Navier-Stokes solver

Dongjiao Hong, Lingxi Song

February 2025

1 Introduction

The goal of this project is to modify the Navier–Stokes solver developed in class to handle surface meshes with a boundary, under the condition that the fluid velocity u remains tangent to the boundary.

We analyze two different boundary conditions and explain why the **Non-penetrating** and **No-slip + Non-penetrating conditions** do not alter the fundamental FEM structure but necessitate modifications in the stream function calculation and time-stepping.

2 Navier-Stokes Equations

2.1 Navier-Stokes equations

The Navier-Stokes equations describe the motion of a fluid and are a set of partial differential equations. In our project, we consider the **incompressible Navier-Stokes equations**, which are applicable to low-speed flows where density remains nearly constant (e.g., most liquids and low-speed gases). The equations are given by:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{1}$$

where:

- $\mathbf{u} = (u, v, w)$ is the velocity field,
- p is the pressure,
- ρ is the fluid density,
- ν is the kinematic viscosity,
- $\mathbf{f}$ represents external forces (e.g., gravity),
- $(\mathbf{u} \cdot \nabla)\mathbf{u}$ is the nonlinear convective term

2.2 The Vorticity-Stream Function Formulation

In 2D incompressible flows, we can introduce the **stream function** ψ and the **vorticity** ω, where $\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$ is the vorticity. ψ is the stream function, with: $u = \frac{\partial \psi}{\partial y}, \quad v = -\frac{\partial \psi}{\partial x}$

Thus, the vorticity-stream function formulation of the Navier-Stokes equations is given by:

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \nabla)\omega = \nu \nabla^2 \omega + f_\omega \tag{2}$$

$$\Delta \psi = \omega \tag{3}$$

where f_ω represents any external forcing in terms of vorticity, for simplicity, we assume it is equal to zero.

1. The stream function is obtained by solving the **Poisson equation**, which corresponds to equation (3).
2. The vorticity is obtained by solving the **Navier-Stokes equation**, which corresponds to equation (2).

2.3 Finite Element Method (FEM)

The stream function is obtained by solving the **Poisson equation**: Using the finite element method (FEM), the system is discretized as:

$$(M + \nu \Delta t S)\omega^{n+1} = M\omega^n + \Delta t T(\omega, \psi) \tag{4}$$

$$S\psi = M\omega \tag{5}$$

where: M is the mass matrix. S is the stiffness matrix. $T(\omega, \psi)$ represents the transport term.

2.4 Impact of Boundary Conditions on the FEM Structure of the Linear system

2.4.1 Summarize

The FEM formulation relies on:

1. Basis functions ϕ_i.
2. Integrals defining the matrices:

$$M_{ij} = \int_\Omega \phi_i \phi_j d\Omega, \quad S_{ij} = \int_\Omega \nabla \phi_i \cdot \nabla \phi_j d\Omega. \tag{6}$$

3. The discrete linear system:

$$(M + \nu \Delta t S)\omega^{n+1} = M\omega^n + \Delta t T(\omega, \psi), \tag{7}$$

which remains structurally unchanged regardless of boundary conditions.

Boundary conditions only modify how the stream function ψ and vorticity ω are enforced at the boundaries:

2.4.2 Reasons

In the implementation of the Navier–Stokes solver during the class, the computational domain was assumed to have no boundary. But now we consider two types of boundary conditions for surface meshes with a boundary:

- **Non-penetrating condition**: The normal velocity component at the boundary is set to zero, ensuring that fluid particles cannot cross the boundary.
$$\mathbf{u} \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega$$
- **No-slip + Non-penetrating condition**: Both the normal and tangential velocity components are set to zero,
$$\mathbf{u} \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega$$
and
$$\mathbf{u} \cdot \boldsymbol{\tau} = 0 \quad \text{on } \partial\Omega$$
so that
$$\mathbf{u} = 0 \quad \text{on } \partial\Omega$$

These conditions do not change the **fundamental FEM discretization** of the Navier–Stokes equations, but they **introduce additional constraints** in the system. Specifically:

- The **Non-penetrating condition** affects the **stream function formulation**, requiring the stream function's **normal derivative to vanish** at the boundary.
$$\nabla\Psi \cdot \boldsymbol{\tau} = 0 \quad \text{on } \partial\Omega$$
which implies that Ψ is constant on (each connected component of) $\partial\Omega$
- The **No-slip + Non-penetrating condition** also result in the similar conclusion that
$$\Psi = 0 \quad \text{on } \partial\Omega$$
- In the time-stepping procedure, these constraints must be accounted for to ensure stability and physical consistency of the numerical solution.

Condition	**Stream Function** ψ on $\partial\Omega$	**Vorticity** ω on $\partial\Omega$
Non-penetrating	$\psi = cst$	Unconstrained
No-slip + Non-penetrating	$\psi = 0$	$\omega = 0$

Table 1: Comparison of boundary conditions and their effects.

3 Code Adjustments for Boundary Conditions

Since the FEM system structure remains unchanged, modifications are required in **stream function computation** and **time-step handling**.

3.1 Adding a Boundary Array to Identify Boundary Points

- In `mesh.h`, we introduced a new array `boundary` to mark whether a vertex is a boundary point. The array is defined as follows:

```
                TArray<uint8\_t> boundary;
```

  where `boundary[i] = 0` indicates an interior point, and `boundary[i] = 1` indicates a boundary point.

