Curved boundaries

To construct meshes of domains with curved boundaries we need to define a function, called `ibdry`, whose aim is to assign properly a `BOUNDARY` structure to each boundary edge/face of a macro element while reading data from file for the macro mesh.

Recall

```
typedef struct boundary       BOUNDARY;

struct boundary
{
void    (*param_bound)(REAL\_D *coord)
S_CHAR   bound;
};
```

The function `ibdry` is called for each edge/face during the initialization of the mesh stored in the structure `mesh`. The sintax is:

```
BOUNDARY *ibdry(MESH *mesh, int bound)
```

`bound` is an integer value read from the macro mesh data file (one integer for each edge/face). It is used inside `ibdry` to distinguish different initializations of `param_bound` with appropriate functions for projecting the midpoint of the refinement edge onto the curved boundary.

The return value of `ibdry` is a pointer to a filled boundary structure (`BOUNDARY`).

Since `ibdry` is used during the initialization of the mesh, it is the third argument to the function `read_macro()` which reads data for the macro triangulation:

```
read_macro(mesh, filename, ibdry);
```

For polygonal domains:

```
read_macro(mesh, filename, nil);
```

and the corresponding `BOUNDARY` pointers are adjusted to the defaults structures `dirichlet_bondary` or `neumann_boundary` for positive respectively negative values of `bound`.

Example: first quarter of the unit disc

Consider the domain given by the first quarter of the unit disc and suppose Dirichlet boundary conditions.

The following data file defines one single triangle as initial triangulation.

quart_circ.amc

```
DIM: 2
DIM_OF_WORLD: 2
number of elements: 1
number of vertices: 3
element vertices:
1 2 0
element boundaries:
1 1 2
vertex coordinates:
 0.0 0.0
 1.0 0.0
 0.0 1.0
element neighbours:
-1 -1 -1
```

We define:

```
const BOUNDARY *ibdry(MESH *mesh, int bound)
{
  FUNCNAME("ibdry");
  static const BOUNDARY circ_dirichlet=
                       {ball_project, DIRICHLET};
  static const BOUNDARY straight_dirichlet=
                       {nil, DIRICHLET};

  switch (bound)
   {
    case 1:  return(&straight_dirichlet);
    case 2:  return(&circ_dirichlet);
    default: ERROR_EXIT("no boundary %d\n",
                         bound);
   }
}
```

and the function which projects a newly generated vertex on the curved boundary:

```
static void ball_project(REAL_D p)
{
  FUNCNAME("ball_project");
  REAL      norm=NORM_DOW(p);

  norm=1.0/MAX(1.0E-10, norm);
  SCAL_DOW(norm,p);
  return;
}
```

we read the initial triangulation with

```
read_macro(mesh, filename, ibdry);
```

Some further refinements of the macro mesh stored in file `quart_circ.amc` will better approximate the curved domain.

Hierarchical mesh traversal routines

Alberta provides recursive and non recursive traversal routines that can be used to perform some desired operation on some selected elements of the hierarchical mesh.

To save computer memory, all possible information that can be available for mesh elements is stored explicitly only for elements of the macro mesh.
Some information is transferred to the other elements while traversing the forest of binary trees.

Therefore Alberta's routines use the following structure to store for the current element all information which is not stored on the element explicitly but may be generated from the parent during a traversal of the hiererchical mesh.

Most entries in the structure are only filled if requested.

```
typedef struct el_info        EL_INFO;

struct el_info
{
  MESH                 *mesh;
  REAL_D               coord[N_VERTICES];
  const MACRO_EL       *macro_el;
  EL                   *el, *parent;
  FLAGS                fill_flag;
  S_CHAR               bound[N_VERTICES];
#if DIM == 2
  const BOUNDARY       *boundary[N_EDGES];
#endif

#if DIM == 3
  const BOUNDARY       *boundary[N_FACES+N_EDGES];
#endif

  U_CHAR               level;

#if ! NEIGH_IN_EL
  EL                   *neigh[N_NEIGH];
  U_CHAR               opp_vertex[N_NEIGH];
#if DIM == 3
  U_CHAR               el_type;
#endif
#endif
  REAL_D               opp_coord[N_NEIGH];

#if DIM == 3
  S_CHAR               orientation;
#endif
};
```

During a traversal of the hiererchical mesh we can operate on selected elements, therefore the traversal routines need to know:

- which data should be available for each element on which we want to operate

- on which elements (internal or leafs) the traversal should stop to perform a desired operation.

