Numerical Simulation of Ion Electrodiffusion and Advection in Atmospheric Ionic Thrusters - IPROP Project

Advance Programming for Scientific Computing

Students: T. ANDENA - G. BOTTACINI
Supervisor: C. DE FALCO - P. BARBANTE - L. VALDETTARO

Report for the project of the course
Advance Programming for Scientific Computing

Master degree in Mathematical engineering
Politecnico di Milano
Italy
November 6, 2024

Contents

Introduction

This work is part of the IPROP project (Ion PROPulsion in Atmosphere), which aims to explore the potential of non-thermal plasma atmospheric propulsion in ion-powered airships. This European initiative brings together collaboration among research groups from four EU countries and is funded by the European Union's Horizon Europe Research and Innovation Programme. The primary objective of the project is to advance ionic air-breathing propulsion systems beyond the pioneering phase, further investigating their capabilities and improving their performance.

In recent years, the increasing interest in "green" technologies has stimulated research into Electrohydrodynamic (EHD) propulsion and atmospheric ionic thruster. EHD propulsion is an innovative technology that utilizes the air molecules, ionized and then accelerated by two electrodes, to generate thrust in the atmosphere.

In this work, a 2D numerical model is implemented to simulate ionic drift under the influence of the electric field created between a collector and an emitter of possibly different shapes. We focused on solving a nonlinear Poisson problem coupled with the Drift-Diffusion equation, to cover the effects of the electric field, and the Navier-Stokes system, to take into account the dynamics of the fluid. The discrete problem is discretized using the Finite Element Method.

The simulations were conducted exploiting the open-source deal.II library, a `C++` software framework specifically designed for finite element computations. The `C++` code was integrated with meshes generated by `gmsh`, a finite-element mesh generator, and a flexible user interface implemented exploiting a `JSON` parsing technique.

The report addresses the following topics in the next chapters: the mathematical model describing the physical phenomena, the numerical algorithm exploited to solve the discrete problem, the instructions to download and execute our code, an extensive documentation of the scripts and lastly a comment on the numerical results and the overall of the project. In particular, we stress that in section (5.1) we outlined in detail the starting point, objectives and obtained goals of this project.

Mathematical Model

EHD systems are extremely complex to model given the richness of different physical natures. In particular, we considered only electrical and fluid dynamic phenomena, completely neglecting the thermal features. These two physical subsystems interacts through the Columb force (electric system towards fluid dynamic one) and the convective phenomena (the other way around). Basically, electric current due to drifting ions generates bulk fluid flow which, in turn, contributes to ion drift. This physical phenomenon is translated mathematically in a coupled problem to be solved. The coupled problem consist in the following equations: on the one hand a Poisson equation and Drift-Diffusion equation to face the electrical part, on the other one, the Navier-Stokes equation to address the fluid dynamics.

2.1 Electrical Problem Equations

To derive the system of the electrical part we start from Maxwell's equations for the electric field $\boldsymbol{E}$ and the magnetic field $\boldsymbol{H}$:

$$\begin{cases} \epsilon_0 \nabla \cdot \boldsymbol{E} = \rho & (2.1) \\ \nabla \times \boldsymbol{E} = -\mu_0 \dfrac{\partial \boldsymbol{H}}{\partial t} & (2.2) \\ \nabla \cdot \boldsymbol{H} = 0 & (2.3) \\ \nabla \times \boldsymbol{H} = \epsilon_0 \dfrac{\partial \boldsymbol{E}}{\partial t} + \boldsymbol{J} & (2.4) \end{cases}$$

where:

- μ_0 is the magnetic permeability,
- ϵ_0 is the electric permittivity of free space,
- ρ is the charge density,
- $\boldsymbol{J}$ is the current density.

The charge density ρ and the current density $\boldsymbol{J}$ are related by the continuity equation, which is derived by taking the divergence of 2.4 and using 2.1:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \boldsymbol{J} = 0 \tag{2.5}$$

Moreover, in the context of plasma, the magnetic field can be considered constant over time (quasi-static approximation) [1]. Thus, from 2.2 we obtain $\nabla \times \boldsymbol{E} \approx 0$, meaning that the electric field can be expressed as the gradient of a scalar potential: $\boldsymbol{E} = -\nabla \Phi$. Substituting this into 2.1 we derive the previously mentioned Poisson equation:

$$-\nabla \cdot \epsilon_r \nabla \Phi = \frac{\rho}{\epsilon_0} \tag{2.6}$$

where:

- Φ is the electric potential,
- ϵ_r is the relative air permittivity.

The charge density ρ can be computed as $\sum_s q_s n_s$ where for each particle species s, the density n_s is multiplied by the respective charge q_s. By rearranging equation 2.6 we obtain the first equation of our system:

$$-\nabla \cdot \epsilon \nabla \Phi = \sum_s q_s n_s \tag{2.7}$$

where $\epsilon = \epsilon_r \cdot \epsilon_0$ is the absolute air permittivity.

The number density of each species s can be evaluated thanks to the following continuity equation based on 2.5:

$$\frac{\partial n_s}{\partial t} + \nabla \cdot \boldsymbol{\Gamma_s} = G_s - R_s \tag{2.8}$$

where:

- $\boldsymbol{\Gamma_s} = n_s \boldsymbol{u_s}$ is the particle s flux,
- G_s is the rate of particles s generation,
- R_s is the rate of particles s loss.

Now we want to derive an equation for the flux $\boldsymbol{\Gamma}$. We start with the steady state force equation [1]:

$$m_s n_s \left[\frac{\partial n_s}{\partial t} + \mathbf{u}_s \cdot \nabla \mathbf{u}_s \right] = q_s n_s \mathbf{E} - \nabla p - m_s n_s \nu_m \mathbf{u}_s \tag{2.9}$$

where:

- m_s is the mass of the species
- ν_m is the frequency of momentum transfer to back-ground particles
- $\mathbf{u}_s \cdot \nabla \mathbf{u}_s$ is the inertial term

Given the assumptions of negligible time variations and that the inertial term is dominated by collisions at high pressures, we can set the left-hand side of the equation to zero. Moreover, using the isothermal relation

$$\nabla p = T_s k_B \nabla n_s, \tag{2.10}$$

the equation simplifies to:

$$q_s n_s \mathbf{E} - T_s k_B \nabla n_s - m_s n_s \nu_m \mathbf{u}_s = 0 \tag{2.11}$$

where:

- k_B is the Boltzman constant
- T_s is the temperature of the species

Dividing by $m_s\nu_m$ we recover an equation for the flux:

$$\mathbf{\Gamma}_s = n_s \boldsymbol{u}_s = \frac{q_s}{m_s\nu_m} n_s \boldsymbol{E} - \frac{T_s k_B}{m_s \nu_m} \nabla n_s \tag{2.12}$$

We introduce now the diffusivity $D = \frac{k_B T}{m\nu}$ and the mobility $\mu = \frac{q_e}{m\nu}$ coefficients which are related by the Einstein relation:

$$D = \mu \frac{k_B T}{q_e} \tag{2.13}$$

This leads us to the second and last electrical relation, namely the drift-diffusion equation:

$$\frac{\partial n_s}{\partial t} + \nabla \cdot [(sign(q_s)\mu \boldsymbol{E} + \boldsymbol{v}) n_s - D \nabla n_s] = G_s - R_s \tag{2.14}$$

Notice that in the last relation we already encoded the effect of the fluid dynamics on the electrical problem. As can be seen the velocity of the fluid $\boldsymbol{v}$ appears inside the transport term, influencing the evolution of the ions density. This conclude the electrical model, we pass now to the description of the second problem.

2.2 Fluid Dynamics Equations

To determine the fluid velocity, we employed the Navier-Stokes equations. Based on experimental results and existing literature, the velocities achieved by this type of propulsion are significantly lower than the speed of sound. Therefore, we use the incompressible form of the Navier-Stokes equations.

$$\begin{cases} \dfrac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} + \dfrac{1}{\rho}\nabla p = \nu \Delta \mathbf{v} + \dfrac{\mathbf{f_{EHD}}}{\rho} & (2.15) \\ \nabla \cdot \mathbf{v} = 0 & (2.16) \end{cases}$$

where:

- $\mathbf{v}$ is the fluid velocity
- p is the pressure
- ν is the kinematic viscosity of the medium
- $\mathbf{f_{EHD}}$ represents the forces due to electromagnetic interactions per unit volume

As we already stressed in the introduction of this chapter, any temperature variation is neglected, hence no energy equation is required. Since the fluid under consideration is mono-phase and the air permittivity is nearly equivalent

to the void's one, the electro-dynamic forces can be reduced to Coulomb's force alone. Therefore,

$$\mathbf{f_{EHD}} = \sum_s q_s n_s \mathbf{E} \tag{2.17}$$

No buoyancy forces are considered, since in first approximation this force is dominated by inertial ones. This conclude the fluid problem discussion.

2.3 Domain and Boundary Conditions

We considered different geometry on which we performed the numerical simulations, here we introduce the shared nomenclature. For all the experiments we considered a 2D domain $\Omega \subset \mathbb{R}^2$, whose boundary $\partial\Omega$ were characterized by:

- Γ_C surface collector boundary
- Γ_E surface emitter boundary
- Γ_{in} the inlet region of the domain
- Γ_{out} the outlet region of the domain

2.3.1 Poisson Boundary Conditions

The electrodes act as equipotential surfaces, so Dirichlet boundary conditions for the potential Φ are applied both on the emitter and collector surface. Assuming the boundaries are sufficiently distant from the emitter, homogeneous Neumann conditions can be imposed for the potential on the rest of the boundary, leading to the following:

$$\begin{cases} \boldsymbol{n} \cdot \nabla\Phi = 0 \quad \text{on} \;\; \partial\Omega/(\Gamma_C \cup \Gamma_E) \\ \Phi = V_C \quad \text{on} \;\; \Gamma_C \\ \Phi = V_E \quad \text{on} \;\; \Gamma_E \end{cases} \tag{2.18}$$

Where $\boldsymbol{n}$ represents the outward normal with respect to the considered edge.

2.3.2 Charge Continuity Boundary and Initial Conditions

A positive corona discharge is considered, since the most used in the literature experiments. We decided to implement the "exponential diode" conditions for both emitter and collector, while a simple Dirichlet condition is imposed on the inlet of the domain (if present in the geometry). We refer [2] to an accurate description and a physical explanation. The "exponential diode" is a Robin condition that collapses into a Dirichlet one since we set to zero the coefficient of the normal derivative:

$$\begin{cases} \alpha_4 n + \beta_4 \partial_{\vec{n}} n = \kappa_4 \;\; \text{on} \;\; (\Gamma_C \cup \Gamma_E) \times (0,t) \\ \alpha_4 = q\mu E_{on}, \quad \beta_4 = 0, \quad \kappa_4 = q\mu E_{on} N_{ref} \exp\left(\frac{E_n - E_{on}}{E_{ref}}\right), \end{cases} \tag{2.19}$$

Where the previous symbols represent respectively:

- E_{on}: a threshold value for the electric field in order to trigger the corona discharge phenomena
- E_{ref}: reference value for the exponent
- N_{ref}: reference value for ion density
- E_n: electric field in the normal direction

Actually, for stability reasons, we implemented a slightly different condition; in particular, a minimum density were considered in order to avoid a null-density condition on the boundaries of the electrodes. Eventually, the maximum between the previous condition an this reference value was taken as value to be imposed at the boundaries. At the inlet we impose a constant value, corresponding to the charge density of ions in the ambient, on the remaining boundary we impose homogeneous Neumann condition:

$$\begin{cases} n = N_0 \ \text{ on } \ \Gamma_{in} \times (0,t) \\ \nabla n \cdot \boldsymbol{n} = 0 \ \text{ on } \ (\partial\Omega \setminus (\Gamma_C \cup \Gamma_E \cup \Gamma_{in})) \times (0,t) \end{cases} \tag{2.20}$$

As the time-varying Drift-Diffusion equations are considered, an initial condition is also required. We decide to impose N_0 in all the domain.

