Sorbonne Université
Implementation of finite element methods
(MU5MAM30)

Some project proposals

Instructions

Projects can be tackled in groups of at most three students (but different groups can work independently on the same subject). Reports should be brief and to the point, and ideally typeset in Latex or Markdown. They must be accompanied with source code and build instructions. They must be uploaded on the git repo in the Project directory or sent to me by email, in the latter case in a single archive (zip or equivalent) with basename format `name_surname-MU5MAM30-project`. **Deadline for upload is February 21st 2025**. Appointments (30min slot, on an individual basis even if working in group) must be taken for a project presentation between January 6th and February 28th 2025, by e-mail at `didier.smets@sorbonne-universite.fr`. It is accepted but not needed to prepare slides for the presentation, I will have read your code and report in advance, you will explain me the difficulties and achievements, and I will ask you some questions related to your submission to evaluate your scientific autonomy.

In case of guidance needed, for the choice or the realization of the project, contact me by e-mail or show-up in my office in 16-26 325. Proposing an alternate project is also accepted, but you must send me the proposal for agreement first.

Project 1 : Adding a boundary constraint

The goal of this project is to modify the Navier-Stokes solver which we have devised in class in order to tackle surface meshes with a boundary, with the condition that the velocity u of the fluid is tangent to the boundary.

In the vorticity/stream function framework that we have adopted, the velocity u of the fluid and the stream function Ψ were related by the equation

$$u = \nabla^{\perp}\Psi.$$

A natural set-up in the presence of boundaries is to assume a non-penetrating condition, that is

$$u \cdot n = 0 \text{ on } \partial\Omega,$$

where here n denotes the normal to the boundary. This can be rewritten as

$$\nabla^{\perp}\Psi \cdot n = 0, \qquad \text{or equivalently} \qquad \nabla\Psi \cdot \tau = 0,$$

where here τ is a tangent vector to the boundary. The latter implies that Ψ is constant on (each connected component of) $\partial\Omega$. In the project you may assume that $\partial\Omega$ is connected (as is the case e.g. for Ω a half-sphere surface mesh, hence only slightly modifying our class framework, or even more simply a 2D square grid). In that case, since Ψ is only also globally defined up to a fixed constant (adding a constant to Ψ does not modify the physical quantity u), you can assume that $\Psi \equiv 0$ on $\partial\Omega$. Therefore, for the Poisson solver part of the resolution, you will have

$$\Delta\Psi = \omega \text{ on } \Omega, \qquad \Psi = 0 \text{ on } \partial\Omega.$$

Bonus/Suggestion if you wish to dig further : Try discussing the boundary conditions for ω that we are implicitly assuming here by not modifying the weak formulation for ω with respect to the case without boundary. Other more physically relevant conditions might impose the value of the tangential part of the velocity (e.g. a no-slip condition corresponding to $u \cdot \tau = 0$, which in addition to the impermeability condition yields $u = 0$), or a so-called Navier type condition. Both of these are mathematically more challenging (non local) in the context of the vorticity-stream formulation.

Coding suggestion: You might clone (some of) my files in the base `include` and `src` directories of the git repository into your own Student directory and then modify them there (e.g. at least `src/fem/navier_stokes.cpp`). That will allow you to test your modification code with graphical feed-back[1] at minimal coding cost.

[1] That will require cloning the `extern` and `shaders` directories too.

Project 2 : Elimination tree and direct sparse Cholesky solver

For this project you will write your own Cholesky factorization code

$$A = LL^T$$

for SPD sparse matrices A in the CSR format.

One key step of this factorization in the framework of sparse matrices is to determine in advance which matrix entries in L might end-up being non zero (the fill-in), because these may be much more than in A and it is necessary or at least useful to "prepare" L (reserve memory etc) before filling it. This phase is usually called the *symbolic* phase, because the potential non zero entries are located and counted in each row, but are not yet computed.

Since the theoretical background behind the process requires a bit of thinking, you are suggested to use as a reference the book *Algorithms for Sparse Linear Systems* by J. Scott & M. Tuma (in open access, see e.g. here for download). Mostly relevant to us here are sections 4.2, 4.3, (5.1) and 5.2. In particular you will review the notion of *elimination tree*, and you will directly translate algorithms 4.2 and 4.3 (for the symbolic phase) and then 5.7 (for the actual factorization) into code. For the factorization step, it is important to avoid computing elements of L which were not marked as potential non zeros in the symbolic phase. This last phase remains the most computationally expensive though.

