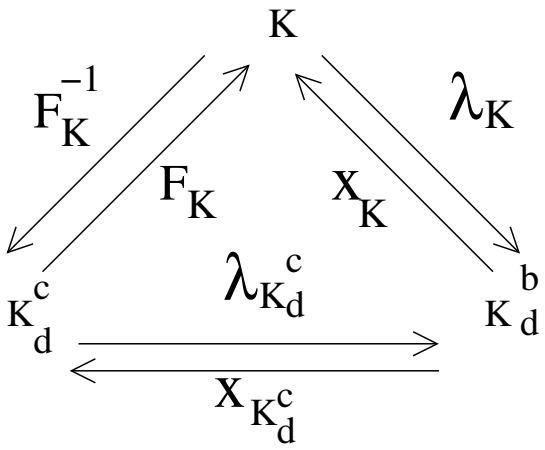
d-Simplexes

Let
$K = \text{conv}(v_0, v_1, \ldots, v_d) \in \mathcal{M} \subset \mathbb{R}^d$ d-simplex,
$K_d^c = \text{conv}(0, e_1, \ldots, e_d) \subset \mathbb{R}^d$ reference d-simplex of cartesian coordinates
$K_d^b = \text{conv}(e_1, \ldots, e_{d+1}) \subset \mathbb{R}^{d+1}$ reference d-simplex of barycentric coordinates



F_K affine $\longrightarrow$ the Jacobian DF_K is constant onK_d^c. Therefore

$$|K| = \int_K 1\, dy = \int_{K_d^c} 1\, |\text{det}DF_K| dx$$
$$= |\text{det}DF_K||K_d^c| = \frac{1}{d!}\, |\text{det}DF_K|$$

and thus

$$|\text{det}DF_K| = d!|K|$$

The map x_K is implemented in the Alberta function:

```
const REAL *coord_to_world(const EL_INFO *,
                           const REAL *,REAL_D);

coord_to_world(el_info,lambda,world);
```

The map λ_K is implemented in the Alberta function: :

```
int world_to_coord(const EL_INFO *,
                   const REAL *,REAL [DIM+1]);

world_to_coord(el_info,world,lambda);
```

for the computation of $|\det DF_K|$:

```
REAL el_det(const EL_INFO *);

det= el_det(el_info);
```

for the computation of $|K|$:

```
REAL el_volume(const EL_INFO *);

vol= el_volume(el_info);
```

Numerical integration on K_d^c

It is very convenient to use $d+1$ barycentric coordinates as a local coordinate system for describing functions on a simplex.
On the other hand, for numerical integration on an element it is much more convenient to use the reference d-simplex of cartesian coordinates.
Therefore we define the quadrature formula $Q_{K_d^c}$ on K_d^c, for the computation of

$$\int_{K_d^c} g(x)dx$$

through the set of pairs

$$\{(\omega_k, \lambda^k) \in \mathbb{R} \times \mathbb{R}^{d+1}, \quad k = 0, \ldots, n_Q - 1\}$$

where
$\omega_k \in \mathbb{R}$ are the weights
$\lambda^k \in K_d^b$ are the quadrature nodes given in baycentric coordinates

$$\int_{K_d^c} g(x)dx \simeq Q_{K_d^c}(g) := \sum_{k=0}^{n_Q-1} \omega_k \, g(x_{K_d^c}(\lambda^k))$$

$Q_{K_d^c}$ is exact of degree p ($\in \mathbb{N}$) if

$$\int_{K_d^c} q(x)dx = Q_{K_d^c}(q) \quad \forall q \in \mathbb{P}_p(K_d^c)$$

it is stable if

$$\omega_k > 0 \qquad \forall\, k = 0, \ldots, n_Q - 1$$

In Alberta a quadrature formula is described by the following data structure:

```
typedef struct quadrature       QUAD;

struct quadrature
{
  char           *name;
  int            degree;

  int            dim;
  int            n_points;
  const double   **lambda;
  const double   *w;
};
```

Stable numerical quadrature formulas exact up to degree 19 in 1d, 17 in 2d and 7 in 3 dimension are implemented and provided by Alberta.

