UNIVERSITÉ DE MONTPELLIER

FACULTÉ DES SCIENCES DE MONTPELLIER

Master de Mathématiques

Parcours *"Modélisation et Analyse Numérique"*

Programmation I - HAX816X - 2024/2025

HomeWork - Finite-Volume Method for Burgers equation

In this HomeWork, we introduce a (simple) design relying on (simple) classes to implement a first (and again simple) Finite-Volume (FV) scheme that (approximately) solves the time-dependent Burger equation (and more generally scalar hyperbolic conservation laws, for advanced students, see last exercice).

Of course, you (certainly) already have implemented such schemes in Matlab or Python using the enjoyable data-structures already available in these interpreted languages. However, we show that even for such a simple problem, the generic features of C++ programming may lead to interesting implementations, which can be easily generalized to more general problems. From a learning viewpoint, this is the opportunity to begin to introduce some classes collaboration strategies.

We focus on the following Initial Boundary Value Problem (IBVP for short):

$$
(1) \qquad \begin{cases} \partial_t u(t,x) + \partial_x F(u(t,x)) &= 0, & \forall x \in [0,L], & \forall t \in [0,T], \\ u(0,x) &= u_0(x), & \forall x \in [0,L], & \\ \partial_x u(t,0) = \partial_x u(t,L) &= 0, & & \forall t \in [0,T], \end{cases}
$$

with $F(u(t,x)) = \frac{1}{2}u^2$ for the Burgers (nonlinear) equation. The reader is referred to UE "Analyse Numérique II" for the detailed study of such problems.

In general FV methods, the PDE is solved numerically on a discrete grid, which is a partition of $[0,L]$ into a finite set of N_e non-overlapping intervals:

$$
[0,L] = \bigcup_{i=1}^{Ne} C_i, \qquad \text{where} \qquad C_i := [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}],
$$

with the convention $x_{\frac{1}{2}} = 0$, $x_{N_e+\frac{1}{2}} = L$, $x_{i+\frac{1}{2}} = x_{i-\frac{1}{2}} + \delta x$ and $\delta x = L/N_e$.

We also define $N_v := N_e + 1$ the corresponding number of vertices in the sequence $(x_{i+\frac{1}{2}})_{0 \le i \le N_e}$ used to define this uniform grid. We highlight that $x_{i+\frac{1}{2}}$ is therefore also an *edge* between the cells C_i et C_{i+1}.

In a similar way, we introduce a discretization of the time interval $[0,T]$ by introducing a sequence of discrete time-values $(t_0 = 0, t_1, \ldots, t_{N_t-1}, t_{N_t} = T)$, where we set $t_{n+1} = t_n + \delta t_n$, δt_n being adaptively computed using the so-called CFL condition, with details given below.

Let denote, for $1 \le i \le N_e$, the sought FV approximations of the solution on the cell C_i at time t^n:

$$
(2) \qquad \bar{u}_i^n := \frac{1}{\delta x} \int_{C_i} u(t^n, x_i)\, dx,
$$

and, knowing the initial approximations $(\bar{u}_i^0)_{1\leq i\leq N_e}$, which is obtained by the computation of such values from the initial data u_0, your goal is therefore to compute the subsequent discrete approximations $(\bar{u}_i^n)_{1\leq i\leq N_e}$, from discrete time t^n to discrete time t^{n+1}, through the general conservative FV methods formula:

$$\bar{u}_i^{n+1} = \bar{u}_i^n - \frac{\delta t_n}{\delta x}\left(\mathcal{F}_{i+\frac{1}{2}}^n - \mathcal{F}_{i-\frac{1}{2}}^n\right), \qquad 1 \leq i \leq N_e,$$

where the sequence $\mathcal{F}^n := \left(\mathcal{F}_{i+\frac{1}{2}}^n\right)_{0\leq i\leq Ne}$ is called (discrete) *numerical flux function*.