2.3.3 Navier-Stokes Boundary and Initial Conditions

The no-slip condition is applied on both the electrodes since solid surfaces. A uniform velocity profile is impose on the inlet boundary, instead the usual homogeneous Neumann condition is impose at the outlet. If they are present in the geometry, a normal zero velocity is imposed on the top and bottom sides of the domain. These conditions are mathematically translated as:

$$\begin{cases} \boldsymbol{v} = V\,\boldsymbol{x} \ \text{ on } \ \Gamma_{in} \times (0,t) \\ \boldsymbol{v} = 0 \ \text{ on } \ (\Gamma_C \cup \Gamma_E) \times (0,t) \\ \boldsymbol{n} \cdot \nabla \boldsymbol{v} = \boldsymbol{0}\ ,\ p = 0 \ \text{ on } \ \Gamma_{out} \times (0,t) \\ \boldsymbol{v} \cdot \boldsymbol{n} = 0 \ \text{ on } \ (\partial\Omega \setminus (\Gamma_{in} \cup \Gamma_{out} \cup \Gamma_C \cup \Gamma_E)) \times (0,t) \end{cases} \tag{2.21}$$

At initial time we have a zero velocity fluid.

Numerical Model

The system of partial differential equations discussed in the previous chapter is both coupled and highly nonlinear. This limits the possibility of finding analytical solutions to only a few simplified ideal cases. Therefore, a numerical approach is necessary to obtain an approximate solution that is practically useful in engineering applications. This section outlines the discretization, linearization, and stabilization techniques used to achieve such solution, as discussed in detail in the Master's thesis by Matteo Menessini [3]. Both software and codes are discussed in the following chapters.

3.1 Electrical Problem

The electrical problem involves finding solutions for the potential Φ and ion density n based on the previously discussed equations (2.7) and (2.14) with their own boundary conditions (2.18), (2.19 - 2.20) and (for the drift-diffusion equation) initial condition.

The two most common iterative methods for this problem are the fully coupled Newton method and the decoupled Gummel map. The fully coupled Newton method is typically more effective for problems involving higher currents, but it requires a good initial guess, and the Jacobian matrix used in Newton's iteration is often ill-conditioned. In contrast, the decoupled Gummel map is more commonly used for low-current scenarios and can converge to the correct solution without requiring a precise initial guess. Since this code is designed to be predictive, the Gummel map approach is chosen. The key aspect behind the Gummel map algorithm (see [4]) is the use of the so-called Slotboom variables in the Poisson equation. This change of variables is used to stabilize the electrical problem, the only drawback is that they introduce a nonlinearity in the Poisson problem. In the following sub-sections we will discuss both the discretized problem that build the electrical part of the phenomena, namely the nonlinear Poisson problem and the drift diffusion system.

3.1.1 Nonlinear Poisson Equation

Considering a control volume with a uniform electron density η, and an uniform positive ion density π, the application of an electric field will move the charges. Electrons will be attracted to regions where the potential Φ is higher, shifting proportionally to $exp(\frac{\Phi}{\beta T_e})$, where we recall that $\beta = \frac{k_B}{q_e}$. Conversely, positive ions will be attracted to lower potential regions, hence their density shift will be proportional to $exp(-\frac{\Phi}{\beta T_p})$. Thus, the following variable change by Slotboom is proposed:

$$n_e = \eta exp(\frac{\Phi}{\beta T_e}), \qquad n_p = \pi exp(-\frac{\Phi}{\beta T_p})$$

This change is enough to stabilize the electrical problem. Since this change introduce a non linearity in the potential, we need to employ a non-linear solver, thus we introduce the Newton method. First, we recover the non-linear equation starting from 2.6 by employing the Slotboom variables and considering only a positive ion density:

$$\nabla \cdot \epsilon_r \nabla \Phi + \frac{q_e}{\epsilon_0} \left(\pi \exp\left(-\frac{\Phi}{\beta T_p} \right) \right) = 0$$

Then, starting from a suitable initial guess Φ^0, we define the newton operator:

$$F(\Phi) = \nabla \cdot \epsilon_r \nabla \Phi + \frac{q_e}{\epsilon_0} \left(\pi \exp\left(-\frac{\Phi}{\beta T_p} \right) \right)$$

that is employed in the following iterative Newton-Raphson algorithm:

Algorithm 1 Newton-Raphson algorithm for the non-linear Poisson equation

while $\delta\phi^k > tol$ **do**
  solve $\nabla F(\delta\phi^k) = -F(\phi^k)$
  update $\phi^{k+1} = \phi^k + \delta\phi^k$
  update $\pi = \pi \exp\left(-\frac{\delta\phi^k}{\beta T_p}\right)$
end while

where $\nabla F(\delta\Phi^k)$ represents the directional gradient of $F(\Phi^k)$ at Φ^k along $\delta\Phi^k$:

$$\nabla F(\delta\Phi^k) = \nabla \cdot \epsilon_r \nabla \delta\Phi^k - \frac{q_e}{\epsilon_0} \pi \exp\left(-\frac{\Phi^k}{\beta T_p} \right) \delta\Phi^k$$

Note that the equation to solve at each step is linear in $\delta\Phi^k$. Despite being physically stable, the Newton method can still present oscillations due to numerical reasons. A possible solution to prevent overshooting is to use some kind of *damping* on the update term. The simplest form of damping is implemented: the infinity norm of the update term $\delta\Phi^k$ is limited to the value βT_p. We refer to [3] for the derivation of the weak form of $\nabla F(\delta\Phi^k) = -F(\Phi^k)$, obtained by left-multiplying by the test function ψ_i and integrating over the domain Ω.

We directly pass to the description of the algebraic system obtained exploiting the Galerkin method. An approximation of the potential $\Phi \approx \Phi_h = \sum_j U_j \psi_j$ is to be found, with ψ_j the j-th finite element shape functions and U_j the unknown coefficients. The system to solve in order to compute the Newton update $\delta\Phi^k$ is $A^k \boldsymbol{U}^k = -\boldsymbol{f}^k$ where:

$$\begin{cases} A_{ij}^k = (\nabla\psi_i, \epsilon_r \nabla\psi_j) + \frac{q_e}{\epsilon_0} \frac{\pi}{T_p} \exp\left(-\frac{\Phi^k}{T_p} \right) (\psi_i, \psi_j) \\ \boldsymbol{f}_i^k = (\nabla\psi_i, \epsilon_r \nabla\Phi^k) - \psi_i \frac{q_e}{\epsilon_0} \left[\pi \exp\left(-\frac{\Phi^k}{T_p} \right) \right] \end{cases}$$

The linear system $A^k \boldsymbol{U}^k = -\boldsymbol{f}^k$ problem is solved via the direct solver UMFPACK, optimized for sparse matrices. The software uses the SuiteSparse library,

which employs a set of routines for solving non-symmetric sparse linear systems using the Unsymmetric-pattern MultiFrontal method, and direct sparse LU factorization. Linear Q_1 finite elements are used for the potential. This conclude the description of the first electrical algorithm.

3.1.2 Drift-Diffusion Equations

The numerical solution of the drift-diffusion equations requires a space and a time discretization. For what concern the former, the most commonly used method, is the Scharfetter Gummel (SG) scheme, that allows to integrate transport and diffusive terms at the same time. Using a 2D grid with 4 node cells and integrating equation (2.14) over a control region we get:

$$\frac{\partial}{\partial t}\int_{V_i} n\, dV + \oint_{A_i} \mathbf{\Gamma}\cdot\boldsymbol{n}\, dS = \int_{V_i} (G_i - R_i)\, dV \tag{3.1}$$

where A_i is the contour of the control volume V_i and Gauss's theorem was used for the second term. The control volume of each node i is defined as the area of the polygon connecting the centers of all grid cells which share node i. Approximating the volume integrals and the contour integral via the mean value theorem, and using nodal values:

$$\frac{\partial n_i}{\partial t} + \frac{1}{V_i}\sum_j l_{ij}\,(\mathbf{\Gamma}\boldsymbol{n})_{ij} = (G_i - R_i) \tag{3.2}$$

where the sum is carried over all nodes j connected to node i and l_{ij} is the measure of the segment connecting the centers of the two cells which share side ij. By approximating $(\mathbf{\Gamma}\boldsymbol{n})_{ij} \approx \mathbf{\Gamma}_n$, with $\mathbf{\Gamma}_n$ the flux going from i to j measured in the intersection between the contour of the control volume V_i and the side ij. Then, the term $\mathbf{\Gamma}_n$ can then be approximated as:

$$\mathbf{\Gamma}_n = (-D\nabla n + \mu\nabla\Phi n + \boldsymbol{v}n)\cdot\mathbf{u}_{ij} \tag{3.3}$$

where $\mathbf{u}_{ij}$ represents the versor going from i to j. Note that we have a positive sign for the second term in the right-hand side since the assumption of positive ions, it would become a minus by considering negative ions. Both generation and recombination are considered negligible between nodes (note that the assumption on generation might not be negligible near the emitter). By exploiting the Einstein relation, assuming the potential varies linearly between nodes and by solving a differential equation, relation (3.3) becomes:

$$\frac{\mathbf{\Gamma}_n}{D} = \frac{\alpha_{ij}}{\exp(\alpha_{ij}L) - 1}\,(n_i \exp(\alpha_{ij}L) - n_j) \tag{3.4}$$

Where L is the measure of the cell segment adjoining node i and j, n_i and n_j the densities respectively at node i and node j, and finally $\alpha_{ij} = \frac{\Phi_j - \Phi_i}{\beta L} + \frac{v_u}{D}$ with $\beta = \beta T$ and $v_u = \boldsymbol{v} \cdot \mathbf{u}_{ij}$. By introducing the Bernoulli equation:

$$B(x) = \frac{x}{\exp(x) - 1}$$

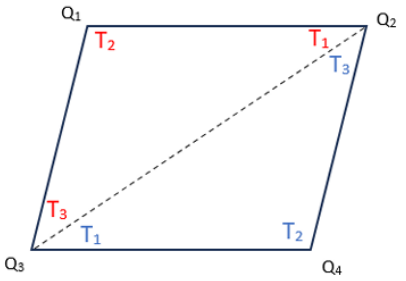
the previous equation can be rewritten as:

$$\frac{\mathbf{\Gamma}_n}{D} = \frac{1}{L}\left[-B(\alpha_{ij}L)n_j + B(-\alpha_{ij}L)n_i\right] \tag{3.5}$$

