UNIVERSITÉ DE MONTPELLIER

FACULTÉ DES SCIENCES DE MONTPELLIER

MASTER DE MATHÉMATIQUES

PARCOURS *"Modélisation et Analyse Numérique"*

PROGRAMMATION II - HAX011X - 2025/2026

PW4 - FINITE-VOLUMES FOR ADVECTION ON TRIANGULATIONS

The **Finite Volume Method (FVM)** is a numerical technique used to solve Partial Differential Equations (PDEs) by dividing a physical domain into a series of small, non-overlapping regions called control-volumes. It is one of the most prevalent approach in Computational Fluid Dynamics (CFD) because it inherently satisfies physical conservation laws, and it is also one of the most simple. Note that in full generality, control-volumes do not necessarily coincide with mesh elements. Yet, we choose such a *cell-centered* approach (where control-volumes do coincide with mesh elements) in this Practical Work, for the sake of simplicity.

The FVM has several assets and advantages:

- **integral formulation**: unlike the Finite Difference Method (FDM), which approximates derivatives at specific points, FVM starts by integrating governing equations over each control volume.
- **conservation**: the method ensures that the flux entering a given volume is exactly equal to the flux leaving the adjacent volume. This "local conservation" property makes it exceptionally robust for several physical problems.
- **averaging**: variables are typically stored at the volumes centers and are defined as the average values of the physical quantity over the entire control volume.
- **geometric flexibility:** FVM is easily formulated on unstructured meshes, allowing to handle complex, irregular geometries that are difficult to account for for traditional FDM.
- **industry standard:** it serves as the foundation for several major commercial and open-source CFD software, including ANSYS Fluent and OpenFOAM.

The basic implementation steps of a standard FVM for an evolution equation are the following

- **domain discretization (meshing):** the geometry is divided into a mesh of contiguous control volumes. These can be regular (Cartesian grids) or irregular (unstructured meshes using triangles or tetrahedrons).
- **integration:** The governing PDEs are integrated over each individual control volume using the Divergence Theorem to convert volume integrals into surface integrals.
- **flux approximation:** fluxes across the faces of each control volume are approximated using interpolation schemes (*i.e.*, up-winding for stability) based on values from neighboring cells.
- **time advancing:** this process converts the original continuous equations into a loop involving linear algebraic equations.

Let get deeper into details. For a scalar quantity u transported by a velocity field $\mathbf{v} := (v_x, v_y)^T$, the linear advection equation in conservative form with homogeneous Dirichlet boundary conditions:

$$
(1) \qquad \begin{cases} \dfrac{\partial u}{\partial t} + \nabla \cdot G(u) = 0, \quad \text{with} \quad G(u) := \mathbf{v}u, & \text{in } [0,T] \times \Omega, \\ u(0,\mathbf{x}) = u_0(\mathbf{x}) & \text{in } \Omega, \\ u(t,\mathbf{x}) = 0 & \text{on } [0,T] \times \partial\Omega. \end{cases}
$$

Introducing a triangulation $\mathscr{T}_h$ of the polygonal computational domain: $\Omega := \bigcup_{T\in\mathscr{T}_h} \overline{T}$, we integrate the equation over a control volume $T \in \mathscr{T}_h$ with area $|T|$, applying the divergence theorem:

$$\int_T \frac{\partial u}{\partial t} d\mathbf{x} + \int_{\partial T} G(u)\cdot \mathbf{n}_T \, ds = 0. \tag{2}$$

In a discrete framework, we define the cell-centered FV unknowns by cell-averaging

$$\overline{u}_T(t) := \frac{1}{|T|}\int_T u(t,\mathbf{x})\, d\mathbf{x},$$

the surface integral is replaced by a summation over the faces $F \in \mathcal{F}_T$ of the cell:

$$|T|\frac{d}{dt}\overline{u}_T + \sum_{F\in\mathcal{F}_T}\int_F G(u)\cdot\mathbf{n}_{TF}\, ds = 0, \tag{3}$$

where $\mathbf{n}_{TF}$ is the unit-normal vector to the face F, outward with respect to the triangle T.