From this, defining a FV method consists in providing a way of computing this numerical flux family. Here, we propose to use the so-called Lax-Friedrichs numerical flux defined as follows:

$$\mathcal{F}_{i+\frac{1}{2}}^n := \frac{1}{2}\left(F(\bar{u}_i^n) + F(\bar{u}_{i+1}^n)\right) - \frac{1}{2\lambda}\left(\bar{u}_{i+1}^n - \bar{u}_i^n\right),$$

where $\lambda = \max_{1\leq i\leq N_e} |F'(\bar{u}_i^n)|$. Another way of considering the scheme is to introduce the FV *residual* as follows:

$$\mathcal{R}_i^n := -\frac{1}{\delta x}\left(\mathcal{F}_{i+\frac{1}{2}}^n - \mathcal{F}_{i-\frac{1}{2}}^n\right),$$

such that the FV scheme is written:

$$\bar{u}_i^{n+1} = \bar{u}_i^n + \delta t_n \mathcal{R}_i^n, \qquad 1 \leq i \leq N_e. \tag{3}$$

There are many strategies to define a FV numerical algorithm and implement it. Within a POO strategy, one may consider the introduction of several C++ objects, mimicking the associated mathematical notions, and to consider the algorithm as a definition of these object's interactions.
More precisely, we investigate in the following the construction of five important classes: mesh, field, residual, time_loop, result_writer and define some basic collaboration strategies between instances of such objects, in order to compute the approximate solution in a way which may allow flexibility and future evolution (generality, ease for maintenance)

Please, keep in mind that we absolutely do not focus on efficiency in this HomeWork. Our goal is to introduce you to some grounded design rules for evolutive scientific computing codes and start from the beginning, with humility. More to learn next year.

To give you an idea of the logical sequence of operations that the main.cpp program should display, here is a copy of my main file. Your goal is to give sense to these instructions, in order to compute and plot the approximated solution of the model at final_time.

```
int main()
{
    double x_left = 0.; // needed initialization parameters
    double x_right = 15.;
    size_t Ne = 1500;
    double final_time = 6.;

    mesh_1d<double> msh(x_left, x_right, Ne); // mesh construction
    field<double> field(GAUSS_PULSE, msh); // initial field defined and initialized on mesh
    output_writer<double> out_stream(msh, "GAUSS_PULSE"); // initialize writer

    out_stream.write_solution(field, "initial"); // write initial field

    time_loop<double> loop_in_time(msh, final_time); // initialize time-loop
    loop_in_time.run(field); // launch time-loop

    out_stream.write_solution(field, "final"); // write final field}
```

Exercice 1. the mesh_1d class

In the $d = 1$ case (one dimension), the notion of "mesh" is trivial and a cell is just a segment, with two faces (two vertices). Additionally, the connectivity relations between the mesh elements and vertices are obvious (the first element is made of the two first vertices, the second element is in the second place ... and so on ...). That being said, it is still interesting to create a real mesh class which gathers and ease all the accessors and informations related to nodes, edges, faces and cells, as this is the usual way a mesh is organized in a multi-dimensional framework, and it allows to follow the path of more general numerical codes.
Note that when $d > 1$, the construction of the mesh connectivity informations is generally difficult and relies on theoretical notions coming the mathematical theory of graphs.
We also highlight that when $d = 1$, it is easy to have a mesh that exactly coincides with the geometry of the problem (both are segments): it is generally not the case when $d > 1$ as one has to generate some polygonal partition of non-polygonal domains, introducing some geometrical approximation errors. Curved meshes exist, with additional mathematical and technical difficulties.

Here are some guidelines to create your mesh class:

(1) the mesh class should have **protected** member to store the vertices coordinates and the cells centers coordinates. The size of this array is not known at compilation time: it has to be dynamically handled and we suggest to use our dynamic_vector class or the std::vector class:

```
template <typename T>
class mesh_1d{
public:
    mesh_1d(T x_left, T x_right, size_t n_cells);
    ~mesh_1d() {};
    inline const size_t n_vertices() const { return n_vertices_; };
    inline const size_t n_cells() const { return n_cells_; };
    inline const T dx() const { return dx_; };

    inline T xc(size_t ii) const { return cells_centers_[ii]; };
    void print();

protected:
    std::vector<T> vertices_coordinates_; // composition
    std::vector<T> cells_centers_; // composition
    size_t n_vertices_{0};
    size_t n_cells_{0};
    T dx_;
};
```

(2) complete the definition the constructor to populate the vertices_coordinates_ and cells_centers_ vectors, and initialize all the **protected** members accordingly.

(3) complete the definition of the print function to check the meshes definitions on screen.