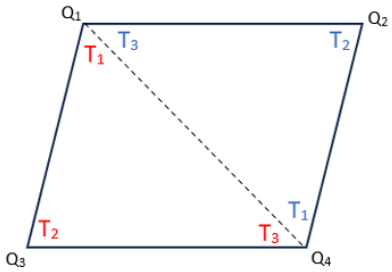
Finally, by replacing (3.5) into (3.2) we get the final form of the spatial discretizzation:

$$\frac{\partial n_i}{\partial t} - \frac{1}{V_i}\sum_j L_{ij}\frac{D}{L}\left[B(\alpha_{ij}L)n_j - B(-\alpha_{ij}L)n_i\right] = (G_i - R_i) \tag{3.6}$$

with L_{ij} distance between node i and node j. It is important to stress that this method ensures stability for triangular meshes, instead when 4-node cells are used, it converges to the correct solution only in cases where the mesh is structured and cartesian. For this reason, (3.6) is used to obtain the matrix for triangular cells, whose elements are then used to assemble the system matrix for 4 node cells compatible with `deal.ii`. In order to do this, the 4 node cells need to be split into two triangular cells. This can be done in two ways, using either diagonal. In order to obtain a cell matrix with the largest diagonal terms, the diagonal to be chosen is the one which splits the largest of the 4 angles of the original cells. Save pathological, the shortest diagonal is the one that satisfies this condition in most cases. The next picture describe exactly this procedure:



(a) 4-node cell split by the longest diagonal



(b) 4-node cell split by the shortest diagonal

Figure 3.1: Splitting of 4-node cells into 2 triangular elements sharing the same nodes

The cell matrix for triangular cells with vertices numbered from 1 to 3 is assembled as follows:

$$D\,A \begin{bmatrix} -(B_{12}^{+} l_{12} + B_{31}^{-} l_{31}) & B_{12}^{+} l_{12} & B_{31}^{-} l_{31} \\ B_{12}^{-} l_{12} & -(B_{12}^{-} l_{12} + B_{23}^{+} l_{23}) & B_{23}^{+} l_{23} \\ B_{31}^{+} l_{31} & B_{23}^{-} l_{23} & -(B_{31}^{+} l_{31} + B_{23}^{-} l_{23}) \end{bmatrix}$$

where D is the diffusivity, A the measure of the cell, B_{ij}^{+} and B_{ij}^{-} indicate the Bernoulli function calculated for $\beta_{ij} = \frac{\Phi_j - \Phi_i}{\beta T} + \frac{v_u}{D} L_{ij}$ and $-\beta_{ij}$ respectively. Finally, $l_{ij} = \frac{\boldsymbol{n}_i}{2A} \frac{\boldsymbol{n}_j}{2A}$, with $\boldsymbol{n}_i$ the vector pointing towards node i from the opposite face and having the same norm as the length of that face. Only six coefficients need to be computed per triangular cell, by adding all the contributions coming from each element of the mesh, we defined the matrix DD as the square matrix with rows equal to the mesh vertices.

For what concern time integration, Backward Euler (BE) method is employed for the Drift Diffusion equation. We introduce the mass matrix M, with coefficients:

$$M_{ij} = \int_{\Omega} \psi_i(s) \psi_j(s) ds \tag{3.7}$$

where ψ_i are the basis of the shape functions of the finite elements used. The integral is determined by numerical quadrature. Now, the BE method reads, with a given step-size Δt:

$$(M + \Delta t DD)\boldsymbol{N}^{n+1} = M\boldsymbol{N}^n + \Delta t \boldsymbol{F}^{n+1} \tag{3.8}$$

where $\boldsymbol{N}^n$ is the solution vector at time step n containing the values of ion density at each node, and $\boldsymbol{F}^n$ is the vector containing the RHS evaluated at each node at time instant n. This algebraic system, can be solved using an iterative method, alternatively, since both DD and M are sparse matrices (as the non-zero entries in each row are those relative to the node itself and its neighbors), it can be solved using the sparse direct solver UMFPACK.

3.1.3 Gummel Map algorithm

We conclude this section by combing the two algebraic systems and introducing the algorithm that solves the electrical part of the problem: the Gummel Map algorithm. The drift diffusion equation is solved after the non-linear Poisson equation as a fixed point problem, until the relative error between the previous and current fixed point iterations is below a certain tolerance, this is repeated for every time-step. In pseudo-code, this can be implemented as:

Algorithm 2 Gummel map algorithm for the time integration of the equations

while $err_T > tol_T$ **do**
  while $err_{FP} > tol_{FP}$ **do**
    set $\pi = \boldsymbol{N}^{old}$ ▷ ion density in NLP problem
    while $\delta\Phi^k > tol_\Phi$ **do**
      solve $\nabla F(\delta\Phi^k) = -F(\Phi^k)$
      update $\Phi^{k+1} = \Phi^k + \delta\Phi^k$
      update $\pi = \pi \exp(-\frac{\delta\Phi^k}{\beta T_p})$
    end while ▷ we find the potential Φ
    build DD from the previous potential Φ
    solve $(M + \Delta t DD)\boldsymbol{N}^i = M\boldsymbol{N}^{old}\Delta t RHS(t)$ ▷ we find ion density $\boldsymbol{N}^i$
    $err_{FP} = \frac{|\boldsymbol{N}^i - \boldsymbol{N}^{i-1}|}{|\boldsymbol{N}^i|}$ ▷ compute the fixed point error
  end while
  $err_T = \frac{|\boldsymbol{N}^i - \boldsymbol{N}^{old}|}{|\boldsymbol{N}^i|}$ ▷ compute the time error
  $\boldsymbol{N}^{old} = \boldsymbol{N}^i$ ▷ update density
end while

3.2 Navier-Stokes System

The fluid problem, governed by the Navier-Stokes system, does not present specific challenges compared to the electric part. The approach we followed is very standard. Our Reynolds number, for all the geometries, lies in the laminar

regime. Nevertheless, Re is greater than 1, hence, inertial forces are not negligible compared to viscous forces. The numerical approach is formulated starting from the tutorial of deal.II [5], where the time-dependent Navier-Stokes equations are solved using an implicit-explicit (IMEX) scheme. The idea is to use Backward Euler for the time discretization, then diffusion term is treated implicitly and the convection term is treated semi-implicit (only difference with respect to the tutorial).

These choices lead to the following time-discretized Navier-Stokes equations:

$$\frac{\mathbf{u}^{n+1}-\mathbf{u}^n}{\Delta t} - \nu\Delta\mathbf{u}^{n+1} + (\mathbf{u}^n\cdot\nabla)\mathbf{u}^{n+1} + (\mathbf{u}^{n+1}\cdot\nabla)\mathbf{u}^n + \nabla p^{n+1} = \mathbf{f}$$

$$\nabla\cdot\mathbf{u}^{n+1} = 0$$

By denoting $(\mathbf{v}, q)$ as the corresponding test functions (velocity and pressure) and integrating over the domain Ω we obtain the corresponding weak form:

$$\begin{aligned}\frac{1}{\Delta t}m(\mathbf{u}^{n+1},\mathbf{v}) + a(\mathbf{u}^{n+1},v) + c(\mathbf{u}^n;\mathbf{u}^{n+1},\mathbf{v}) + c(\mathbf{u}^{n+1};\mathbf{u}^n,\mathbf{v}) + b(p^{n+1},v) \\ = \frac{1}{\Delta t}m(\mathbf{u}^n,\mathbf{v}) - c(\mathbf{u}^n;\mathbf{u}^n,\mathbf{v}) + \langle f, v\rangle\end{aligned} \tag{3.9}$$

$$b(u^{n+1},q) = 0 \tag{3.10}$$

where:

- $m(u,v)$ is the mass term:

$$m(u,v) = \int_\Omega u\cdot v\, d\Omega$$

- $a(u,v)$ is the bilinear form for the diffusion term:

$$a(u,v) = \nu\int_\Omega \nabla u : \nabla v\, d\Omega$$

- $b(p,v)$ is the pressure term:

$$b(p,v) = -\int_\Omega p\,\nabla\cdot v\, d\Omega$$

$$b(u,q) = -\int_\Omega q\,\nabla\cdot u\, d\Omega$$

- $c(u^n,u^{n+1},v)$ is the trilinear form related to the convective term:

$$c(u^n,u^{n+1},v) = \int_\Omega (u^n\cdot\nabla)u^{n+1}\cdot v\, d\Omega$$

The system we want to solve can be synthesized in the matrix form:

$$\begin{pmatrix} A & B^T \\ B & 0\end{pmatrix}\begin{pmatrix}\mathbf{U}\\ P\end{pmatrix} = \begin{pmatrix}\mathbf{F}\\ 0\end{pmatrix}$$

Grad-Div stabilization

To enhance the efficiency and robustness of our method we add the Grad-Div stabilization:

$$\begin{pmatrix} A + \gamma B^T W^{-1} B & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}$$

It is observed that the choice of the parameter γ is quite important for the efficiency and the convergence of the code. A value of the same order of magnitude of the velocity is recommended. A detailed explanation of the Grad-Div stabilization can be found in [6].

Block Preconditioner

The block preconditioner that we use is taken from [7], with the addition of two terms: the inertial term (mass matrix) and the Grad-Div term. The block preconditioner can be written as:

$$P^{-1} = \begin{pmatrix} \tilde{A}^{-1} & -\tilde{A}^{-1} B^T S^{-1} \\ 0 & S^{-1} \end{pmatrix}$$

where S is the Schur complement of $\tilde{A} = A + \gamma B^T W^{-1} B$. Note that, it can be decomposed into the Schur complements of the mass matrix, diffusion matrix, and the Grad-Div term:

$$S^{-1} \approx S^{-1}_{\text{mass}} + S^{-1}_{\text{diff}} + S^{-1}_{\text{Grad-Div}} \approx [B^T (\operatorname{diag} M)^{-1} B]^{-1} + \Delta t(\nu + \gamma) M_p^{-1}$$

Conclusive remarks

We decided to employ second order FE base functions Q_2 for the velocity, while first order FE base functions Q_1 for pressure. This choice satisy inf-sup stability condition, hence no stabilization is needed. Moreover, we stress that the fluid dynamics are much slower than the electric dynamics, thus, the NS equations are not solved at every time step, but only every 40 electrical time steps.

3.3 Coupling algorithm

In this last section we want to stress the couplings between the two problems, and introduce the final form of the algorithm that we implemented in order to perform simulation. The sources of coupling are two:

1. The fluid problem influences the electrical problem by changing the flux of the ion density. This behavior can be seen in the fluid velocity $\boldsymbol{v}$ appearing in (3.3) and, in a less obvious way, in the definition of both α_{ij} and β_{ij}.
2. On the other way, the electrical problem build the force term on the RHS of the fluid problem (3.9).

To conclude this chapter we present the final algorithm that solves both physics:

Algorithm 3 Gummel map with Navier-Stokes

Given $\boldsymbol{N}^{old} = \boldsymbol{N}_0$
while $err_T > tol_T$ **do**
  while $err_{FP} > tol_{FP}$ **do**
    set $\pi = \boldsymbol{N}^{old}$ ▷ ion density in NLP problem
    while $\delta\Phi^k > tol_\Phi$ **do** ▷ start Newton algorithm
      solve $\nabla F(\delta\Phi^k) = -F(\Phi^k)$
      update $\Phi^{k+1} = \Phi^k + \delta\Phi^k$
      update $\pi = \pi \exp\left(-\frac{\delta\Phi^k}{\beta T_p}\right)$
    end while ▷ we find the potential Φ
    build DD from the previous potential Φ
    solve $(M + \Delta t DD)\boldsymbol{N}^i = M\boldsymbol{N}^{old}\Delta t RHS(t)$ ▷ we find ion density $\boldsymbol{N}^i$
    $err_{FP} = \frac{|\boldsymbol{N}^i - \boldsymbol{N}^{i-1}|}{|\boldsymbol{N}^i|}$ ▷ compute the fixed point error
  end while
  $err_T = \frac{|\boldsymbol{N}^i - \boldsymbol{N}^{old}|}{|\boldsymbol{N}^i|}$ ▷ compute the time error
  $\boldsymbol{N}^{old} = \boldsymbol{N}^i$ ▷ update density $\boldsymbol{N}$
  if time step % 40 == 1 **then**
    solve NS system ▷ update velocity $\boldsymbol{v}$
  end if
end while

We stress that the first condition in the `while` loop can be replaced with a final time to be reach.