- The `boundary` array is then initialized in `cube.cpp` and `sphere.cpp` to properly identify boundary points.

3.2 Adding the `hemisphere` and `grid2D` Domains

In this project, we extended the available computational domains by introducing `hemisphere` and `grid2D`. This required the creation of the following new files: `hemisphere.h`, `hemisphere.cpp`, `2Dsquare_grid.h`, and `2Dsquare_grid.cpp`.

1. **Hemisphere**: This domain is derived from the original `sphere.cpp`. Initially, a cube mesh was projected onto a sphere. However, due to distortion issues during the projection, we opted for a different approach: first, we performed a normal projection of the mesh onto the sphere and then removed the portion where $z < 0$, thereby obtaining the upper hemisphere.

2. **Grid2D**: To support this new domain, we added appropriate file paths to `test_navier_stokes.cpp` and `CMakeLists.txt`. However, since the solver was originally designed for three-dimensional domains, the adaptation to two dimensions introduces certain challenges, such as differences in the numerical formulation and mesh representation. As a result, the code currently does not execute correctly for the 2D case.

3.3 Modifications in `navier_stokes.cpp`

1. **Adjustments in Stream Function Computation**

   To enforce the Dirichlet boundary condition (No-slip + Non-penetrating) and apply the vorticity boundary condition at the boundary, the following modification is introduced:

```
    for (size_t i = 0; i < N; ++i) {
        if (boundary[i]) {
            psi[i] = 0;    // Enforce Dirichlet boundary
                condition
            omega[i] = 0;  // Apply vorticity boundary
                condition
        }
    }
```

Listing 1: Boundary Condition for Stream Function

2. **Modifications in Conjugate Gradient Solver for Stream Function**

   The function `size_t NavierStokesSolver::compute_stream_function()` has been modified to ensure that the conjugate gradient method correctly accounts for the boundary constraints:

```
    for (size_t i = 0; i < N; ++i) {
        if (boundary[i]) {
            Psi[i] = 0;
            R[i] = 0;
            P[i] = 0;
        }
    }
```

Listing 2: Conjugate Gradient Solver for Stream Function

3. **Modifications in the Vorticity Computation (Time Stepping)**

   This modification applies only under the No-slip + Non-penetrating boundary condition. The function `void NavierStokesSolver::time_step(double dt, double nu)` has been adjusted as follows:

```
    void NavierStokesSolver::time_step(double dt, double nu) {
        iter1 = compute_stream_function();  // Compute psi
        compute_transport(p.data); // Compute non-linear
            convection
        blas_axpby(1, Momega.data, dt, p.data, N); // Compute
            right-hand side
        iter2 = conjugate_gradient_solve(S, p, omega, r, p, Ap
            , &rel_error, tol, iter_max, false);

        set_zero_mean(omega.data);  // Maintain zero mean for
            stability
    }
```

Listing 3: Time-stepping Adjustment for Vorticity Computation

3.4 Modifications in `test_navier_stokes.cpp`

To support additional computational domains, we modified the main function to include two new input regions: `hemisphere` and `2Dsquare`. The corresponding changes in the mesh loading function are as follows:

```
static int load_mesh(Mesh &mesh, int argc, char **argv)
{
    int res = -1;
    bool is_2D = false;

    if (argc > 2 && strncmp(argv[1], "cube", 4) == 0) {
        res = load_cube(mesh, atoi(argv[2]));
    } else if (argc > 2 && strncmp(argv[1], "sphere", 5) == 0) {
        res = load_sphere(mesh, atoi(argv[2]));
    } else if (argc > 2 && strncmp(argv[1], "hemisphere", 10) == 0)
     {
        res = load_hemisphere(mesh, atoi(argv[2]));
    } else if (argc > 2 && strncmp(argv[1], "2Dsquare", 8) == 0) {
        res = load_square_grid(mesh, atoi(argv[2]));
        is_2D = true;
    } else if (argc > 1) {
        res = load_obj(argv[1], mesh);
    }

    if (res == 0 && !is_2D) {
        rescale_and_recenter_mesh(mesh);
    }

    return res;
}
```

Listing 4: Modifications in `load_mesh()`

4 Problems Encountered When Dealing with Different Boundary Conditions

Since the FEM matrices M and S are determined by the mesh structure rather than boundary conditions, the fundamental structure of the discrete system remains unchanged. However, modifications are required in the enforcement of boundary conditions, particularly in the stream function computation and time-stepping process.

The theoretical aspects of the boundary conditions have been discussed earlier. During the implementation process, we encountered several challenges:

Initially, we aimed to implement an interactive mechanism that would allow the solver to dynamically adapt to different boundary conditions based on user selection. However, due to constraints in our coding capabilities, this approach was ultimately unsuccessful. Consequently, we decided to explicitly separate different boundary condition cases into distinct implementations.

4.1 Initial Approach

Our initial strategy involved the following modifications:

1. Adjusting `navier_stokes.cpp` to correctly implement the selection logic for boundary conditions.

2. Modifying `test_navier_stokes.cpp` to ensure proper interactive functionality.

3. Updating `CMakeLists.txt` to guarantee correct file references.
4. Modifying `navier_stokes.h` to ensure the necessary functions are correctly defined and accessible.

4.2 Final Solution

Instead of implementing a dynamic selection mechanism, we opted for a simpler and more maintainable approach: creating two separate directories, each dedicated to handling a specific boundary condition case. This ensures clarity in implementation and avoids potential errors arising from dynamic boundary condition switching.