Both information are passed to the traversal routines with the help of flags.

Flags to indicate on which elements the operation should take place:

CALL_EVERY_EL_PREORDER	on all hierarchical elements
CALL_EVERY_EL_INORDER	"
CALL_EVERY_EL_POSTORDER	"
CALL_EL_LEVEL	on all tree elements at a specified tree depth
CALL_LEAF_EL	on all leaf elements
CALL_LEAF_EL_LEVEL	on all leaf elements at a specified tree depth

The first three differs in the sequence of operation on elements:

CALL_EVERY_EL_PREORDER	parent - child[0] - child[1]
CALL_EVERY_EL_INORDER	child[0] - parent - child[1]
CALL_EVERY_EL_POSTORDER	child[0] - child[1] - parent

Additional flags are defined that specify which local information in EL_INFO has to be generated during the hierarchical mesh traversal:

FILL_NOTHING	No information needed at all
FILL_COORDS	vertex coordinates EL_INFO.coord are filled
FILL_BOUND	boundary information is filled in the entries E_INFO.bound and E_INFO.boundary
FILL_NEIGH	neighbour element information E_INFO.neigh and E_INFO.opp_vertex are generated
FILL_OPP_COORDS	coordinates of the opposite vertex of neighbours through common edges are stored in E_INFO.opp_coords

Recursive mesh traversal roules

The sequence of elements which are visited during the traversal follows the following roules:

- All elements in the binary tree of one `MACRO_EL mel` are visisted before any elements in the tree of `mel->next`.
- For every `EL el`, all elements in the subtree `el->child[0]` are visited before any element in the subtree `el->child[1]`.
- The traversal order of an element and its two child trees is determined by the flags:

```
CALL_EVERY_EL_PREORDER     parent-child[0]-child[1]
CALL_EVERY_EL_INORDER      child[0]-parent-child[1]
CALL_EVERY_EL_POSTORDER    child[0]-child[1]-parent
```

Recursive mesh traversal routines:

```
void mesh_traverse(MESH *mesh, int level,
                   FLAGS fill_flags,
                   void (*el_fct)(const ELINFO *));
```

el_fct is a pointer to a user defined function which performs a desired operation on a single selected element.

Example: Computation of the domain's measure.

On each leaf element the volume of the simplex can be computed by the library function el_volume() and added to a global variable area_omega previously initialized to 0. After a mesh traversal, area_omega finally holds the measure of the domain.

```
REAL area_omega;

void area_fct( const EL_INFO * elinfo)
{
 area_omega+=el_volume(elinfo);
return;
}
```

and in main:

```
area_omega = 0.0;
mesh_traverse(mesh,-1,CALL_LEAF_EL|FILL_COORDS,area_fct);
MSG(``|Omega| = %e\n'', area_omega);
```

Alberta's MACRO for printing messages

#define FUNCNAME(nn) const char * funcName=nn

```
MSG("indice el:%d, area:%lf\n" el->index,area);


ERROR("cannot open file %s\n", filename);
ERROR_EXIT("allocated size too small\n");


TEST(level>=0)("invalid level:%d\n",level);
TEXT_EXIT(el)("no element for refinement\n");
```

Non-recursive traversal routines:

The implementation of the non-recursive traversal routines uses a stack to save the tree path from a macro element to the current one. The data structure

```
typedef struct traverse_stack TRAVERSE_STACK
```

holds such information.

The administration of the space of memory for the stack is done by the following library functions:

```
TRAVERSE_STACK * get_traverse_stack(void);
```

```
void free_traverse_stack (TRAVERSE_STACK * stack);
```

A mesh traversal is launched by a call to the function:

```
const EL_INFO * traverse_first( TRAVERSE_STACK * stack,
                                MESH mesh, int level,
                                FLAGS fill_flags);
```

Advancing to the next element is done by the function:

```
const EL_INFO * traverse_next( TRAVERSE_STACK *stack,
                               const EL_INFO * el_info);
```

Example:

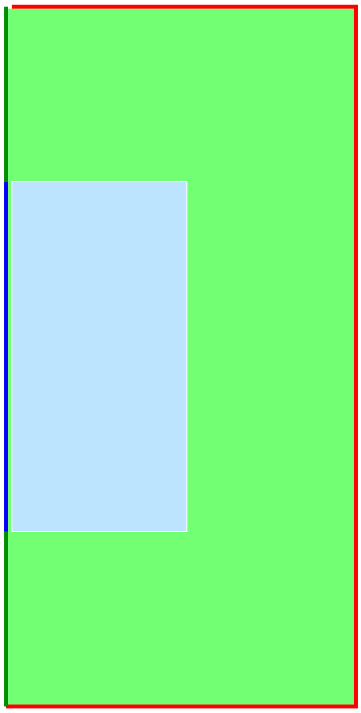
```
TRAVERSE_STACK  *stack;
EL_INFO         *el_info;
FLAGS           fill_flag=CALL_LEAF_EL|FILL_COORDS;
int             level;

stack=get_traverse_stack();
for(el_info=traverse_first(stack,mesh,level,fill_flag);
    el_info;
    el_info=traverse_next(stack,el_info);)

      /* operation on element */
   }
free_traverse_stack(stack);
```

or using a conditioned cycle:

```
el_info=traverse_first(stack,mesh,level,fill_flag);
while(el_info)
     {
        /* operations on element */
       el_info=traverse_next(stack,el_info));
     }
```

Example 3: Heat diffusion in a refrigerator at steady state

We want to study the distribution of the temperature inside a refrigerator at steady state (i.e. no change in time). To construct a model for the refrigerator we consider the following pattern



The light blue region, Ω_0, is the interior of the refrigerator. The green one, Ω_1, corresponds to the sides and the door of the fridge. Γ_0, in blue, is the refrigerated wall, Γ_1,in red, is the boundary at room temperature and Γ_2, in green, is part of the boundary where we suppose the temperature varies linearly.

The model problem

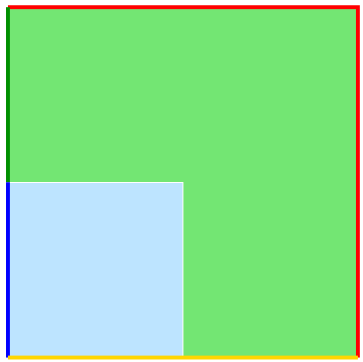
$$\begin{cases} -\text{div}(A(x)\,\nabla u) = 0 & \text{in } \Omega = \Omega_0 \cup \Omega_1 \\ u = 5 & \text{on } \Gamma_0 \\ u = 20 & \text{on } \Gamma_1 \\ u(x_1, x_2) = 15\,x_2 + 5 & \text{on } \Gamma_2 \end{cases}$$

$$A(x) = \begin{cases} 1 & \text{if } x \in \Omega_0 \\ 0.1 & \text{if } x \in \Omega_1 \end{cases}$$

$A(x)$ describes the thermal conductivity, i.e. the property of the material to conduct heat
(the bigger is A the less the meterial is insulating).

u is the temperature.

For symmetric reason we can restrict our model to the upper half of the previous scheme.



here the yellow line correspond to an artificially created boundary, Γ_3, where we assume homogeneous Neumann condition in order to ensure thermal insulation (i.e. no heat flux). The problem reads now

$$\begin{cases} -\text{div}(A(x)\,\nabla u) = 0 & \text{in } \Omega = \Omega_0 \cup \Omega_1 \\ u = 5 & \text{on } \Gamma_0 \\ u = 20 & \text{on } \Gamma_1 \\ u(x_1, x_2) = 15\,x_2 + 5 & \text{on } \Gamma_2 \\ \frac{\partial u}{\partial n} = 0 & \text{on } \Gamma_3 \end{cases}$$

where $\Omega = (-1,1)^2 = \Omega_0 \cup \Omega_1$
and $\Omega_0 = (-1,0)^2$

We investigate the temperature with the help of numerical simulations in order to find an answer to the following questions:

- How is the shape of the graph of u? is the temperature function smooth?
  (There is an edge across $\partial\Omega_1 \cap \partial\Omega_2$)

- What happens if A doubles?
  (No linear dependency of the solution from data A)

- Where is the highest temperature inside the fridge? What is its value?

  To preserve better food don't put it at the corners of the door!