The quantity $G(u)\cdot\mathbf{n}_{TF} = u(\mathbf{v}\cdot\mathbf{n}_{TF})$, restricted to $F \in \partial T$, is called *Finite-Volume interface flux* and, considering the discrete approximation setting which involves piecewise-constant elements values, is constant value along the face F. Let denote by $\mathcal{G}_{TF}$ such an interface *numerical flux*, such that we have:

$$|T|\frac{d}{dt}\overline{u}_T + \sum_{F\in\mathcal{F}_T} |F|\,\mathcal{G}_{TF} = 0. \tag{4}$$

For the advection equation, the numerical flux $\mathcal{G}_{TF}$ through face $F = \partial T \cap \partial T'$ is computed as follows (upwind):

$$\mathcal{G}_{TF} := \left[\max(\mathbf{v}_F\cdot\mathbf{n}_{TF},0)\overline{u}_T + \min(\mathbf{v}_F\cdot\mathbf{n}_{TF},0)\overline{u}_{T'}\right]. \tag{5}$$

The numerical flux definition should of course be adapted for more elaborated (nonlinear) equations. The initial-data is obtained by averaging u_0:

$$\overline{u}_T(0) := \frac{1}{|T|}\int_T u_0(\mathbf{x})\, d\mathbf{x} =: \overline{u}_T^0.$$

Considering the triangulated polygonal computational domain $\Omega := \bigcup_{T\in\mathscr{T}_h} \overline{T}$, the basic semi-discrete FVM reads as: *knowing* $\{\overline{u}_T(0),\ T\in\mathscr{T}_h\}$, $\forall t>0$, *find* $\{\overline{u}_T(t),\ T\in\mathscr{T}_h\}$ *such that*

$$\forall T\in\mathscr{T}_h,\ |T|\frac{d}{dt}\overline{u}_T + \sum_{F\in\mathcal{F}_T}|F|\,\mathcal{G}_{TF} = 0.$$

Adding a first-order in time upward discretization, the fully-discrete FVM scheme reads as:

knowing $\{\overline{u}_T^0,\ T\in\mathscr{T}_h\}$, $\forall n\geq 0$, *find* $\{\overline{u}_T^{n+1},\ T\in\mathscr{T}_h\}$ *such that*

$$\forall T\in\mathscr{T}_h,\ \overline{u}_T^{n+1} := \overline{u}_T^n - \frac{\delta_t^n}{|T|}\sum_{F\in\mathcal{F}_T}|F|\,\mathcal{G}_{TF}^n, \tag{6}$$

where the superscript n in $\mathcal{G}_{TF}^n$ refers to the fact that the numerical fluxes are computed from the values at discrete time n, to obtain new values at time $n+1$, and δ_t^n is a discrete time-step which should be chosen to ensure stability of the computation.

Exercice 1. upgrade the element **class**

We only focus, of course, on the implementation for triangles to keep things simple. From the formulation (6), we see that several geometrical and topological data are missing from our embryonal class element and several fields and methods are needed. More precisely, regarding geometric informations, new **protected** fields should be added:

```
int index_;
// geometry (element dependent)
double area_; // value of the triangle area
POINT centroid_; // coordinates of the centroid
std::array<POINT, NVERTICES> normals_; // coordinates of the outgoing unit normals to faces
std::array<double, NVERTICES>  faces_length_; //  faces lengths
```

Corresponding methods should be added:

(1) a method compute_area() and a method area() that returns the area of the triangle computed from the coordinates of its vertices,
(2) a method compute_centroid() and a method centroid() that returns the coordinates of the center of the triangle, as a point2d,
(3) a method compute_faces_length() and a method face_length(**size_t** i) that compute, populate the std::array and return the corresponding values,
(4) methods compute_outgoing_unit_normals() and normal(**size_t** i) that compute, populate the std::array and return the corresponding values,

As far as topological data are concerned, one should keep in mind that these are mesh-dependent. We add the following protected field:

```
// global indices of neighboring triangles (-1 if boundary face)
std::array<int, NVERTICES> neighbors_;
// global indices of triangle faces seen as edges (-1 if boundary face)
std::array<int, NVERTICES> faces_;
```

together with the corresponding access methods

```
inline int neighbor(std::size_t ii) const { return this->neighbors_[ii]; };
inline void set_neighbor_index(std::size_t ii, int jj) { this->neighbors_[ii] = jj; };
inline int face(std::size_t ii) const { return this->faces_[ii]; };
inline void set_face_index(std::size_t ii, int jj) {this->faces_[ii] = jj; };
```

which gathers the global triangle indices (0-based numbering) of the neighbors (with an index -1 if there is no neighboring element), face by face, and the global edge indices (0-based numbering) of the faces regarded as edges. Note that a more efficient implementation would rely on a direct use of the addresses of such neighboring triangles, without mediating through indirect indexation.

We are left with the issue of populating such new members. This may be achieved through some slight modifications of the algorithm introduced in PW3 for the method build_edges, in the next Exercices.