Dependencies, installation and execution

In this chapter, we provide a detailed description of the technical aspects related to the compilation process and the instructions on how to execute our code. In particular, we discuss how to download it, the necessary dependencies, software requirements, and offer a step-by-step guide to run our simulations.

4.0.1 Supported platforms and software requirements

`deal.II` is an original work of the Numerical Methods Group at Universität Heidelberg in Germany. It is widely used in both academia and industry to develop complex simulation with applications in various fields, including fluid dynamics, structural mechanics, and electromagnetism. `deal.II` is a platform-independent library written in `C++`, with build support provided by `CMake` $\geq$ `3.13.4`. The user need `C++ 17` in order to run the code since version `9.5.1` of `deal.II` was used. This library can be compiled on most widely used systems. In particular it works with the `GCC` compiler on both `linux` and `windows` systems, `Clang` compiler on both `Mac` and `linux` systems. In our case we used a `GCC 11.2.0` compiler on both a `linux` machine (`pop-os 22.04`) and a `windows` machine equipped with the `WSL` version `2` (the windows subsystem linux exploits `ubuntu 22.04` distribution). The library relies on a large number of other packages, we report here all of them with the related version:

- `boost = 1.76.0`
- `trilinos = 15.0.0`
- `tbb = 2021.4`
- `blacs = 1.1`
- `mumps = 5.4.0`
- `suitesparse = 5.10.1`
- `adol-c = 2.7.2`
- `arpack = 3.8.0`
- `gsl = 2.7`
- `petsc = 3.21.0`
- `hypre = 2.22.0`
- `scalapack = 2.1.0`
- `fftw = 3.3.9`
- `p4est = 2.8.6`

- `openblas = 0.3.15`
- `scotch = 7.0.4`
- `metis = 5.1.0`
- `netcdf = 4.9.2`
- `hdfs = 1.12.0`

In addition to the core libraries, other packages need to by installed for supplementary reasons:

- `gmsh` $\geq$ `4.8.4`
- `ParaView` $\geq$ `5.9.1`

In particular, we reported `gmsh` a widely-used tool that is able to generate high-quality meshes, and `ParaView` for the visual inspection of the solutions.

4.0.2 Installation

If the packages are installed, the user can move on and download the code following this procedure:

1. First of all the user can either download `deal.II` from the official website or exploit the module structure from the MOX toolchain. If the second path is chosen we recall to type from terminal (or to source it in the .bashrc file) the following:

```
$ module load deal.ii
```

   We stress that the current release is the `9.6.0`, still we require the `9.5.1`. We suggest to follow the second way since making the installation from the website can be cumbersome due to the rich dependencies.

2. The user can download the code in his/her local machine in two ways, both exploiting `git` a command-line interface distributed version control software:

```
$ git clone git@github.com:MrAndena/IPROP_project.git
$ git clone https://github.com/MrAndena/IPROP_project.git
```

   Notice that the previous operation requires a SSH key authentication in the first case; we suggest to see the GitHub dedicated page for more information.

3. With a text editor, modify the `CMakeLists.txt` file of the desired simulation code, by changing `FIND_PACKAGE` related to `deal.II` library. This item specifies the location of the library/module inside your machine.

```
FIND_PACKAGE(deal.II 9.5.1
HINTS /path/to/my/dealii/9.5.1/location
)
```

4.0.3 Execution

Assuming the user cloned the repository, we will now explain how to execute our simulations. First of all the user has to decide which code to run, in particular, the following possibilities are available:

1. Validation of the electrical algorithm: directory `PNjunction/`
2. Validation of the fluid algorithm: directory `TimeDependentNS/`
3. Simulation of the coupled algorithm: directory `CoupledProblem/`

Once the choice has been taken, the user has to enter inside that directory:

```
$ cd IPROP_project
$ cd Directory_of_the_code_you_want_to _run
```

Now, the user should create the `build` directory, the one that will host the execution of the program. Once inside this new directory, the configuration of the building process is setup through the command `cmake ..`; subsequently the compilation of the code is performed through the command `make`:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

At this point, the executable of the code is being created. The most attentive user will surely have noticed that at the end of the compilation process the following screen in printed:

```
[100%] Build target name_of_the_executable
```

Hence, we also know the correct name of the executable to run. Finally, the code can be run in parallel exploiting `MPI`, moreover, the user can specify the number of desired cores:

```
$ mpiexec -n <number_of_cores> ./name_of_the_executable
```

We recall that the number of cores depend on the architecture of the machine and it can be checked by typing in the terminal:

```
$ lscpu
```

The results of the simulation, namely a list of .vtu and .pvtu files, are locate in the directory `../output7`, inside a folder that will be created during the simulation. You can inspect the evolution of the simulation opening these files in `ParaView`.

4.0.4 Custom commands

We report here a couple of custom made commands that we think can be handy. From the terminal, while inside the `build` directory of a generic case, the user can type:

```
$ make custom_clean
```

to delete the build directory (and obviously everything inside it), all the simulation results, and the file "compile_commands.json" that is created in the root after the "make" operation. This command can be useful if the user decide to edit the codes or the CMakeLists.txt file. Moreover, the user can be interested in deleting only the results in the `output` directory by typing:

```
$ make clean_output
```

Implementation

In this chapter, we present the implementation of the numerical model described in the previous sections and the philosophy behind the choices we made for the development of our code. As we wrote before, all the simulations rely on the open-source library `deal.II`, which offers a robust and flexible framework for solving partial differential equations (PDEs) exploiting finite elements. One of the most interesting features of this library is the richness of material that a generic user could find online. In particular we refer to the "Tutorials", "Classes" and "Namespaces" sections of the official website, that compose an extensive documentation. To provide a clear and user-friendly parsing interface we exploited `JSON`, a file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs. The `JSON` file serves as a configuration tool, allowing users to easily define physical and geometrical parameters without the need to modify the source code. This approach enhances flexibility, usability, and reduces potential for errors during the setup phase of simulations.

This chapter is structured as follows: section (5.1) contains an introduction that describe the starting point, the set objectives and the obtained goals of our project. In section (5.2) we give a broad overview of the architecture of the code. In Section (5.3) we explain the setting controls of the `JSON` file, instead, in section (5.4) (and the related subsections) we describe the source code.

5.1 Starting point, objectives, obtained goal

The starting point of our project was a sequential `deal.II` code written by the ex-student Matteo Menessini during his master thesis at Politecnico di Milano [3]. The original project contained the following material:

- Sequential code to perform validation of the electrical solver on a pn-junction
- Sequential code to perform validation of a steady Navier-Stokes solver on an airfoil naca geometry
- Sequential code to perform the complete algorithm (both electrical and fluid dynamic part) on an airfoil naca (or ellipsoidal) geometry

These codes represented a good starting point, however they were characterized by different types of problems; for this reason the objectives of our project could have taken multiple paths. We decided to focus on the following goals:

1. Develop a more articulate structure to execute the codes, since all the starting materials were stored in only 3 different .cc files. Hence, introduce a distinction between header (.hpp) and source (.cpp) files following one of the best practice of object oriented philosophy.

2. Parsing of physical and geometrical parameters in order to introduce a user-friendly flexible configuration, and to avoid re-compilation of the source code
3. Employ a time dependent Navier-Stokes system
4. Parallelization of the original codes
5. Generate a tool to create the computational mesh with the user specification, in order to perform the simulations on multiple geometries

Of the objectives indicated above we managed to achieve the first four, in particular the coupled problem is performed only on a geometry composed of two concentric circles. At the beginning of the project we obtained good results on a code capable of: first, generating an .msh file representing a mesh containing user's specifications, then performing a simulation on this grid. We then realized that it was not possible (at least in our code) to find a way to limit the user's choices regarding the parameters characterizing the geometry, so it was possible to generate extremely deformed and non-functioning meshes. Moreover, this type of approach excludes the possibility of viewing the mesh before the simulation. An eventuality that we wanted to remove since it is important to see the quality of the elements before running a simulation. That is the reason why we abandoned this idea and we created instead a folder called `meshes` where we store different .msh files on which our experiments were tested. As we said previously the complete problem works, for now, only on a particular type of mesh. Nonetheless, the code was written in such a way that it is easy to test the solver on new geometries. An hypothetical new developer has to to add the geometric specifications in the custom data struct and in the configuration file. Then, generate a coherent .msh file and modify a couple of class methods.

5.2 Directory structure

In this section we present the directory structure that we choose to articulate our project, in particular we discuss the content and the aim of each folder. Inside the `IPROP_project/` folder we can find four directories: three of them contain the code to perform simulations, in particular, two validation cases (`PNjunction/` , `TimeDependentNS/`) and one coupled problem (`CoupledProblem/`), the fourth one contains the meshes for all the problems (`meshes/`). Inside each of the first three cited directories, the reader can find the usual division of a `C++` project, in particular, we separated all the header files (.hpp extension) inside the `include/` folder and all the source files (.cpp extension) inside `src/`. As usual, the header files contain the declaration of all the classes, the declaration of helper functions and implementation of templates methods. Instead, inside `src/` we can find the main file, and the definitions of other functions. At the same level of `include/` and `src/` is located the empty folder `output/`, which will contains the results of the performed simulation. Moreover, at the same level, each of the case studies comes with a `CMakeListes.txt` file which handles the compilation process. Unlike validation cases, inside `CoupledProblem` the user can find

the folder `config/`, which contains the `JSON` configuration file. This is the file that bridges the hard-coded part of the project with the user specifications. We stress that this is the only file that a user should modify in order to perform a custom simulation (with the only exception of CMakeLists.txt file, see Ch 4). We refer to the next subsection for the details of the configuration.json file. We conclude this section with a brief description of the files that are located at the root level, inside `IPROP_project` folder. Among them we can find the classic `README` file, that describes the project and how to compile it, the report of this project and the hidden `.gitignore`, that specifies all the files that cannot be pushed on GitHub. We report here a diagram to help the reader and facilitate understanding.

```
IPROP_project/
├─ CoupledProblem/
│  ├─ config/
│  │  └─ configuration.json
│  ├─ include/
│  │  ├─ BlockSchurPreconditioner.hpp | BlockSchurPreconditioner_impl.hpp
│  │  ├─ CollectorGeometry.hpp
│  │  ├─ config_reader.hpp
│  │  ├─ data_struct.hpp
│  │  ├─ BoundaryValues.hpp
│  │  ├─ json.hpp
│  │  └─ CompleteProblem.hpp | CompleteProblem_impl.hpp
│  ├─ src/
│  │  ├─ main.cpp
│  │  └─ config_reader.cpp
│  ├─ output/
│  └─ CMakeLists.txt
├─ PNjunction/
│  ├─ include/
│  │  ├─ DriftDiffusion.hpp | DriftDiffusion_impl.hpp
│  │  ├─ Electrical_Constants.hpp
│  │  └─ Electrical_Values.hpp | Electrical_Values_impl.hpp
│  ├─ src/
│  │  └─ 2d.cc
│  ├─ output/
│  └─ CMakeLists.txt
├─ TimeDependentNS/
│  ├─ include/
│  │  ├─ BlockSchurPreconditioner.hpp | BlockSchurPreconditioner_impl.hpp
│  │  ├─ BoundaryValues.hpp
│  │  ├─ InsIMEX.hpp | InsIMEX_impl.hpp
│  │  └─ Time.hpp
│  ├─ src/
│  │  └─ time_dependent_NS.cc
│  ├─ output/
│  └─ CMakeLists.txt
├─ meshes/
├─ README
├─ PACS_report.pdf
└─ .gitignore
```