Bonus/Suggestion if you wish to dig further : Already after the symbolic phase, you may test using your code (e.g. computing the amount of fill-in) the importance of the ordering of the unknowns (recall that in the case of FEM P1 mass or stiffness matrices, unknowns simply correspond values at mesh vertices, so it is just vertex reordering). For the specific and relatively simple case of the cube mesh, you can try to reorder vertices according to a nested dissection strategy[2] "by hand" (no need to revert to complicated graph partitioning algorithms) : take all the vertices on the edges of the cube as first separator set, and then divide the hence formed 6 remaining square faces recursively and alternatively into one or another direction (for a 2D square grid you would just cut it into two alternatively in the horizontal and vertical directions; in 3D you simply cut it in the direction in which it is presently the longest).

[2] as discussed in the before last lecture, and in section 8.4 of the book suggested above

Project 3 : A (convex) nonlinear elliptic problem

Let Ω be a bounded domain in $\mathbb{R}^2$ and let $f : \Omega \to \mathbb{R}$. The area of the graph of f is given by

$$A(f) := \int_\Omega \sqrt{1 + |\nabla f|^2}.$$

The graph of f is said to be a minimal surface / minimal graph if

$$A(f) \leq A(g) \text{ for any } g : \Omega \to \mathbb{R} \text{ s.t. } f = g \text{ on } \partial\Omega,$$

in other words if its area is minimal among all graphs assuming the same boundary curve. A minimal graph satisfies the minimal surface equation

$$\operatorname{div}\left(\frac{\nabla f}{\sqrt{1 + |\nabla f|^2}}\right) = 0 \text{ on } \Omega,$$

which is nothing but the Euler-Lagrange equation associated to $A(\cdot)$.

Given a planar 2D triangular mesh Ω_h, and piecewise affine function f over that mesh (hence representable using a P1 Lagrange finite element basis), present and code an algorithm that will compute/converge to the (unique) minimal graph f_* in that same finite element space and assuming the same boundary values as f.
Hint : Note that the set of admissible candidates is an affine subspace of the P1 Lagrange space over Ω_h, and the functional $A(\cdot)$ is convex on that set. Any reasonable gradient descent method should therefore converge.

Bonus/Suggestion if you wish to dig further : In nature minimal surfaces (e.g. soap films) need not be graphs. The mathematical set-up required to study the so-called Plateau's problem (finding a minimal surface given a boundary curve) is therefore somewhat more complex, but the resulting Euler-Lagrange equation is simpler since it amounts to the homogeneous linear Poisson equation $\Delta f = 0$. It comes with additional nonlinear constrains though, and besides f is no longer a scalar function (above the graph was $z = f(x, y)$, in the parametric formulation $f(x, y, z) = (f_1(x, y), f_2(x, y), f_3(x, y))$ is a vectoru function. Solutions (they need no longer be unique) may still be searched for using a P1 finite element methods, you might for example try to reproduce the strategy explained in that paper, in section 5.

Project 4 : Using existing software

Understanding the math and the difficulties/challenges associated to the implementation of finite element methods is important in order to build on solid ground. Since the subject is broad, it is also helpful if not important to be able to reuse (and potentially improve or make evolve) existing code bases. For this last subject proposal, the goal is to reproduce the exact same 2D Navier-Stokes solver on a sphere mesh that we coded from scratch, but only using third party libraries/software for each step (meshing, build-up of matrices and solution of linear systems). You might choose one among the following well established open source ones :

- **FreeFem** `https://freefem.org`
  More a software than a library, it comes with its own scripting language leading to short single file scripts. In particular, you won't write any C/C++ code if choosing it.
- **FEniCS** `https://fenicsproject.org`
  Or it's DOLPHINx Python wrapper, if you also wish to avoid C/C++.
- **Deal.II** `https://www.dealii.org`

Demos and tutorials exist for all three, they might be easier to begin with w.r.t. reading their documentation. The first one, FreeFem, is developed here at SU.