Exercice 2. upgrade the face **class and the** build_edges **method**

Interestingly, we observe that gathering the required connectivity data for the neighboring elements through faces may be easily achieved by first gathering the connectivity data for the neighboring elements for edges. For a given edge $F \in \mathcal{F}_h$, we aim at identifying the two elements T and T' that share F (or only the sole element T if F belongs to the domain boundary). This may be either by knowing their (global) indices or knowing their adresses. To achieve this, we modify the class face and add the following protected members:

```
int index_;
std::array<int, NVERTICES> neighbors_; // indices of neighboring triangles (-1 if boundary edge)
//std::array<triangle*, NVERTICES> p_neighbors_; // get the addresses...not followed here
```

where NVERTICES is intended to take the value 2 for edges and we observe that these arrays may be filled during the build_edges algorithm. We also need to introduce a more elaborated constructor for the face objects:

```
face(node<POINT>&, node<POINT>&, int ind = -1, int tri_1 = -1, int tri_2 = -1);
```

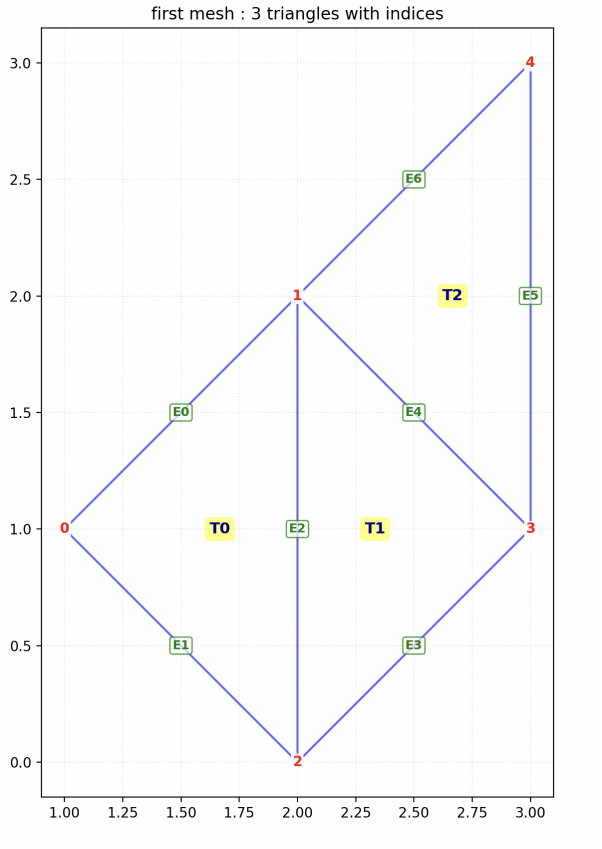
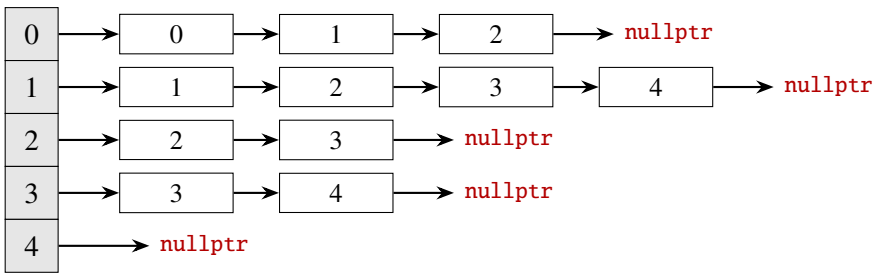


FIGURE 1. A first (very simple) mesh

so that we can directly inject as soon as it is built (and the data available), for each **new** edge, its index_ (by order of construction) and its first sharing triangle index stored in neighbors_[0] (one may even store its address if needed ...). Note that if it exists, the second sharing element also has an index and an address but these informations are generally not available at the exact time of creation of the edge. These missing data should therefore be updated later, as soon as possible, when the current edge will be encountered again during the triangle traversing algorithm of build_edges.

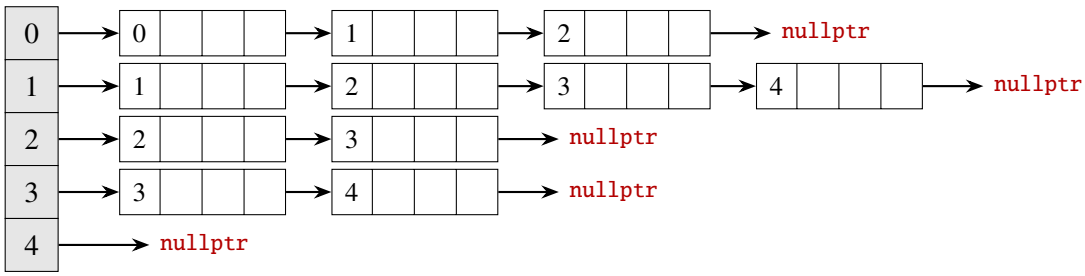
Let now consider how to modify the class mesh and the build_edges method in order to populate these new protected members related to connectivity for both face and element classes. As an example, let us recall the simple 3 elements mesh of PW3, see Fig. 3, where we show the various indices associated with geometrical (vertices) and topological (triangles and edges) entities, for further code validation purpose, together with the targeted hash-table associated with the build_edges algorithm:



