bas_fcts`

Example

$$q = 1, \qquad J_E = \{1, 2, 4\}$$



$$j_E : \{0,1,2\} \to J_E$$

$$\begin{array}{ccc} 0 & \longrightarrow & 1 \\ 1 & \longrightarrow & 2 \\ 2 & \longrightarrow & 4 \end{array}$$

`get_dof_indices(el,admin,dof)` $\longrightarrow$ `dof`$= [1, 2, 4]$

Lagrange finite elements up to order 4 are implemented in ALBERTA

$$\mathcal{S}^{q,0}(\mathcal{M}) = \{v \in C^0(\overline{\Omega}) \mid v_{|E} \in \mathbb{P}_q,\ \forall E \in \mathcal{M}\}\ 0 \le q \le 4$$

with $\{\varphi_i\}_{i=1}^N$ basis of $\mathcal{S}^{q,0}(\mathcal{M})$ s.t.

$$\varphi(x_j) = \delta_{ij} \quad i,j = 1,\dots N$$

where $\{x_i\}_{i=1}^N$ are Lagrange nodes

Lagrange finite elements are given by the basis functions $\{\phi_i\}_{i=1}^m$ of $\mathbb{P}_q(K_d^b)$ associated to the Lagrange nodes $\{\lambda^i\}_{i=1}^m$ given in baricentric coordinates:

$$\phi_k(\lambda^i) = \delta_{ki} \quad i,k = 1,\dots m$$

$$x_j = x_E(\lambda^{i_E(j)}) \quad \forall j \in J_E$$

Example:

Local basis functions for linear finite elements in 2d are:

$$\phi_0(\lambda) = \lambda_0, \quad \phi_1(\lambda) = \lambda_1, \quad \phi_2(\lambda) = \lambda_2$$

while Lagrange nodes in barycentric coordinates are

$$\lambda^0 = (1,0,0), \quad \lambda^1 = (0,1,0), \quad \lambda^2 = (0,0,1).$$

The library function `get_lagrange(degree);` returns a pointer to a filled `BAS_FCTS` structure for Lagrange elements of order `degree` ($0 \leq$ `degree` ≤ 4) on the reference simplex of barycentric coordinates K_d^b for all dimensions.

The entry `bas_fcts_data` of the `BAS_FCTS` structure is a pointer to a vector storing the Lagrange nodes in barycentric coordinates.

How to generate finite elements spaces with ALBERTA

A finite element space on a mesh can only be accessed by the function:

```
const FE_SPACE *get_fe_space(MESH *mesh,const char *name,
             const int ndof[DIM+1],const BAS_FCTS *);
```

The call
`get_fe_space(mesh, name,nil,bas_fcts)`
defines a new finite element space on mesh. it returns a newly created FE_SPACE structure where name is duplicated, and the members mesh, bas_fcts and admin are adjusted correctly. In particular admin manages DOFs uniquely defined by bas_fcs->n_dof.

Degrees of freedom are directly connected with the mesh, therefore the `MESH` structure contains a reference to all sets of `DOFs` which are used on a mesh, i.e. to the corresponding `DOF_ADMIN`s

$$+$$

To each finite element space corresponds a set of `DOFs`, i.e a `DOF_ADMIN`

$$\Downarrow$$

Finite element spaces and corresponding sets of `DOFs` are generated during the initialization of the mesh via the `GET_MESH()` routine.

To this end a user defined function `init_dof_admin()` is called during the initialization of the mesh by `GET_MESH`, such a function is the second argument of the `GET_MESH()` routine. The function `init_dof_admin()` should define finite element spaces and related `DOF_ADMIN`s by calling `get_fe_space()`.

Since a mesh give only access to the `DOF_ADMIN`-s defined on it, pointers to the `FE_SPACE` structures should be stored in some global variables.

No further `DOF_ADMIN`s can be added to a mesh after its generation.

Example: from file `ellipt.c`

We define:

```
static const FE_SPACE *fe_space;
...

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;
}

int main(int argc, char **argv)
{
 FUNCNAME("main");
 MESH   *mesh;
 ...

 mesh = GET_MESH("ALBERTA mesh", init_dof_admin,nil);

 ...
}
```

Error of finite element approximations

Once chosen a finite element space $\mathcal{S}$ as before, a function $u_h \in \mathcal{S}$ is identified by the vector of its coefficents which are stored in a `DOF_REAL_VEC *u_h`

For test purpose it is often convenient to calculate the "exact error" between a finite element approximation u_h and the exact solution u of a given PDE problem.

ALBERTA supplies functions to calculate such error in several norms (for example: $||.||_{L^2}$, $|.|_{H^1}$).

```
REAL L2_err(REAL (*)(const REAL_D), const DOF_REAL_VEC *,
            const QUAD *, int, REAL *(*)(EL *), REAL *);

REAL H1_err(const REAL *(*)(const REAL_D),
            const DOF_REAL_VEC *, const QUAD *, int,
            REAL *(*)(EL *), REAL *);
```

The local element errors $\int_E |\nabla(u - u_h)|^2$ or $\int_E |u - u_h|^2$ can be stored on the element leaf data.

```
L2_err(u,u_h,quad,0,rw_el_err,nil)
```

returns an approximation of the error
$(\int_\Omega |u - u_h|^2)^{1/2}$

```
H1_err(grd_u,u_h,quad,0,rw_el_err,nil)
```

returns an approximation of the error
$(\int_\Omega |\nabla(u - u_h)|^2)^{1/2}$

If `rw_el_err` is not `nil` the corresponding local error is stored on leaf elements.
`(* rw_el_err)(el)` is a user defined function which provides for each leaf element `el` an address where the local error is stored
(see `rw_el_est()` in `ellipt.c`).