To get a quadrature formula we can use the library function:

```
const QUAD *get_quadrature(unsigned dim,
                           unsigned degree);
```

```
quad = get_quadrature(dim,degree);
```

it returns a pointer to a filled `QUAD` data structure for numerical integration in dimension `dim` which is exact of degree `degree` (more precisely min(19, `degree`) for `dim==1`, min(17, `degree`) for `dim==2` and min(7, `degree`) for `dim==3`).

Numerical integration on a simplex K

A given numerical quadrature $Q_{K_d^c}$ on K_d^c defines a numerical quadrature Q_K for the approximation of

$$\int_K f(y)\,dy = \int_{K_d^c} f(F_K(x))|\mathrm{det}DF_K|dx$$

$$\begin{aligned} Q_K(f) &:= Q_{K_d^c}((f\circ F_K)\,|\mathrm{det}DF_K|) \\ &= \sum_{k=0}^{n_Q-1} \omega_k\, f(F_K(x_{K_d^c}(\lambda^k)))|\mathrm{det}DF_K| \\ &= \sum_{k=0}^{n_Q-1} \omega_k\, f(x_K(\lambda^k))|\mathrm{det}DF_K| \\ &= d!|K| \sum_{k=0}^{n_Q-1} \omega_k f(x_K(\lambda^k)) \end{aligned}$$

since $F_K \circ x_{K_d^c} = x_K$.
Thus we approximate

$$\int_K f(y)\,dy \simeq d!|K| \sum_{k=0}^{n_Q-1} \omega_k f(x_K(\lambda^k)) \qquad (1)$$

Numerical integration on Ω

Given $f : \Omega \longrightarrow \mathbb{R}$, $\mathcal{M}$ s.t. $\overline{\Omega} = \cup_{K \in \mathcal{M}} K$

To approximate

$$\begin{aligned} \int_\Omega f(y)\, dy &= \sum_{K \in \mathcal{M}} \int_K f(y)\, dy \\ &= \sum_{K \in \mathcal{M}} \int_{K_d^c} f(F_K(x)) |\text{det} DF_K| dx \end{aligned}$$

we can use a quadrature formula $Q_{K_d^c}$, and compute

$$\sum_{K \in \mathcal{M}} d! |K| \sum_{k=0}^{n_Q - 1} \omega_k f(x_K(\lambda^k)) \qquad (2)$$

In practice: if f is a given function in cartesian coordinates, to compute $\int_\Omega f$ we implement formula (2), i.e. we traverse the mesh and sum up in a global variable the contributions to the integral on each element K.

Once fixed a quadrature formula
`quad=get_quadrature(DIM,degree);`
to approximate $\int_K f$ we have to compute the sum in equation (1) with the help of the library functions `coord_to_world` and `el_det`.

Note: The mesh taversal can be performed recursively or not, in the first case the `quad` structure storing the quadrature formula should be a global variable since the function called on every element during mesh traversal has a unique parameter `el_info` containing information of the element K.

A library function for numerical integration

Alberta provides the function:

```
const REAL integrate_std_simp(
const QUAD * quad, REAL (*f)(const REAL *));
```

The call

```
int= integrate_std_simp(quad,g);
```

approximates the integral of g by the numerical quadrature described by quad;
g is a pointer to the function to be integrated, evaluated in barycentric coordinates.
The return value is

$$\sum_{k=0}^{\texttt{quad->n\_points}-1} \texttt{quad->w[k]} * (*\texttt{g})(\texttt{quad->lambda[k]})$$

If f is a given function in cartesian coordinates, an alternative way to approximate $\int_K f$ by means of a quadrature formula `quad` is:

- we define the function $f_l = f \circ x_K$ evaluated in barycentric coordinates as:

```
REAL f_l( const REAL lambda[DIM+1])
{
  REAL_D  x;

  coord_to_world(elinfo, lambda, x);
  return(f(x));
}
```

- we set the external variable `elinfo` so that it refers to K, and is visible inside `f_l`.

- we call `integrate_std_simp(quad,f_l)` and multiply the result by the return value of `el_det(elinfo)`, i.e. by $|\det DF_K|$.

Note: We have to pay attention that the variable `elinfo` is correctly setted.

Storing data on the leaf elements of the hierarchical mesh

Many finite element application need special information on each element of the current triangulation, i.e. the leaf elements of the hierarchical mesh
(Ex. error indicators in adaptive codes).

Leaf data elements do not have children and thus both pointers to the children are `nil` and not used.