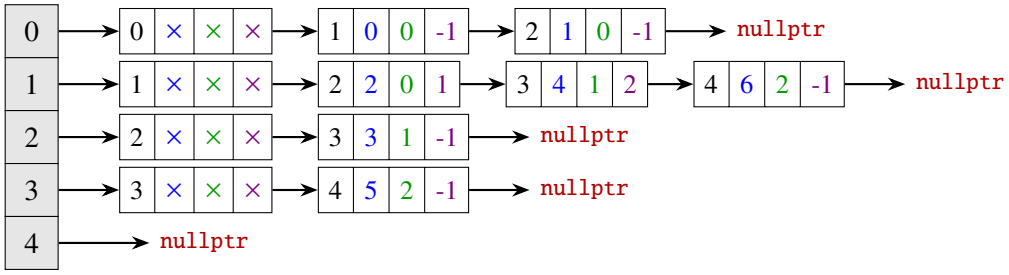
from which we may extract the seven edges, listed by couples of indices and order of construction: (0,1), (0,2), (1,2), (2,3), (1,4), (3,4), (1,4).

To gather the missing topology data, we may **enrich the data-structure associated with this hash-table**, so that each element is able to store more useful informations than the simple index of the connected vertices through edges. As a minimal requirement, we'd like to store, for each edge occurring in the

mesh, four integer values, leading to the following visual formal representation:



The new additional data-storage should be respectively devoted to store the edge index (0-based index, given by the construction rank), the "left" (interior) neighbor index and the "right" (exterior) neighbor index. For instance and validation purpose, here is the expected hashtable for the test first.mesh:



Note that, by construction, for edges belonging to the domain boundary $\partial\Omega$, the "right" neighbor has index -1 and the "left" neighbor is always interior to the domain.

How can we build and populate this structure in practice ?

Instead of working with the basic structure

`list<linked_list<std::size_t>> hashv(n_vertices)`

used in the previous attempt in PW3, we merely consider the following hash-table declaration:

`list<linked_list<connectivity_data>> hashv(n_vertices)`

where connectivity_data is a simple *ad-hoc* structure devoted to the storage and management of the topology data:

```
class connectivity_data
{
protected:
    int vertex_index_;
    int edge_index_;
    int sharing_1_index_;
    int sharing_2_index_;
public:
    connectivity_data():
    vertex_index_(-1), edge_index_(-1), sharing_1_index_(-1), sharing_2_index_(-1){};
    connectivity_data(int vv, int ed = -1, int e1 = -1, int e2 = -1):
    vertex_index_(vv), edge_index_(ed), sharing_1_index_(e1), sharing_2_index_(e2){};
    inline int vertex_index() const { return vertex_index_; };
    inline void set_vertex_index(int value) { vertex_index_ = value; };
    inline int edge_index() const { return edge_index_; };
```

```
    inline void set_edge_index(int value) { edge_index_ = value; };
};
```

With this upgraded structure in hand, we may reconsider the traversing algorithm of build_edges() method in the class mesh and locally improve it to gather and store the required data. **This is the technical part of this Exercice**. Look at the build_edges() method and be ready to modify it as follows:

- the scanners should be adapted to fit the new container:

```
// iterate through linked_list<connectivity_data>
linked_list<connectivity_data>* pc_cell      = nullptr;
linked_list<connectivity_data>* pc_cell_prev = nullptr;
```

- the traversing main loop is controlled as before, but we may **take care of the current triangle index** (and address if needed, through a tracking pointer, although not done here), as this triangle owns the current edge, and this information is sought after:

```
// looping on elements->next
    for (mesh<T>* scanner = this; scanner; scanner = (mesh<T>*)scanner->p_next())
    {
        auto current_element_index  = scanner->item().index();
        //triangle* p_current_element = &scanner->item(); // if needed
        ...
```

- of course, when initiating a new line in the hash-table, the call operator should be adapted to rely on connectivity_data instead of std::**size_t**:

```
if (!hashv.item(ismin))
   hashv.item(ismin) = new linked_list<connectivity_data>(ismin);
```

- when the position of a vertex is found in a given line (and, therefore, when we are about to add a new object in the current linked_list), two main modifications are emphasized:

  (i) we instantiate a new connectivity_data object that gathers the required data (vertex index as in PW3, but also current edge index and first ("left" or "interior") owning triangle index) and insert it into the linked_list,

  (ii) **conversely, we also add this new** edge **global index** as a face of the current element (note that this face also has a local index $0 \leq kk \leq 2$).