5.3 JSON setting

In this section we describe the entries of each attribute of the `configuration.json` file inside the `CoupledProblem` directory. One of the few disadvantages of `JSON` is the lack of possibility of commenting the file, hence, for the sake of clarity, a detailed description is in order. First of all, the file is subdivided into four main dictionary: `fluid_parameters`, `electrical_parameters` , `geometrical_parameters` and `simulation_specification`. The following is the description of all the dictionaries; we proceed with the same order of the code:

```
"fluid_parameters": {
  "viscosity": 1.8e-5,
  "gamma":1.0
},
```

- `viscosity` represents the physical viscosity of the fluid
- `gamma` is the constant that tunes the Grad-Div stabilization (see section 3.2)

```
  "electrical_parameters": {
  "eps_0": 8.854e-12,
  "eps_r": 1.0006,
  "q0": 1.602e-19,
  "kB": 1.381e-23,
  "mu0": 2e-4,
  "stratosphere": false,
  "E_ON": 7.1146e+6,
  "E_ref": 4.1436e+5,
  "N_ref": 1.0e+14,
  "N_min":1.0e+9,
  "Mm": 29.0e-3,
  "Ve":2.0e+4
},
```

- `eps_0` value of the permittivity of free space
- `eps_r` value of the relative permittivity of the media
- `q0` value of the elementary charge
- `kB` value of the Boltzman constant
- `mu0` value of the mobility coefficient
- `stratosphere` this is a boolean that enable the user to set the simulation in the stratosphere, it influences temperature, density, ambient ion density and ambient pressure variables.
- `E_ON` threshold value for the electric field in order to trigger the corona discharge phenomena
- `E_ref` value of a reference electric field used in the exponential diode boundary condition
- `N_ref` value of a reference cation density used in the exponential diode boundary condition

- `N_min` value of a minimal density in order to avoid null boundary conditions due to the difference between different electrical fields (see DD boundary condition)
- `Mm` value of the average air molar mass
- `Ve` value of the potential imposed at the emitter boundary

Before going on with the description of the last two dictionaries, we break this section to briefly describe the computational 2D domains. First of all, we would like to point out that our algorithm for the coupled problem found solutions matching those in the literature only in the case of concentric circles. This is therefore the geometry on which we performed the numerical tests reported in the final part of the report. Nevertheless, since our project is only a relatively small part of a larger and more ambitious one, we have still kept open the possibility of performing simulations on two other geometries of interest. This is the reason why in the `JSON` file it is possible to find the description of three computational domains, and (the reader will see in section(5.4)), this is also why in the coupled algorithm class we have implemented different type of boundary conditions for different geometry. The geometries of interest are the following three:

1. `NACA` geometry formed by a circular emitter and a wing-shaped collector
2. `WW` geometry formed by a circular emitter and a circular collector
3. `CYL` geometry formed by two concentric circles, where the innermost circle represents the emitter while the outermost one represents the collector

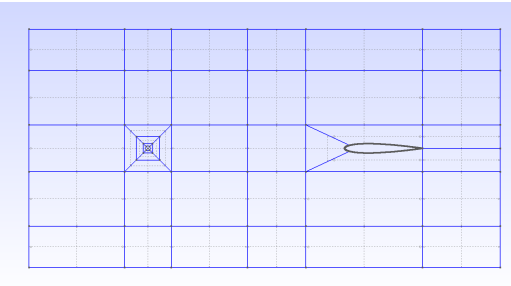


Figure 5.1: NACA geometry

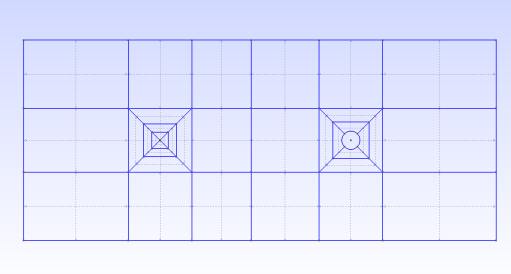


Figure 5.2: Wire-Wire geometry

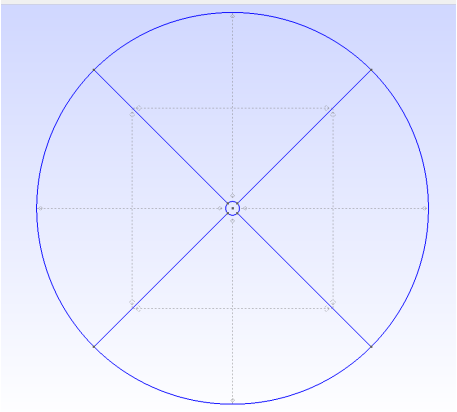


Figure 5.3: concentric circles geometry

```
    "geometrical_parameters": {
      "emitter_center_X":0.0,
      "emitter_center_Y":0.0,

     "NACA": {
        "chord_length": 0.02,
        "naca_digits": 12,
        "emitter_radius": 30e-5,
        "distance_emitter_collector": 0.03,
        "distance_trailing_edge_outlet": 0.015,
        "distance_emitter_inlet": 0.015,
        "distance_emitter_up_bottom": 0.1
    },
    "WW": {
        "emitter_radius":30e-5,
        "collector_radius":1e-3,
        "distance_emitter_collector":0.025,
        "distance_collector_outlet":0.015,
        "distance_emitter_inlet":0.015,
        "distance_emitter_up_bottom":0.01
    },
    "CYL": {
        "emitter_radius":0.7e-3,
        "collector_radius":2e-2
    }
  },
```

We think that all the names used in these dictionaries are very self-explicative, hence we do not describe all of them. We just stress that all the distances are taken starting from the surfaces of the objects, and that the so called `distance_emitter_up_bottom` represents the distance between the surface of the emitter and the up/bottom edges of the outer domain. Moreover, since in all the configurations the emitter is always a circle, the coordinates of its center are described outsite the sub-dictionaries.

```
      "simulation_specification": {
    "mesh_name":"concentric_cylinders.msh",
    "mesh_TAG":"CYL"

  }
}
```

- `mesh_name` name of the mesh on which the user perform the simulation
- `mesh_TAG` tag that specifies the type of geometry chosen by the user

5.4 Structure parallel `deal.II` code

In this section, we present all the classes, methods and functions of our `deal.II` parallel code concerning the complete `CoupledProblem`. The description of the codes that perform the validation of the two physics is redundant, therefore skipped. We will describe the process that, starting from the set up of the degrees of freedom, will lead to the solution of a finite element problem. Moreover, we will describe all the side objects that will help the main program.

5.4.1 main.cpp

We start the explanation of our code by describing the `main.cpp` file, namely the entry point of the program. The file starts with the inclusion of two header files. The first one, `config_reader.hpp`, contains the function that bridges the user specifications with the core code. It reads the variable values from the configuration `JSON` file and transmits them in a struct ready-to-use (see subsection 5.4.2 and 5.4.3). The second one, `CompleteProblem.hpp`, contains the declarations and definitions (since templates) of both class and class methods that will be used to perform all the numerical simulation.

```
#include "config_reader.hpp"
#include "CompleteProblem.hpp"
```

Then the program starts and we enter in the main. Ideally we can split the main into two sections: the first part sets up the parallel structure of the code and manages the user-defined data.

```
    int main(int argc, char** argv){

    using namespace dealii;

    Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);

    data_struct my_data; //empty struct data

    reader(my_data); //user data are accessible to all processors
```

In order to avoid all the `dealii::` specification we exploits the `using namespace` directive. We then pass the main arguments `argc` and `argv` to the `MPI` initializer, from now on all the code is processed in parallel. At this point, an empty struct of custom type `data_struct` is being created, and subsequently filled throught the function `reader` (see subsection 5.4.3). We stress that all the processors have access to the information coming from the struct.

The second section of the main is needed to handle the parallel distributed numerical operations. We exploited a try-cath block that:

- Executes the main operation of the program, which involves creating and managing a parallel distributed triangulation and running a numerical parallel simulation through an instance of `CompleteProblem`.
- Handle exceptions, displaying detailed error messages if any issue arises, ensuring the user is informed of a specific or generic error.

```
    try
        {

                parallel::distributed::Triangulation<2> tria(MPI_COMM_WORLD);

                create_triangulation(tria, my_data);

                CompleteProblem<2> problem(tria, my_data);

                problem.run();

        }
        catch (std::exception &exc)
        {
                std::cerr << std::endl
                        << std::endl
                        << "----------------------------------------------------"
                        << std::endl;
                std::cerr << "Exception on processing: " << std::endl
                        << exc.what() << std::endl
                        << "Aborting!" << std::endl
                        << "----------------------------------------------------"
                        << std::endl;
                return 1;
        }
        catch (...)
        {
                std::cerr << std::endl
                        << std::endl
                        << "----------------------------------------------------"
                        << std::endl;
                std::cerr << "Unknown exception!" << std::endl
                        << "Aborting!" << std::endl
                        << "----------------------------------------------------"
                        << std::endl;
                return 1;
        }


    return 0;
}
```

In particular, the creating of the mesh, is coded in the first two instructions. `parallel::distributed::Triangulation<dim>` is a `deal.II` class that provides a framework for managing distributed meshes (triangulations) across multiple processes in parallel. This is particularly useful for large-scale finite element computations on high-performance computing systems. The number 2 is the template input that clarifies the mathematical dimension of the problem, instead the argument passed to the object, `MPI_COMM_WORLD`, is a communicator provided by the Message Passing Interface `MPI` standard, representing the default communication group that includes all processes participating in the program. Subsequently, upon this object, the function `create_triangulation(tria, my_data)` is called for the actual creation of the computational grid. The main ends with the definition of an instance of the class `CompleteProblem`, and the start of the numerical simulation through the method `.run()`. In the next subsections we will describe all the objects and functions called inside the main. The idea is to describe each of them following the order of appearance in the main.