Exercice 2. the field class

One of the cornerstone of scientific computing is to provide suitable and efficient date-structures to store the data, possibly submitted to intensive computations and memory access. For a 1d problem, the linear indexing identifies with the linear memory storing of values in a one-dimensional array. Indeed, the sought unknowns $(\bar{u}_i^n)_{1\leq i\leq N_e}$ are scalar real values, distributed along a linear mesh. Thus, it absolutely makes sense to store the values in a one-dimensional array (hence, a dynamic_vector or a std::vector as the memory allocation should still be dynamically performed).

It also makes sense to define a simple *wrapper* that embed the vector in the context of the storage of our FV unknowns: we call this wrapper the field class, which aims at successively storing the values of the field unknowns, form time to time, in a time-loop.

Here is the declaration of such a field class:

```
template <typename T>
class field{
public:
    field(int test_label, mesh_1d<T> const& msh);

    inline const size_t n_cells() const { return n_cells_; };
    inline const std::vector<T>& operator()() const { return values_; };
    inline double operator()(size_t ii) const { return values_[ii]; };

    field<T>& operator+=(residual<T> res);

protected:
    std::vector<T> values_; // composition
    size_t n_cells_{0};
};
```

Here are the steps to create your field class:

(1) observe that the class field has a **protected** member which is himself an instance of a template class (here it it std::vector<T> but it may also be our dynamic_vector<T> class). The existence of this member is strictly ruled by the existence of the containing instance of field: their lifespans exactly coincide. Such a class relation is called *class composition*,

(2) define the constructor to initialize the values_ member with the values of the initial condition $(\bar{u}_i^0)_{1\leq i\leq N_e}$, relying on the selector test_label. Several initial conditions may be chosen, you may find some ideas in your previous lessons. Here, as a starting point, I suggest to begin with a "Gauss pulse" defined as follows (and displayed at the end of this file):

```
double gauss_pulse(double xx)
{
    return exp(-pow((xx-5.),2));
}
```

with a computational domain $[x_{\text{left}}, x_{\text{right}}] = [0, 15]$, as in the main example.

To this end, note that the proposed FV method is only first-order accurate in space and time (again, read again your previous lessons), and as a consequence the initialization does not deserve to compute an accurate numerical integral for (2) with high-order quadrature formula, as we simply have:

$$\bar{u}_i^0 = \text{gauss\_pulse}(x_i^c) + O(\delta x),$$

where x_i^c is the coordinate of the center of C_i,

(3) observe that this class field has to interact with the class residual<T>, which is not defined yet. This appears in the method field<T>& **operator**+=(residual<T> res). We suggest to let this method undefined for the time being, and to define it later, when the residual class will be introduced. As there is no possibility to add a

```
#include "class_residual.hpp"
```

header line at the beginning of the current class_field.hpp file (the residual class is not even thought for the time being), the solution is to add a preventing declaration, by adding the lines:

```
template <typename T>
class residual;
```

before the declaration of the field class. This allows the compiler to be aware that a template class names residual is defined "somewhere" else and should not worry about it ... for the time being.

Exercice 3. the time_loop class

The cornerstone of this code is the time_loop object. Indeed, it has to get the mesh_1d and the initial field as arguments, together with the required parameters (like the final_time), so that the instruction loop_in_time.run(field) seen in the main file launches and achieves the global computation with the FV method, producing the final field corresponding to the soluation approximated at final_time. Here is the suggested declaration:

```
template <typename T>
class time_loop{
public:
    time_loop(mesh_1d<T> const& msh, T final_time) : mesh_(msh), final_time_(final_time),
        residual_(residual<T>(msh)) {};
    ~time_loop() {};

    inline const T final_time() const { return final_time_; };
    void run(field<T>& uh);
    void compute_dt(field<T>& uh);

protected:
    size_t iteration_counter_{0};
    T dt_{0.};
    T physical_time_{0.};
    T final_time_{0.};
    bool last_iteration_{false};
    double cfl_number_ = 0.5;

    residual<T> residual_; // class composition
    mesh_1d<T> const& mesh_; // class aggregation
};
```

The purpose of this class is manyfold, and explained in the following items:

- the time_loop class contains a residual member. The lifespan of this residual has no reason to exceed the lifespan of the time_loop instance that owns it. So, it is legit to use a *class composition* strategy.
  On the other hand, any time_loop instance also has to interact with the global mesh instance associated with the computtaional domains, and this mesh has already been initialized before the time_loop. For the mesh, the legit relation is not *class composition* but *class aggregation*: we

only want time_loop to have a reference (or a pointer) to the related mesh instance. Thus, the **protected** member:

```
mesh_1d<T> const & mesh_;
```

is not a mesh_1d member, but a *reference* to a mesh_1d which lives outside the time_loop instance. It is important to know that such a particular member **has to be initialized in the constructor *initialization list***. The *class aggregation*, strategy is mostly characterized by the fact that the lifespan of these objects are not related (here, the mesh outlives the time_loop).

- the method compute_dt should take the current field as argument, containing the $(\bar{u}_i^n)_{1\leq i\leq N_e}$ values, and use these values to compute the time-step, with the stability CFL-based fomula:

$$\delta t_n := \text{CFL\_number}\frac{\delta x}{\max|F'(\bar{u}_i^n)|},$$

where CFL_number is set to 0.5,

- the **void** run(field<T>& uh) method should launch the time-loop on a **while**() structure, checking that the current physical_time (which should be updated at each time iteration) is not greater than final_time.
At each time-iteration, one should update a residual object, storing the $\left(\mathcal{R}_i^n\right)_{1\leq N_e}$ values, with an instruction like:

```
residual_.assemble_from_field(uh);
```

which is detailed in the next Exercice dedicated to the residual class. With such an updated residual available, one can update the field:

```
uh += dt * residual_;
```

mimicking formula (3). Note that such an instruction requires the **operator***$*$ to be overloaded between a type T and residual_ object, and **operator**+= to be overloaded to sum a field and a residual object as in the mathematical formulation (3),

- note that it is possible to simply adjust the final time-step dt so that the exact value final_time is reached:

```
if (physical_time + dt > final_time())
{
    dt = final_time() - physical_time;
     last_iteration_ = true;
}
```

Exercice 4. the output_writer class

I suggest to write the initial data and the solution at final_time in two different files named GAUSS_PULSE_initial.txt and GAUSS_PULSE_final.txt (of course, more snapshots of the solution during the computation can be written in the time-loop, in order to produce an animation of the pulse time-evolution. But as a minimal requirement to check the validity of the code, let focus at least on these two snapshots). Here is the declaration of such a output_writer class:

```
template <typename T>
class output_writer{
public:
    output_writer(mesh_1d<T> const& msh, string filename):mesh_(msh), radical_name_(filename){};

    void write_solution(field<T> solution, string loop_counter);
    inline const mesh_1d<T>& mesh() const {return mesh_; };

protected:
```

```
    mesh_1d<T> const& mesh_; // aggregation
    string radical_name_;
};
```

Your goal is to define the missing method, so that the command out_stream.write_solution(field, "initial"); results in writing the initial field values, together with the abscissa of the corresponding cells centers for plotting purpose. Examples of the expected file GAUSS_PULSE_initial.txt and GAUSS_PULSE_final.txt obtained with my own code and the parameters of the initial example are provided for comparison purpose. Using these files, here are the initial and final approximated solutions:

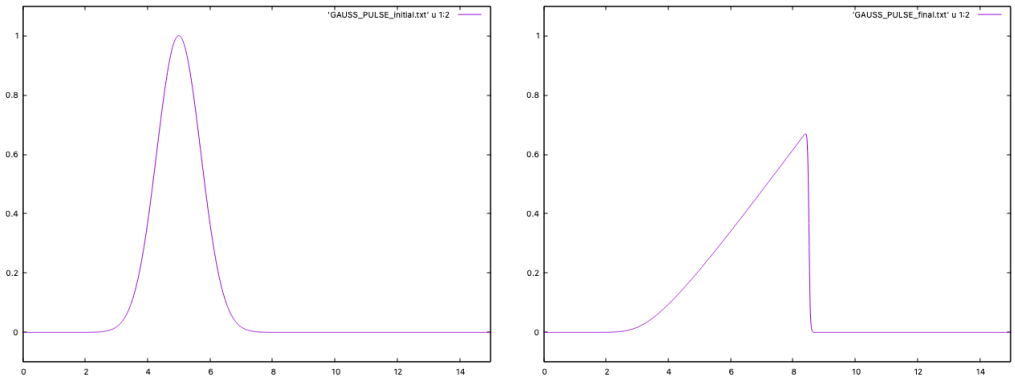

FIGURE 1. Initial and final solutions for the GAUSS_PULSE test with Burgers equations