  Here we go:

```
if (pc_cell_prev->item().vertex_index() < ismax)
{
   // std::cout << "  position found " << std::endl;
   // new edge is created / edge index is modified
   connectivity_data tmp(ismax, edges_counter, current_element_index);
   pc_cell_prev->insert_next_item(tmp);

   // add this edge as an element's face with local index kk
   scanner->item().set_face_index(kk, edges_counter);
   ...
```

  We should also construct and push the corresponding new edge, relying on the new face constructor as follows:

```
++adj_vertices[ismin];
++n_edges;
edges_.push_back(
   edge(*scan_min, *scan_max, edges_counter++, current_element_index, -1));
```

Note that we now have easy access, through the edge object, to its own index, its left sharing triangle index (and even its "left" sharing triangle address if needed). Note also that by default, and until later update, the "right" sharing triangle index is set to −1.

– if the edge has already been accounted for through its "first" ("left") sharing triangle, then we have to handle the **else**{} alternative to store the "second" ("right") sharing triangle informations, as follows:

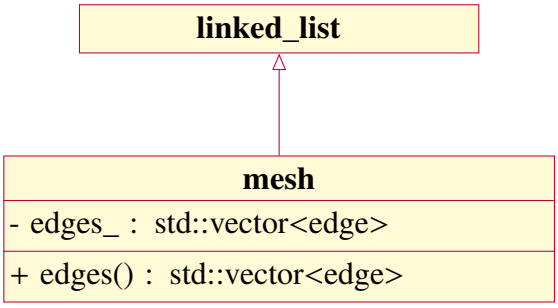
```
else
{
 // thank's to connectivity_data, we can access the current edge index
 // and modify the neighboring triangle index accordingly in std::vector<edge> edges_
 edges_.at(pc_cell_prev->item().edge_index()).neighbor(1) = current_element_index;

 // also, we add this edge index as an element's face with local index kk
 scanner->item().set_face_index(kk, pc_cell_prev->item().edge_index());
}
```

At the end of the execution of build_edges(), we now have populated the following connectivity fields:
triangle::std::array<**int**, 3> faces_,
edge::std::array<**int**, 2> neighbors_,

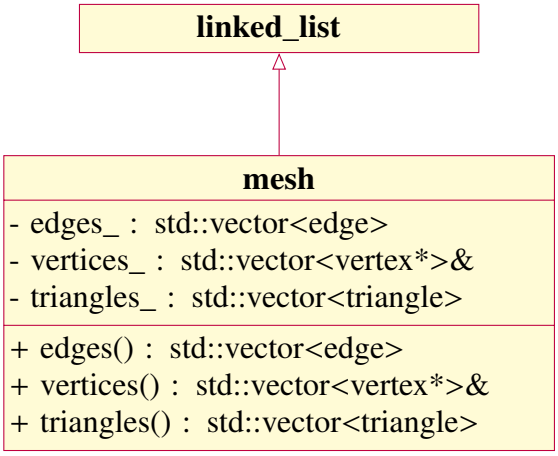
It remains to populate the field triangle::std::array<**int**, 3> neighbors_

Exercice 3. upgrade the mesh class

After Exercice 2, calling the build_edges() method for a given triangulation object generates (more precisely populates) a suitable std::vector<edge> object, which belongs to the current (calling) triangulation instance, in a *composition* class collaboration way.



Of course, other design choices may be defended. By design, we also build our triangulation object from std::vector<vertex*> and std::vector<triangle> objects, by appending successively the vector's elements into the underlying linked_list. The linked_list underlying structure is mostly useful as soon as mesh construction, modification, refinement or agglomeration strategies are required.
For several other situations, a linear access to the mesh triangles may be more efficient and for this reason, it may be useful to also provide an easy access to the previous vectors as follows:



Observe that we choose to use an *aggregation* collaboration strategy between classes (we use references to these new members, for the sake of simplicity and to avoid useless data-copying) for the vertices, as the std::vector<vertex*> already exists independently from the triangulation instance (note however that the local container std::vector<triangle> triangles_ should be preferably built from the triangles coming from the linked_list). **The main consequence is that the** mesh **constructor should be modified accordingly to initialize such reference**:

```
mesh(std::vector<triangle>&, std::vector<vertex*>& vertices);
```

and the associated definition:

```
template <class T>
mesh<T>::mesh(std::vector<triangle>& triangles, std::vector<vertex*>& vertices)
        : linked_list<T>(triangles[0]), n_elements_(triangles.size()),
                                        vertices_(vertices), n_vertices_(vertices.size())
{...
```

Note also that we conveniently delegated several geometrical computations at the triangle level of the hierarchy, by introducing the methods compute_area(), compute_centroid(), compute_faces_length(), compute_outgoing_unit_normals(). As a consequence, we have to sequentially call these methods for all mesh elements, from the triangulation object. This requires to introduce several new methods in the class mesh:

```
int compute_elements_areas();
int compute_elements_centroids();
int compute_faces_lengths();
int compute_faces_outgoing_unit_normals();
```

which only call the corresponding methods at the element level, for each element. We also need a protected member mesh_size_ and a method **double** mesh_size() returning the minimum edge's length. This value may be used later to compute a suitable time-step.