To distinguish leaf elements from other elements in the hiererchical mesh it is enough to control if the pointer to the first child is zero.

These facts suggest to use one of the `child` pointers to enable access to special data for leaf elements without introducing additional pointers in the element data structure.

In order to make Alberta's element data structure as small as possible, and store data elementwise on the mesh leafs, it is allowed to "hide" leaf data at the pointer of the second child on leaf elements.

The following macro for testing leaf elements and accessing data stored on the leafs are provided:

```
#define IS_LEAF_EL(el) (!(el)->child[0])
#define LEAF_DATA(el) ((void *)(el)->child[1])
```

Informations for leaf elements depends strongly on the application, therefore it seems not to be appropriate to define a fixed data type in Alberta for storing them.

Alberta's users can define their own type for data that should be present on leaf elements, Alberta only need to know the size of memory that is required to store such data, (one entry in the structure `leaf_data_info` described below)

During refinement and coarsening Alberta automatically allocates and deallocates memory for user data on leaf elements if the data size is bigger than zero.

In order to handle an arbitrary kind of user data on leaf elements it is defined the following data structure:

```
typedef struct leaf_data_info        LEAF_DATA_INFO;


struct leaf_data_info
{
 const char *name;
 unsigned   leaf_data_size;
 void      (*refine_leaf_data)(EL *parent, EL *child[2]);
 void      (*coarsen_leaf_data)(EL *parent, EL *child[2]);

 const void  *leaf_data_block_info; /* not for user!!!*/
};
```

The entry `leaf_data_size` contains the size of memory space which is used for storing leaf data.
The entries `refine_leaf_data` and `coarsen_leaf_data` are used to allow the user to provide routines for transformation of leaf data from parent to childern during refinement or respectively from children to parent during coarsening.

A pointer to such a structure is an entry in the `MESH` structure.

```
typedef struct mesh        MESH;

struct mesh
{
  const char       *name;
  int              n_vertices;
  int              n_elements;
  int              n_hier_elements;

#if DIM == 2
  int              n_edges;
#endif

  ...............

  int              n_macro_el;
  MACRO_EL         *first_macro_el;

  LEAF_DATA_INFO   leaf_data_info[1];

  DOF_ADMIN        **dof_admin;
  int              n_dof_admin;

  ................................

/*--------------------------------------------*/
/*-- pointer for administration; don't touch!--*/
/*--------------------------------------------*/

  void             *mem_info;
};
```

Example from file `ellipt.c`

In order to store the error estimates elementwise on the current mesh, we need:

1) To define a leaf data type:

```
struct ellipt_leaf_data
{
  REAL estimate;  /*one real for the estimate */
};
```

2) to define e function for initializing the structure `leaf_data_info`, which gives Alberta information about size of leaf data needed

```
void init_leaf_data(LEAF_DATA_INFO *leaf_data_info)
{
  leaf_data_info->leaf_data_size =
                    sizeof(struct ellipt_leaf_data);
  leaf_data_info->coarsen_leaf_data = nil;
  leaf_data_info->refine_leaf_data = nil;
  return;
}
```

3) We have to pass information about the size of leaf data to be stored during mesh initialization by GET_MESH() in the main program. This is done by calling the function init_leaf_data() in GET_MESH(). Therefore init_leaf_data shoud be an argument in the call of GET_MESH.

```
int main(int argc, char **argv)
{
 FUNCNAME("main");
 MESH   *mesh;
 ...
 char   filename[100];


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


 GET_PARAMETER(1,"macro file name","%s",filename);
 read_macro(mesh, filename, nil);
 .....
 graphics(mesh, nil, nil);
 ...........
}
```

4) Finally, we define a function which gives read and write access to the local element error estimate

```
static REAL *rw_el_est(EL *el)
{
 if (IS_LEAF_EL(el))
  return(
    &((struct ellipt_leaf_data *)LEAF_DATA(el))->estimate);
 else
 return(nil);
}
```

and a function which returns the local error estimate

```
static REAL get_el_est(EL *el)
{
 if (IS_LEAF_EL(el))
  return(
    ((struct ellipt_leaf_data *)LEAF_DATA(el))->estimate);
 else
 return(0.0);
}
```

where `LEAF_DATA` is the macro that returns the pointer to the hidden data in the second child of a leaf element.