5.4.2 data_struct.hpp

`data_struct` is a `C++` struct data, namely an object that groups several related (possibly heterogeneous) variables into one place. This object follows the defini-

tion of the JSON configuration file, in fact the purpose was to create a structure that would store all the user data in one handful object.

```
// Compleate struct
struct data_struct {

    FluidParameters fluid_parameters;
    ElectricalParameters electrical_parameters;
    GeometricalParameters geometrical_parameters;
    SimulationSpecification simulation_specification;

};
```

As can be seen, this struct is made up of four different sub-structures that are self-explicative; each of them store a different subset of values related to the simulation.

```
// Fluid Physical Parameters Struct
struct FluidParameters {

    double viscosity;     // viscosity of the fluid
    double gamma;         // stabilizaton parameter in NS

};

// Electrical Physical Parameters Struct
struct ElectricalParameters {

    double eps_0;     //permittivity of free space [F/m]= [C^2 s^2 / kg / m^3]
    double eps_r;     //permittivity [F/m]= [C^2 s^2 / kg / m^3]
    double q0;        //unit charge [C]
    double kB;        //Boltzman constant[J/K]
    double mu0;       //Mobility of ions
    bool stratosphere; //bool, if false use atmospheric 0 km condition
    double E_ON;      // onset field threshold [V/m]
    double E_ref;     // maximum field value   [V/m]
    double N_ref;     // maximum density value [m^-3]
    double N_min;
    double Mm;        // average air molar mass [kg m^-3]
    double Ve;        // emitter voltage [V]

};


//Simulations infos Struct
struct SimulationSpecification {

   std::string mesh_name;
   std::string mesh_TAG;

};
```

A more articulate struct is used to describe the geometry of the computational domain. As we elaborated before, we focused on three different geometry and, for each of them, we decided to create a specific struct that describes its peculiarities. The following are the three different structs:

```
// Geometrical infos Struct
struct NACA {

    double chord_length;                  // length of the chord [m]
    int naca_digits;                      // type of naca
    double emitter_radius;                // radius of the circular emitter [m]
    double distance_emitter_collector;    // distance circular emitter surface,
        airfoil collector surface
    double distance_trailing_edge_outlet;  // distance trailing edge outlet
    double distance_emitter_inlet;         // distance circular emitter surface and
         inlet
```

```
    double distance_emitter_up_bottom;     // distance circular emitter surface and
        up/bottom boundary

};


struct WW {

    double emitter_radius;                 // radius of the circular emitter [m]
    double collector_radius;               // radius of the circular collector [m]
    double distance_emitter_collector;     // distance circular emitter surface,
        airfoil collector surface
    double distance_collector_outlet;      // distance trailing edge outlet
    double distance_emitter_inlet;         // distance circular emitter surface and
        inlet
    double distance_emitter_up_bottom;     // distance circular emitter surface and
        up/bottom boundary

};

struct CYL {

    double emitter_radius;                 // radius of the circular emitter [m]
    double collector_radius;               // radius of the circular collector [m]

};
```

And finally the struct that wraps all the geometrical information:

```

 struct GeometricalParameters {
   double emitter_center_X;  // X coordinate of the emitter center [m]
   double emitter_center_Y;  // Y coordinate of the emitter center [m]
   NACA naca;
   WW ww;
   CYL cyl;

};
```

5.4.3 config_reader.hpp and config_reader.cpp

The aim of the function `reader(data_struct& data)` is to store in the previously described `data_struct`, all the information specified in the configuration `JSON` file. The header file is fairly easy, it contains the usual header guards, a set of "includes", an alias for the namespace of the library `nlohmann::json` and the declaration of the function.

```
#ifndef CONFIG_READER_HPP
#define CONFIG_READER_HPP

#include "data_struct.hpp"
#include <string>
#include <iostream>
#include <fstream>
#include <json.hpp>

using json = nlohmann::json;

void reader(data_struct& data);

#endif //CONFIG_READER_HPP
```

In order to fill the struct, we pass it by reference. The reader will find in the description of the file `main.cpp` the creation of a blank `data_struct` and subsequently the call to `reader` function. We pass now to the description of the

definition of the function. First of all, we created on-fly a file stream to read the `JSON` file, we then check, read and close the file stream:

```
void reader(data_struct& data){

// Open a file stream for reading the json file
std::ifstream inFile("../config/configuration.json");

// Check if the file stream is open
if (!inFile.is_open()) {
std::cerr << "Failed to open the file for reading." << std::endl;
return ;
}

// Read JSON data from the file
json dati_json;
inFile >> dati_json;

// Close the file stream
inFile.close();
```

Once we have in the file stream all the data, we start to fill the passed-by-reference struct. The syntax is the same for every variable, and it exploits both the method `.at()` and `.get_to()` of the `nlohmann::json` library. We leave here an example of how to use these methods, but we stress that it is the same for the rest of the other variables.

```
dati_json.at("fluid_parameters").at("gamma").get_to(data.fluid_parameters.gamma);
```

We go through all the items inside `configuration.json`, and then we `return;`. Now we have a structure that holds all the user-defined information for our simulations.

5.4.4 CollectorGeometry.hpp

The header file `CollectorGeometry.hpp` contains the definition of the function `create_triangulation`, designed to initialize a parallel distributed triangulation (mesh) based on specific geometrical configurations (NACA airfoil, wire-wire or concentric cylindrical geometries) defined in an input data structure (`s_data`). This function is called in the `main.cpp` file (see subsection 5.4.1). The function starts by defining an unordered map. It stores key-value pairs of type `<std::string,int>` that are used to identify the correct mesh to initialize; for each type (tag) of mesh we created an unique integer related to a `switch` case. The input data struct, passed by const reference, specifies the chosen tag and then, thanks to the method `.find(input)` an iterator pointing to the correct integer is created. The iterator is then used to choose the correct `switch` case:

```
void create_triangulation(parallel::distributed::Triangulation<2> &tria,
                          const data_struct& s_data)
{

  std::unordered_map<std::string, int> stringToCase{
  {"NACA", 1},
  {"WW", 2},
  {"CYL", 3}
  };

  const std::string input = s_data.simulation_specification.mesh_TAG;
  auto iter = stringToCase.find(input);

  if (iter != stringToCase.end()) {
    switch (iter->second) {
```

For the sake of synthesis we report the code of a specific `switch` case. The other cases have a similar structure with very little differences, this is why we just focus on the following one:

```
case 1:{

    std::string name_mesh = s_data.simulation_specification.mesh_name;

    std::string filename = "../../meshes/"+name_mesh;

    std::cout <<"   Reading the mesh from " << filename << std::endl;

    std::ifstream input_file(filename);
    GridIn<2>       grid_in;
    grid_in.attach_triangulation(tria);
    grid_in.read_msh(input_file);

    const types::manifold_id emitter = 3;
    const types::manifold_id collector = 4;

    const double distance_emitter_collector = s_data.geometrical_parameters.naca.
        distance_emitter_collector;
    const double r_emi = s_data.geometrical_parameters.naca.emitter_radius;
    double X = -distance_emitter_collector - r_emi;
    const Point<2> center(X,0.0);

    SphericalManifold<2> emitter_manifold(center);

    CollectorGeometry<2> collector_manifold;

    tria.set_all_manifold_ids_on_boundary(3, emitter);
    tria.set_manifold(emitter, emitter_manifold);
    tria.set_all_manifold_ids_on_boundary(4, collector);
    tria.set_manifold(collector, collector_manifold);

    std::cout  << "   Active cells: " << tria.n_active_cells() << std::endl;

    break;
}
```

The function constructs the mesh filename by prepending the directory `../meshes/` to the name, then reads the mesh data from this file using the `GridIn<2>` class. This `deal.II` class implements an input mechanism for grid data, it allows to read a grid structure into a triangulation object. Then, two manifold identifiers are defined for specific regions: the emitter (ID 3) and the collector (ID 4). Subsequently, some domain specifications are retrieved from the input structure in order to center the manifolds. A `SphericalManifold<2>` centered at the computed `Point<2>` represents the emitter's shape, while `CollectorGeometry<2>` represents the collector's custom profile. `CollectorGeometry<dim>` is a custom template class that defines a manifold object with peculiar and possibly complex shape, in this case, the profile of a 2D NACA airfoil. The code then applies these manifolds to the mesh boundaries, assigning IDs and setting boundary properties accordingly. Finally, it prints the number of active cells in the mesh. As we stress before, this is only one mesh type, the other two differs in the type of `Manifold` exploited and obviously in the geometric features.

5.4.5 BlockSchurPreconditioner.hpp and BlockSchurPreconditioner_impl.hpp

The `BlockSchurPreconditioner` class provides a specialized preconditioner for solving block-structured linear systems.

```
class BlockSchurPreconditioner : public Subscriptor
{
public:
  BlockSchurPreconditioner(
    TimerOutput &timer,
    double gamma,
    double viscosity,
    double dt,
    const std::vector<IndexSet> &owned_partitioning,
    const PETScWrappers::MPI::BlockSparseMatrix &system,
    const PETScWrappers::MPI::BlockSparseMatrix &mass,
    PETScWrappers::MPI::BlockSparseMatrix &schur);

  void vmult(PETScWrappers::MPI::BlockVector &dst,
             const PETScWrappers::MPI::BlockVector &src) const;

private:

  TimerOutput &timer;
  const double gamma;
  const double viscosity;
  const double dt;

  const SmartPointer<const PETScWrappers::MPI::BlockSparseMatrix> system_matrix;
  const SmartPointer<const PETScWrappers::MPI::BlockSparseMatrix> mass_matrix;
  const SmartPointer<PETScWrappers::MPI::BlockSparseMatrix> mass_schur;
};
```

It inherits from `Subscriptor` for safe object management and is initialized with all the fluid parameters which influence the preconditioning strategy: `gamma`, `viscosity`, and `dt` (time step size). The constructor takes references to `TimerOutput` for performance tracking and `IndexSet` for partitioning across parallel processes, as well as three block sparse matrices: `system_matrix`, `mass_matrix`, and `mass_schur`, which represent the primary system matrix, mass matrix, and Schur complement matrix, respectively. These matrices are stored as `SmartPointers` for efficient memory management and access. The core of the class is the `vmult` method, which performs the matrix-vector multiplication necessary for applying the preconditioner by taking an input vector src and writing the result to dst, facilitating the solution of large systems in iterative solvers.

The `BlockSchurPreconditioner::vmult` method applies a Block Schur preconditioning operation to an input vector `src`, storing the result in `dst`. It begins by creating temporary vectors.

```
void BlockSchurPreconditioner::vmult(
    PETScWrappers::MPI::BlockVector &dst,
    const PETScWrappers::MPI::BlockVector &src) const
  {

    // Temporary vectors
    PETScWrappers::MPI::Vector utmp(src.block(0));
    PETScWrappers::MPI::Vector tmp(src.block(1));
    tmp = 0;

    {
      TimerOutput::Scope timer_section(timer, "CG for Mp");
      SolverControl mp_control(10*src.block(1).size(),
                               5e-1 * src.block(1).l2_norm(), true);

      PETScWrappers::SolverBicgstab cg_mp(mp_control);

      PETScWrappers::PreconditionBlockJacobi Mp_preconditioner;
      Mp_preconditioner.initialize(mass_matrix->block(1, 1));
      cg_mp.solve(mass_matrix->block(1, 1), tmp, src.block(1), Mp_preconditioner);
```

```
      tmp *= -(viscosity + gamma);

    }
```

In the first block, a BiCGStab solver with a block Jacobi preconditioner is used to solve the linear system involving `mass_matrix->block(1, 1)`, updating `tmp` by scaling with the negative sum of viscosity and gamma to incorporate their stabilizing effects.

```
      {
      TimerOutput::Scope timer_section(timer, "CG for Sm");
      SolverControl sm_control(10*src.block(1).size(),
                               5e-1 * src.block(1).l2_norm(), true);

      PETScWrappers::SolverBiCG cg_sm(sm_control);

      PETScWrappers::PreconditionNone Sm_preconditioner;
      Sm_preconditioner.initialize(mass_schur->block(1, 1));
      cg_sm.solve( mass_schur->block(1, 1), dst.block(1), src.block(1),
          Sm_preconditioner);
      dst.block(1) *= -1 / dt;

    }
```

In the second block, a CG solver with no preconditioner handles the Schur complement `mass_schur->block(1, 1)`, modifying `dst.block(1)` and scaling it by $\frac{-1}{dt}$. Next, `tmp` is added to `dst.block(1)`, ensuring contributions from both solves are included. The method then computes `utmp`.

```
    dst.block(1) += tmp;

    system_matrix->block(0, 1).vmult(utmp, dst.block(1));
    utmp *= -1.0;
    utmp += src.block(0);

    {

      TimerOutput::Scope timer_section(timer, "CG for A");
      SolverControl a_control(10*src.block(0).size(),
                              5e-1 * src.block(0).l2_norm(), true);

      PETScWrappers::SolverBicgstab cg_a(a_control);
      PETScWrappers::PreconditionNone A_preconditioner;
      A_preconditioner.initialize(system_matrix->block(0, 0));
      cg_a.solve(system_matrix->block(0, 0), dst.block(0), utmp, A_preconditioner);

    }
  }
```

Finally, a BiCGStab solve with no preconditioner is performed on `system_matrix->block(0, 0)` to determine `dst.block(0)`, completing the preconditioning by capturing dependencies between system components and improving solver convergence for coupled systems.