Exercice 4. upgrade again the mesh class

In this Exercice, we aim at populating the member

```
triangle::std::array<int, 3> neighbors_
```

from the connectivity data available in the upgraded edge and triangle objects. This can be done easily with another traversing loop on mesh element: for each element, each face now has a global edge index, and from the knowledge of this edge we can obtain the indices (and even the addresses if stored) of the two sharing triangles (or of the only owning interior triangle if we encounter a boundary face).

We observe that the topology objects std::array<int, 3> neighbors_ are located in the triangle objects but can be populated only through the traversing of a triangulation instance. Hence, we introduce a new method in the mesh class:

```
int populate_faces_neighbors_indices();
```

Let's go !

```
template <class T>
int mesh<T>::populate_faces_neighbors_indices()
{
  for (mesh<T>* scanner = this; scanner; scanner = (mesh<T>*)scanner->p_next())
  {
    auto current_element_index = scanner->item().index();
    for (auto kk = 0; kk < 3; ++kk)
    {
       scanner->item().set_neighbor_index(kk,
       edges_.at(scanner->item().face(kk)).neighbor(0) == current_element_index ?
                        edges_.at(scanner->item().face(kk)).neighbor(1) :
                        edges_.at(scanner->item().face(kk)).neighbor(0));
    }
  }
  return 1;
}
```

Exercice 5. a template class advection_solver

```
class advection_solver
{
protected:
    std::vector<double> u_;
    triangulation& mesh_;

public:
    advection_solver(triangulation& mesh) : mesh_(mesh) {}; // get the mesh
    void set_initial_data(point2d Xc, double amp, double sig); // initialize u_
    void advance(double T_final, double dt, point2d velocity); // advance u_
    // void export_to_VTK(std::string const& filename); // visualize u_
  ...
```

With all the geometrical and topological data in place, we can now focus on solving the PDE and generate the missing piece of code in the construction of our simple FVM method on triangulations. The main steps are the following:

- define the initial condition, on the current triangulation, from a discrete viewpoint,
- handle the issue of boundary conditions,
- produce a time-loop that advances the discrete initial-data from discrete-time to discrete-time, computing the numerical fluxes at each interface (upwind flux) and assembling the local variations through triangles boundaries,
- post-process the results and visualize the corresponding data.

We handle the first and second items simultaneously by considering the following initial-data $u_0(x) := A\exp(-\frac{|\mathbf{x}-\mathbf{x}_c|^2}{2\sigma^2})$ which produces the profile seen on Fig. 2 on the ex.mesh triangulation. Note that this function vanishes (in a machine accuracy meaning) at the domain boundary $\partial\Omega$, which is consistent with the enforced homogeneous Dirichlet boundary condition of (1). We consider a small enough propagation

time to ensure that the solution u does not interact with the boundary of the domain, making the issue of numerical boundary conditions meaningless. This initial-data may be defined into a more general class devoted to the resolution of the PDE, called FVM_advection_solver, with the following members and methods:

- define the method
  void set_initial_data(point2d Xc, **double** amp, **double** sig)
  such that the initial gaussian profile is initially centered at $\mathbf{x}_c$ = Xc = $(0, 0)$, with an amplitude A = amp = 1 and σ = sig = 2,

- define the main method advance which control the time-loop associated with the time-marching algorithm, with a prescribed constant velocity $\mathbf{v} = (1, 0)$ and a time-step computed as δ_t^n = 0.5mesh_size/$|\mathbf{v}|$.

```
while (t < T_final)
{
  // loop on triangles
  for (triangulation* p_scan = &mesh(); p_scan; p_scan = (triangulation*)p_scan->p_next())
  {
     double flux_total = 0;
     // loop on faces
     for (auto kk = 0; kk < 3; ++kk)
     {
        // compute upwind flux
        double vn = velocity * p_scan->item().normal(kk);
        flux_total += ...
        // update the solution
        next_u[tri_index] = u_[tri_index] - (dt / p_scan->item().area()) * flux_total;
     }
   }
   // prepare next time iteration
   u_ = next_u;
   t += dt;
   // outputs and update loop-counter
   if (++step % 10 == 0)
   {
       std::cout <<  ...
   }
}
```