Exercice 5. the residual class

The residual object may be seen as a way of injecting the definition of the FV scheme into the time_loop occurence. As already said, it is only used in time_loop, and should not outlive this owning container. Here is the corresponding declaration:

```
template <typename T>
class residual{
public:
    residual(mesh_1d<T> const& mesh):mesh_(mesh){};

    inline void clear() {values_.clear();};
    void assemble_from_field(field<T> const& uh);

    const mesh_1d<T>& mesh() const { return mesh_; };
    inline const size_t n_cells() const { return mesh_.n_cells(); };

    T operator()(size_t ii) const { return values_[ii];};

    template <typename S>
    friend residual<S> operator*(S lambda, residual<S> const& res);

protected:
    std::vector<T> values_; // composition
    mesh_1d<T> const& mesh_; // aggregation
};
```

Here are some comments:

(1) it is in the residual instance that one computes the residual values at discrete time t^n, relying on the provided field values corresponding to the solution $(\bar{u}_i^n)_{1\leq i\leq N_e}$. As a consequence, it is exactly the place where the definition of the *numerical fluxes* should be put. The Lax-Friedrich flux

formula is given as follows:

$$F(u,v) = \frac{1}{2}\left(F(v) + F(u)\right) - \frac{1}{2\lambda}\left(v - u\right),$$

where $\lambda := \max |F'(\bar{u}_i^n)|$, which straightforwardly translates into:

```
template <typename T, typename Func>
T flux_LF(T const& ul, T const& ur, T max_vel, Func flux)
{
    return 0.5*(flux(ul) + flux(ur)) - 0.5/max_vel*(ur-ul);
}
```

(2) be aware that you have to take care of boundary conditions in the computation of the numerical fluxes, at the two extremities : we suggest to enforce $\bar{u}_0^n = \bar{u}_1^n$ and $\bar{u}_{N_e+1}^n = \bar{u}_{N_e}^n$ whenever needed,

(3) note that one of the **template** parameters (**typename** Func) refers to the physical flux function F and may therefore be replaced by the flux function for the Burgers equation. But this function may possibly vary later, if another equation is considered, while we would still be willing to consider the Lax-Friedrichs flux function. This is the reason why we suggest to pass it as a **template** parameters. A possible definition of such a function to be passed as **template** parameters during computation is:

```
template <typename T>
T flux_Burgers(T const& u)
{
    return 0.5*u*u;
}
```

(4) the **friend** function **friend** residual<S> **operator***(S lambda, residual<S> **const**& res) aims at allowing to give a meaning to the instruction uh += dt * residual_ already seen in time_loop. It should multiply the values_ vector component by dt and return the resulting modified residual object.

These guidelines should not be blocking constraints, but rather helping/orienting advises. Feel free to ask questions, to ask for further details or explanations, or to introduce slight (and reasonable) changes that fit more closely to your own ideas. The goal is to make you have some "long-term" reflections about the program construction.
From scratch, it took me one hour and a half to produce the solution program (approx. 350 lines) that serves as a ground for this HomeWork. I estimate that approximatively ten hours of serious work should be enough to achieve a working program.

Exercice 6. more

For more advanced students, let consider the following perspective: in a second time, we'd like a second version of our code to handle any arbitrary scalar conservation law of the initial form, with a given physical function F (for instance the advection (linear) equation $F(u) := \beta(t,x)u(t,x)$ or the Buckley–Leverett equation for instance.
Additionally, we'd like to offer to the user the possibility to vary the numerical flux function (let say, the Godunov flux function when available or the Lax-Wendroff flux for instance) at execution time, in the main program.
Of course, such a flexibility can be achieved by multiplying the source codes, the functions and using **switch** and/or **if** /**else** instructions. But it is also possible to achieve such a result in some more elegant ways relying on **template**s and/or class inheritance.
What would be your own solutions to these requirements ?