5.4.6 BoundaryValues.hpp

```
 template <int dim>
class BoundaryValues : public Function<dim>
  {
  public:

    // Default constructor that initializes inlet_value to 0.0
    BoundaryValues() : Function<dim>(dim + 1), inlet_value(0.0) {}

```

```
    // Constructor that takes a double parameter to initialize inlet_value
    BoundaryValues(double inlet_val) : Function<dim>(dim + 1), inlet_value(
        inlet_val) {}

    double inlet_value;

    virtual double value(const Point<dim> &p,
                         const unsigned int component) const override;
    virtual void vector_value(const Point<dim> &p,
                              Vector<double> &values) const override;
  };
```

The `BoundaryValues` class template defines a set of boundary conditions for a fluid simulation in a bidimensional space, inheriting from `Function<dim>` to represent spatially varying boundary conditions for a system with three components (two velocities and one pressure).

```
template <int dim>
double BoundaryValues<dim>::value(const Point<dim> & /*p*/,
                                  const unsigned int component) const
{
 Assert(component < this->n_components,
    ExcIndexRange(component, 0, this->n_components));

 if (component == 0)
  return inlet_value;

 if (component == dim)
    return 0.;      // Boundary condition at fluid outlet

 return 0.;
}


template <int dim>
void BoundaryValues<dim>::vector_value(const Point<dim> &p,
                                       Vector<double> &values) const
  {
    for (unsigned int c = 0; c < this->n_components; ++c)
      values(c) = BoundaryValues<dim>::value(p, c);
  }
```

It contains an `inlet_value` member, initialized to either 0.0 by the default constructor or to a specified value through a parameterized constructor, to set the boundary condition at the domain's inlet. The `value` method, which overrides `Function<dim>::value`, returns the boundary condition value at a given point p for a specified component: if component is 0, it returns inlet value, representing an inlet condition (such as initial velocity or pressure); if component equals dim, it returns 0.0, which could represent a zero-gradient or outflow condition at the outlet; and for other components, it defaults to 0.0, indicating a neutral or homogeneous boundary. Additionally, the `vector_value` method provides boundary values across all components at a point p by iterating over each component, calling value for each, and populating the provided values vector accordingly. This setup allows for customized inlet and outlet conditions.

5.4.7 CompleteProblem.hpp and CompleteProblem_impl.hpp

In this subsection we describe the template class that defines the physical problem we want to solve. The `CompleteProblem` class is the core object in our code; its purpose is to manage the numerical simulation and generate output

to inspect through `ParaView`. The template parameter is `<dim>` and it specifies the mathematical dimension of the physical problem. Even if our methods are oriented for 2D cases only, we decided to give more generality to the code by writing a template class, in this way we make it easier the extension to the third dimension. The class is rich of members object coming from `deal.II` and methods performing specific tasks during the evolution of the simulation. In the following, we will describe different parts of the class, we invite the reader to look at the GitHub repository for the complete code. The reader will notice that most of the methods are private, in particular, only two methods are public: the constructor and the `run` method that start the simulation. This choice is due to the fact that the user interface is intentionally minimized; a hypothetical user, can only create an instance of this class, start a simulation or eventually changing the data in the `JSON` configuration file.

We start the description of the class from its private members.

FiniteElement<dim>

First of all we need objects capable of describing the types of elements we want to use in our simulations. In particular, we can identify two types of objects: `FE_Q<dim>` and `FESystem<dim>`, both inheriting from the abstract class `FiniteElement<dim>`.

```
1    FE_Q<dim>         fe;      // electrical problem
2    FESystem<dim>     NS_fe;   // fluid problem
```

The former class provides an implementation of the scalar Lagrange Q_p finite element for the electrical problem. Our choice is Q_1. The latter class provides an interface to group several elements together into one, vector-valued element for the fluid problem. We will consider the Taylor-Hood element: the velocity (of which there are as many components as the dimension `<dim>`, the template input) is discretized with Q_2 elements and the pressure with Q_1 elements. This choice satisfy the inf-sup condition, thus, we do not need further stabilizing terms in the Navier-Stokes system. Even if pressure and electrical quantities (potential and ion density) share the degrees of freedom (DOFs) on the elements, we still need two `FiniteElement<dim>` objects to perform the simulation.

DoFHandler<dim>

The next object we introduce is a class capable of linking the given triangulation with the type of elements presented in the previous subsection:

```
1    DoFHandler<dim>  dof_handler;     // electrical handler
2    DoFHandler<dim>  NS_dof_handler;  // Navier-Stokes handler
```

When we set up the degrees of freedom, the fundamental structure is `DoFHandler<dim>` class. Given a triangulation and a description of a finite element, this class enumerates degrees of freedom on all vertices, edges, faces, and cells of the triangulation. As a result, it also provides a basis for a discrete space V_h whose elements are finite element functions defined on each cell.

Parallel Objects

In this subsection we present all the fundamental objects for a parallel simulation and the philosophy behind the jobs division. Inside the class we specify the `MPI` communicator, an object that defines a group of processes that can communicate with each other during the run. We decided to employ the most commonly used communicator: `MPI_COMM_WORLD`, which includes all the processes that are part of the `MPI` program. We then added a `ConditionalOStream` object. Ordinarily, a developer would use `std::cout` to print messages in the terminal. However, in parallel programs, this means that each of the `MPI` processes write to the screen, which yields many repetitions of the same text. To avoid it, the present class can be used: objects of its type act just like a standard output stream, but they only print something based on a condition that can be set (in our case we decided to set rank zero as the printing processor).

```

    // Parallel data

    MPI_Comm mpi_communicator;
    ConditionalOStream pcout;
```

The next fundamental object for a parallel code is `parallel::distributed::Triangulation<dim>`, this class relies on the `p4est` library and distributes the mesh across a number of different processors. In essence, what the class and code inside `DoFHandler`, do is to split the global mesh so that every processor only stores a subset of the cells along with one layer of "ghost" cells that surround the ones it owns. What happens in the rest of the domain on which we want to solve the partial differential equation is unknown to each processor and can only be inferred through communication with other machines if such information is needed. This implies that no processor can have the entire solution vector for postprocessing, for example, and every part of a program has to be parallelized because no processor has all the information necessary for sequential operations.

```

    // Object for the mesh
    parallel::distributed::Triangulation<dim> &triangulation;
```

To conclude this part we introduce the class `IndexSet`, a class that represents a subset of indices among a larger set. In our code they are used to denote the set of degrees of freedom that are stored on a particular processor.

```

    // Electrical Indexsets
    IndexSet locally_owned_dofs;
    IndexSet locally_relevant_dofs;

    // Fluid Indexsets
    IndexSet owned_partitioning_p;
    IndexSet NS_locally_relevant_dofs;

    std::vector<IndexSet> owned_partitioning;
    std::vector<IndexSet> relevant_partitioning;
```

`locally_owned_DoFs` and `owned_partitioning` are the `IndexSet` that specify the indexes of the degrees of freedom that lives on locally owned cells, namely

cells stored in the current processor. Instead, `locally_relevant_dofs` those that live on locally owned or ghost cells. Consequently, inside the latter object we can find also cells that are "owned" by other processors. Ghost cell are fundamental in order to perform computation at the boundary between cells owned by different processors.

Vector, BlockVector, SparseMatrix and BlockSparseMatrix

All the computed values during a simulation are either stored inside a `PETScWrappers::MPI::Vector` or `PETScWrappers::MPI::SparseMatrix`. These are basically implementations of parallel vector and parallel sparse matrix classes based on `PETSC` and using `MPI` communication to synchronize distributed operations. These classes follow a parallel philosophy, hence if one process wants something from another, that other process has to be willing to accept this communication. A process cannot query data from another process by calling a remote function, without that other process expecting such a transaction. The consequence is that most of the operations have to be called collectively. We also exploited the "Block" version of these classes in order to deal with the fluid problem, where a division of velocities and pressure DOFs is employed. Inside our class we use a lot of these objects: the vectors store the unknowns and the right-hand-sides of systems, while the sparse matrix objects store all the matrices coming from the numerical discretization. The variables have self-explicative names, however, we will explain them when computed inside the methods of the class.

```
1
2       // -- Electrical Part --
3
4       // Poisson Matrices
5     PETScWrappers::MPI::SparseMatrix laplace_matrix_poisson;
6     PETScWrappers::MPI::SparseMatrix mass_matrix_poisson;
7     PETScWrappers::MPI::SparseMatrix system_matrix_poisson;
8     PETScWrappers::MPI::SparseMatrix density_matrix;
9
10    PETScWrappers::MPI::SparseMatrix initial_matrix_poisson;
11
12    // Drift-Diffusion Matrices
13    PETScWrappers::MPI::SparseMatrix ion_system_matrix;
14    PETScWrappers::MPI::SparseMatrix ion_mass_matrix;
15
16    // Poisson Vectors
17    PETScWrappers::MPI::Vector poisson_newton_update;
18    PETScWrappers::MPI::Vector potential;
19    PETScWrappers::MPI::Vector poisson_rhs;
20
21    PETScWrappers::MPI::Vector initial_poisson_rhs;
22
23    // Electric fields
24    PETScWrappers::MPI::Vector Field_X;
25    PETScWrappers::MPI::Vector Field_Y;
26
27    // Drift-Diffusion Vectors
28    PETScWrappers::MPI::Vector old_ion_density;
29    PETScWrappers::MPI::Vector ion_density;
30    PETScWrappers::MPI::Vector ion_rhs;
31    PETScWrappers::MPI::Vector eta;
32
33    // -- Fluid Part --
34
35    // Vectors
36    PETScWrappers::MPI::Vector Vel_X;
37    PETScWrappers::MPI::Vector Vel_Y;
```

```
38      PETScWrappers::MPI::Vector pressure;
39
40      PETScWrappers::MPI::Vector current_values;
41
42      // BlockMatrices
43      BlockSparsityPattern      NS_sparsity_pattern;
44      PETScWrappers::MPI::BlockSparseMatrix  NS_system_matrix;
45      PETScWrappers::MPI::BlockSparseMatrix  pressure_mass_matrix;
46      PETScWrappers::MPI::BlockSparseMatrix  NS_mass_matrix;
47
48      // BlockVectors
49      PETScWrappers::MPI::BlockVector NS_solution;
50      PETScWrappers::MPI::BlockVector NS_update;
51      PETScWrappers::MPI::BlockVector NS_solution_update;
52      PETScWrappers::MPI::BlockVector NS_system_rhs;
```

Before moving on to the next object, let us take this opportunity to highlight the fact that there are actually two types of vectors:

- vectors that take into account ghost cells
- vectors that does not take into account ghost cells

In our implementation we exploited both to solve two different tasks; vectors having ghost cells are necessary to see the plot of the solution, while vectors that have not ghost cells are useful for computation and element-wise access.