With everything in place, the driver could be written as follows:

```
int main()
{
    std::vector<vertex*> vertices;
    std::vector<triangle> triangles;

    // load mesh data from .mesh file
    std::string meshFile = "ex.mesh";
    bool res = mesh_reader(meshFile, vertices, triangles);

    triangulation mesh(triangles, vertices);

    advection_solver gaussienne(mesh);

    point2d velocity(1., 0.);
```

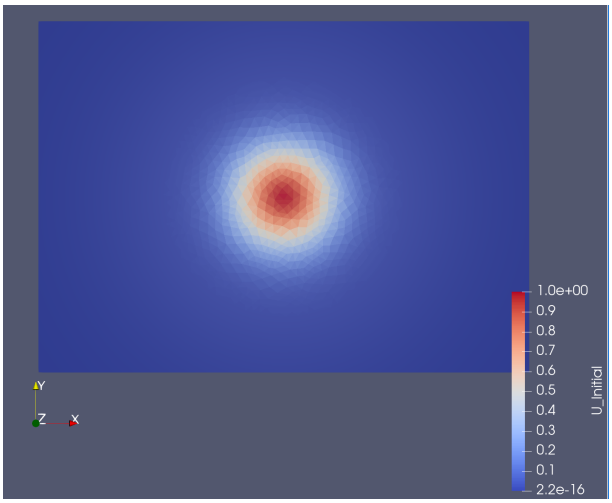


FIGURE 2. Initial data as a colormap

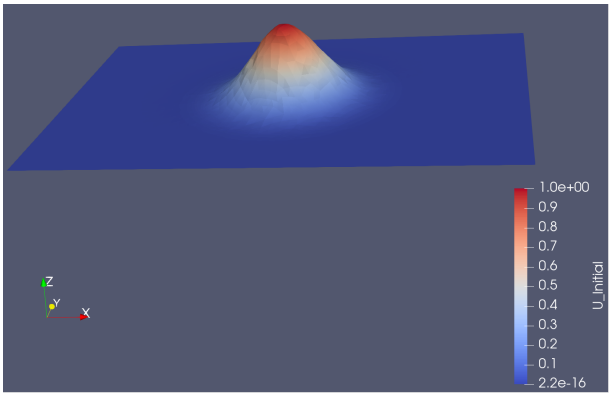


FIGURE 3. Initial data with "warp by scalar" filter

```
    double Tmax = 3.;

    gaussienne.advance(Tmax, velocity);

    return 0;
}
```

Exercice 6. Write the solution's values in (ASCII) VTK format

When using triangulations, and more generally unstructured meshes, the visualization of the solutions may be challenging as one has to put in relation the computed discrete values (and the mathematical notions of freedom-degrees or polynomial moments) with the geometrical and topological entities used to define the mesh. Depending on the complexity of the problem, this may require additional data processing, to produce convenient and informative graphics.

For 2d or 3d computations, such visualization is generally delegated to some specialized software. Sometimes, software dedicated to mesh generation (like gmsh or medit) also provide some features to visualize the data on the corresponding meshes. But the more generalized way to go is to use industry-standard data formats, which may vary depending on the applications, the software etc ...

For this practical application, we use the Visualization data ToolKit (VTK) format (in ASCII format) to export the solution together with the mesh informations, and the software Paraview to load and visualize the picture.

Here is a simple method that may be included into your class advection_solver, or used as the first step for a new data_visualization class.

```
void export_to_VTK(std::string const& filename)
{
  std::ofstream file(filename);
   if (!file.is_open()) return;
     file << "# vtk DataFile Version 3.0" << std::endl;
     file << "Advection Initial Condition" << std::endl;
     file << "ASCII" << std::endl;
     file << "DATASET UNSTRUCTURED_GRID" << std::endl;

     // 1. write the vertices
     file << "POINTS " << mesh_.n_vertices() << " double" << std::endl;
     for (const auto& v : mesh_.vertices())
     {
         file << v->x() << " " << v->y() << " 0.0" << std::endl;
     }
     // 2. write the cells - format : CELLS [nb_cellules] [nb_total_entiers]
     // for each triangle : 3 + index1 + index2 + index3 (4 ints)
     file << "CELLS " << mesh_.n_elements() << " " << mesh_.n_elements() * 4 << std::endl;
     for (triangulation* scanner = &mesh_; scanner; scanner = (triangulation*)scanner->p_next())
     {
         file << "3 " << scanner->item().vertex(0)->index() << " "
                      << scanner->item().vertex(1)->index() << " "
                      << scanner->item().vertex(2)->index() << std::endl;
     }
     // 3. cells type (5 for triangle)
     file << "CELL_TYPES " << mesh_.n_elements() << std::endl;
     for (auto i = 0; i < mesh_.n_elements(); ++i)
     {
         file << "5" << std::endl;
     }
     // 4. values at cells
     file << "CELL_DATA " << u_.size() << std::endl;
     file << "SCALARS U_Initial double 1" << std::endl;
     file << "LOOKUP_TABLE default" << std::endl;
     for (double val : u_)
     {
         file << val << std::endl;
     }
     file.close();
 }
};
```

