

Master de Mathématiques  
Parcours "Modélisation et Analyse Numérique"

Programmation II - HAX011X - 2025/2026

CC4 - final

The mandatory deliveries are specified in *red color*. Provided files names are specified in *blue color*. Please, respect the naming convention. The use of LLM chatbots is strictly forbidden.

Use the provided classes coming from the FVM hierarchy studied during the PW1 to PW4, available in the src folder on the Moodle website CC4 section.

The overall goal of this last evaluation is to implement the edge-based FVM method described in the last Exercise of PW4.

Let us recall that, for a given mesh instance, calling the method `build_edges()` populates the container `std::vector<edge> edges_` and that for each edge, the member `neighbors_` stores the indices of the neighboring triangles (`neighbor(0)` is called the "left" triangle and `neighbor(1)` is called the "right" triangle, following the conventions described during PW4).

**Exercise 1. the face class: length, centroid and unit-normal**

Following the lines of the element class, you have to supplement the face class with the following new members and methods to allow an easier workflow with `edges` instances:

(1) the centroid and length of the `edges` are needed. To this end:

- **add two protected members** declared as `point2d centroid_` and **double** `length_` in the face class, together with the usual corresponding public accessing functions,
- **add two public methods** `point2d compute_centroid()` and **double** `compute_length()` in the face class, to populate the two previous members,

(2) a unit normal vector to each `edge` of the mesh is needed. To this end:

- **add a protected member** declared as `point2d normal_` in the face class, together with the usual corresponding public accessing functions,
- **add a public method** `point2d compute_unit_normal()` in the face class, to populate the previous member. The orientation of the normal vector should be *outgoing* for the boundary faces, and is arbitrarily chosen for the interior faces: *the vectors should point from neighbor(0) towards neighbor(1)* (from the "left" triangle to the "right" triangle), see Fig. 1 for our buddy mesh.

(3) in the mesh class, **add the following new methods:**

```
int compute_edges_centroids();  
double compute_edges_length();  
int compute_edges_unit_normals();
```

which delegate the computations for each `edge` instance contained in the `std::vector<face> edges_` member (and are of course assumed to be called after `build_edges` has populated the `edges_`

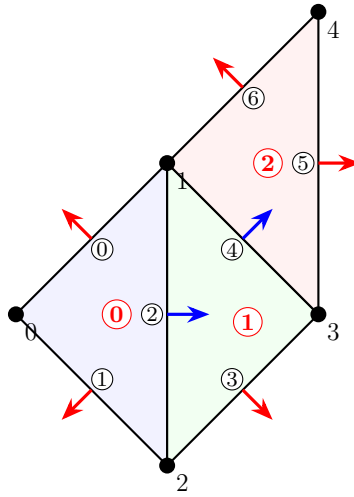


Figure 1. Triangulation with normal vectors to edges.

vector, see the [main\\_ex1.cpp](#) file). Note that `compute_edges_length()` should return a **double** which is the minimum of the edges lengths.

- (4) **test your functions** with the provided [main\\_ex1.cpp](#) file. Deliver the modified [class\\_face.hpp](#) and [class\\_mesh.hpp](#) file.

## Exercise 2. the `advection_solver` class: a new edge-based `advance_edge` method

- (1) in the `advection_solver` class, **add a new method** called **`void advance_edge(double T_final, point2d velocity)`** which modifies the initial **`void advance(double T_final, point2d velocity)`** method to implement the edge-based loop algorithm described in Exercise 7 of PW4, using the new data-structures and methods obtained in Exercise 1. Another help may be provided by the following algorithm flowchart:

---

### Algorithm 1:

---

```

1   $t \leftarrow 0$ ;  $step \leftarrow 0$ 
2  while  $t < T_{final}$  do
3      foreach  $T_i \in \mathcal{T}_h$  do
4           $\Phi_i \leftarrow 0$  // Init flux
5      foreach  $e \in edges()$  do
6           $v_n \leftarrow \vec{v} \cdot \vec{n}_e$ ;
7           $T_L, T_R \leftarrow neighbors(e)$ 
8          if  $v_n > 0$  then  $f \leftarrow v_n * u[T_L] * |e|$ 
9          else  $f \leftarrow v_n * (T_R \neq -1 ? u[T_R] : 0) * |e|$  // 0 for bound. cond.
10
11          $\Phi_{T_L} \leftarrow \Phi_{T_L} + f$ ;
12         if  $T_R \neq -1$  then  $\Phi_{T_R} \leftarrow \Phi_{T_R} - f$ 
13     foreach  $T_i \in \mathcal{T}_h$  do
14          $u[i] \leftarrow u[i] - (\Delta t / area_i) * \Phi_i$ 
15      $t \leftarrow t + \Delta t$ ;  $step \leftarrow step + 1$ 

```

---

- (2) **test your function** with the provided [main\\_ex2.cpp](#) file.

- (3) it is possible to further (and simply) optimize this method by avoiding the last loop on elements. How can we proceed ?

Deliver the modified files [advection\\_solver.hpp](#).