Exercice 7. An improved edge-loop FVM implementation

One may observe that in formulation (6), the interior numerical fluxes are computed twice. Hence, a more efficient implementation may rely on a edge-centered strategy. Starting from (6) and summing over T one has:

$$\sum_{T\in\mathcal{T}_h} \overline{u}_T^{n+1} = \sum_{T\in\mathcal{T}_h} \overline{u}_T^{n} - \sum_{T\in\mathcal{T}_h}\sum_{F\in\mathcal{F}_T} \frac{\delta_t^n}{|T|}|F|\,\mathcal{G}_{TF}^n$$
$$= \sum_{T\in\mathcal{T}_h} \overline{u}_T^{n} - \delta_t^n \times (\text{Sum of the net fluxes across the boundaries of the elements})$$

and

$$\sum_{T\in\mathcal{T}_h} \overline{u}_T^{n+1} = \sum_{T\in\mathcal{T}_h} \overline{u}_T^{n} - \delta_t^n \left[\sum_{F\in\mathcal{F}_{int}} \frac{|F|}{|T|} \underbrace{(\mathcal{G}_{TF}^n + \mathcal{G}_{T',F}^n)}_{=0} + \sum_{F\in\partial\Omega} \frac{|F|}{|T|}\mathcal{G}_{TF}^n \right]$$

where:

- $\mathcal{F}_{int}$ represents the set of internal faces shared by two adjacent elements T et T'.
- the term $(\mathcal{G}_{TF}^n + \mathcal{G}_{T',F}^n)$ vanishes because, by the construction of finite volume schemes, the flux leaving a cell is strictly equal to the flux entering its neighbor (consistency of the numerical flux).
- by notation abuse, $\partial\Omega$ gathers the faces located on the domain boundary, which are the sole contributors to the global variation of the quantity u.

Note in passing that if 'nothing happens' at the domain boundary (as in the particular situation computed in our example), we recover the global conservation property:

$$\overline{u}_\Omega^{n+1} := \sum_{T\in\mathcal{T}_h} \overline{u}_T^{n+1} = \sum_{T\in\mathcal{T}_h} \overline{u}_T^{n} - \underbrace{\sum_{F\in\partial\Omega} \delta_t^n \frac{|F|}{|T|}\mathcal{G}_{TF}^n}_{=0} =: \overline{u}_\Omega^{n}.$$

Thus, in our code, we do not use the theoretical double sum; we iterate only once over each edge to update the two adjacent mesh elements. The algorithmic formulation that highlights this loop is as follows:

$$\text{(7)} \qquad \begin{cases} \forall T \in \mathcal{T}_h, \quad \overline{u}_T^{n+1} = \overline{u}_T^{n} \\ \textbf{for all internal face } F \in \mathcal{F}_{int} \text{ between } T \text{ and } T' : \\ \quad \begin{cases} \overline{u}_T^{n+1} \leftarrow \overline{u}_T^{n+1} - \dfrac{\delta_t^n}{|T|}|F|\mathcal{G}_{TF}^n \\ \overline{u}_{T'}^{n+1} \leftarrow \overline{u}_{T'}^{n+1} + \dfrac{\delta_t^n}{|T'|}|F|\mathcal{G}_{TF}^n \end{cases} \\ \textbf{for all boundary face } F \in \partial\Omega \text{ belonging to } T : \\ \quad \overline{u}_T^{n+1} \leftarrow \overline{u}_T^{n+1} - \dfrac{\delta_t^n}{|T|}|F|\mathcal{G}_{TF}^n \end{cases}$$

Observe that we now have unique flux computation and associated conservation: $\mathcal{G}_{TF}^n$ is computed only once per face, which by definition guarantees that the flux leaving T is equal to the flux entering T'. Moreover, this edge-based loop approach has a complexity of $|\mathcal{F}_h|$, which is optimal for complex unstructured meshes.

Some data related to edges orientations with respect to mesh elements are needed to implement this optimized loop on edges. In practice, one should introduce an arbitrary-oriented unit normal vector associated with each edge and compute the associated edge-related interface fluxes. Then, one should compare the orientations of such edge normals with the triangles outgoing normals, for sharing elements T and T', to decide wether the corresponding interface fluxes should be added or subtracted to the local balance on the considered sharing elements.