AffineConstraints<number>

We pass now to describe the template class capable of imposing linear constraints (possibly inhomogeneous) on degrees of freedom. For both fluid and electrical problems we exploited instances of `AffineConstraints<number>` class, creating different types of boundary conditions.

```
1
2       // electrical constraints
3       AffineConstraints<double> ion_constraints;
4       AffineConstraints<double> zero_constraints_poisson;
5       AffineConstraints<double> constraints_poisson;
6       AffineConstraints<double> constraints_poisson_update;
7
8       // fluid constraints
9       AffineConstraints<double> zero_NS_constraints;
10      AffineConstraints<double> nonzero_NS_constraints;
```

We refer to the official documentation for an extensive explanation of this class, where the concept and origin of such constraints is extensively described. The class is meant to deal with a limited number of constraints relative to the total number of degrees of freedom. It is not meant to describe full rank linear systems. Each "line" in objects of this class corresponds to one constrained degree of freedom. We employed these objects for the imposition of Dirichlet boundary conditions.

POD, struct and MappingQ1<dim>

We conclude the description of the private members by presenting the last data that are managed by the class. First of all, since we need to take into consideration time passing, we store two `double` that represent two different time

stepping. We follow this procedure because the two physical phenomena (fluid dynamics and ion transport) have very different time evolution. Hence, we set:

```
1
2     double timestep_NS = 40*1e-5;  // NS time step
3     double timestep = 1e-5;        // DD time step
4     mutable TimerOutput timer;
```

Since the Navier-Stokes system is solved only every 40 drift-diffusion time steps, it has to advance in time with a bigger time variable. We describe time flow not only with POD (plain old data), but we also exploited the class `TimerOutput`, this class can be used to generate formatted output from time measurements of different subsections in a program.

The only struct stored inside the class is the custom made `data_struct` described in the previous section. The idea is to save inside it the user defined specifications that will be used during the simulations. To conclude the description of the private members, we introduce the `MappingQ1<dim>` template class.

```
1
2     // Data for the simulation
3     data_struct m_data;
4
5     // Mapping
6     MappingQ1<dim>  mapping;    // DD mapping
7     MappingQ1<dim> NS_mapping;  // NS mapping
```

Implementation of a dim-linear mapping from the reference cell to a general quadrilateral/hexahedron.

With `MappingQ1<dim>` we finish the description of the private members, we now proceed with the detailed explanation of all the methods, both private and public.

Constructor

In the public part of the class definition the user will find:

```
1 CompleteProblem(parallel::distributed::Triangulation<dim> &tria,
2                 const data_struct &d);
```

We start of corse with the only implemented constructor that the class posses. This is a public method that creates an instance of the class, and sets some of the previously described member objects. It requires two inputs:

1. a parallel distributed triangulation (created through the procedure explained in 5.4.4)
2. an instance of `data_struct`

The idea behind this method is fairly simple: from the input references the function takes information and store them inside the instance of the object:

```
1 template <int dim>
2 CompleteProblem<dim>::CompleteProblem(parallel::distributed::Triangulation<dim> &
      tria,  const data_struct &d)
3   : m_data(d)
4   , mpi_communicator(MPI_COMM_WORLD)
5   , pcout(std::cout, (Utilities::MPI::this_mpi_process(mpi_communicator) == 0))
```

```
  , triangulation(tria)
  , fe(1)
  , dof_handler(tria)
  , mapping()
  , viscosity(d.fluid_parameters.viscosity)
  , gamma(d.fluid_parameters.gamma)
  , degree(1)
  , NS_fe(FE_Q<dim>(degree+1), dim, FE_Q<dim>(degree), 1)
  , NS_dof_handler(tria)
  , volume_quad_formula(degree + 2)
  , face_quad_formula(degree + 2)
  , NS_mapping()
  , timestep(1e-5)
  , timestep_NS(40*1e-5)
  , timer(mpi_communicator, pcout, TimerOutput::never, TimerOutput::wall_times)
{}
```

In particular, we stress some default values that we choose for all the simulations, thus, specification that the user can not change! For instance, the parallel communicator for `MPI` is always `MPI_COMM_WORLD`, as well as the degree of the elements are Q_1 for the electrical variables and Q_2, Q_1 respectively for velocity and pressure. `face_quad_formula` and `volume_quad_formula` are objects that are used for computing integrals, and both of them have a degree equal to three. We decided also to fix the time step. We did not implement neither the copy-constructor nor the copy assignment one.

Setup methods

We have three different setup methods, one for each algebraic system to solve:

1. non-linear Poisson
2. drift-diffusion
3. Navier-Stokes

```

    void setup_poisson();
    void assemble_initial_system();
    void setup_NS();
```

All the three methods share common setup operations, for instance the linking between `DoFHandler<dim>` and `FiniteElement<dim>` classes through the method `.distribute()`, the differentiation between locally owned and locally relevant degrees of freedom and initialization of vectors.

```

//setup_poisson

dof_handler.distribute_dofs(fe);

locally_owned_dofs = dof_handler.locally_owned_dofs(); //local dofs
locally_relevant_dofs = DoFTools::extract_locally_relevant_dofs(dof_handler);    //
    local dofs + ghost dofs

potential.reinit(locally_owned_dofs,
                 locally_relevant_dofs,
                 mpi_communicator);   //ghosted
poisson_newton_update.reinit(locally_owned_dofs,
                             mpi_communicator);    //non-ghosted
poisson_rhs.reinit(locally_owned_dofs,
                   mpi_communicator);  //non-ghosted

```

```

// setup NS
NS_dof_handler.distribute_dofs(NS_fe);

std::vector<unsigned int> block_component(dim + 1, 0);
block_component[dim] = 1;
DoFRenumbering::component_wise(NS_dof_handler, block_component);

dofs_per_block = DoFTools::count_dofs_per_fe_block(NS_dof_handler, block_component)
    ;

unsigned int dof_u = dofs_per_block[0];
unsigned int dof_p = dofs_per_block[1];

owned_partitioning.resize(2);
owned_partitioning[0] = NS_dof_handler.locally_owned_dofs().get_view(0, dof_u);
owned_partitioning[1] = NS_dof_handler.locally_owned_dofs().get_view(dof_u, dof_u +
     dof_p);

DoFTools::extract_locally_relevant_dofs(NS_dof_handler,NS_locally_relevant_dofs );

relevant_partitioning.resize(2);
relevant_partitioning[0] = NS_locally_relevant_dofs.get_view(0, dof_u);
relevant_partitioning[1] = NS_locally_relevant_dofs.get_view(dof_u, dof_u + dof_p);
```

As the reader can see, the setup of Navier-Stokes related variables is a bit more complex since it has to take into account the vectorial nature of the problem. Moreover, we split into block components velocity and pressure degrees of freedom. Inside these methods we can also find the setup of boundary conditions and sparse matrices. For the sake of brevity, as before, we report only one example, but the procedure is general and it is indeed applied for all the different Dirichlet boundary conditions.

```
//setup poisson
constraints_poisson.clear();
constraints_poisson.reinit(locally_relevant_dofs);

VectorTools::interpolate_boundary_values(dof_handler,
                                         3,
                                         Functions::ConstantFunction<dim>(Ve),
                                             constraints_poisson); //emitter

VectorTools::interpolate_boundary_values(dof_handler,
                                          4,
                                          Functions::ZeroFunction<dim>(),
                                              constraints_poisson); // collector

constraints_poisson.close();
```

We clear the object, then we exploited the function `interpolate_boundary_values` in order to impose a specific value on a specific edge. This function uses the dofhandler to identify the degrees of freedom on the boundary tagged by the specified id which is the boundary subject to Dirichlet boundary conditions. After that, the method `.close()` is used to end the procedure. Right now inside the `AffineConstraints` object all the information related to a specific boundary condition are stored. We then use this instance to create a `SparsityPattern` and initialize the related matrices:

```

// setup Poisson

DynamicSparsityPattern dsp_init(locally_relevant_dofs);

DoFTools::make_sparsity_pattern(dof_handler,
```

```
                                dsp_init,
                                constraints_poisson,
                                false);

SparsityTools::distribute_sparsity_pattern(dsp_init,
                                           dof_handler.locally_owned_dofs(),
                                           mpi_communicator,
                                           locally_relevant_dofs);

initial_matrix_poisson.clear();
initial_matrix_poisson.reinit(locally_owned_dofs,
                              locally_owned_dofs,
                              dsp,
                              mpi_communicator);
```

In conclusion, the setup methods in the `CompleteProblem` class perform the following tasks:

1. distribute the finite element type on the mesh
2. differentiate between owned and relevant DOFs
3. initialize vectors with the correct size, separating them in ghosted and non-ghosted vectors
4. defines boundary conditions
5. initialize sparse matrices with the correct size

Assemble methods

In total we have seven assemble methods inside our template class, however, we can divide them into two main categories:

1. Assemble methods building a single sparse matrix
2. Assemble methods building a sparse matrix with the respective right-hand-side, namely a system

In particular the former category is characterize by:

```
    void assemble_poisson_laplace_matrix();
    void assemble_poisson_mass_matrix();
    void assemble_drift_diffusion_mass_matrix();
```

Instead, the latter by:

```
    void assemble_initial_system();
    void assemble_nonlinear_poisson();
    void assemble_drift_diffusion_matrix();
    void assemble_NS(bool use_nonzero_constraints, bool assemble_system);
```

We begin the description with the first category of methods, as an example and for the sake of brevity, we report here the building process related only to `assemble_poisson_mass_matrix()`, however the structure is very similar for the other two methods.

```
template <int dim>
void CompleteProblem<dim>::assemble_poisson_mass_matrix()
{
const QTrapezoid<dim> quadrature_formula;

mass_matrix_poisson = 0;
```

```
7
8  FEValues<dim> fe_values(fe,
9              quadrature_formula,
10             update_values | update_gradients |
11             update_quadrature_points | update_JxW_values);
12
13 const unsigned int dofs_per_cell = fe.n_dofs_per_cell();
14 const unsigned int n_q_points    = quadrature_formula.size();
15
16 FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
17
18 std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
19
20 unsigned int q_point = 0, idof = 0, jdof = 0;
```

We start by defining the quadrature rule that we will employ for the computation of the integrals and by setting all elements of the matrix to zero. We take the opportunity to stress out that we will always employ the trapezoidal rule for the computation of integrals. Subsequently we introduce `FEValues<dim>`, a template class that evaluate finite elements in quadrature points of a cell. Finally, we create a local matrix to store the contributions coming from every single cell, a local set of indexes to retrieve the DOFs related to the cell and we initialize some indexes.

```
1  for (const auto &cell : dof_handler.active_cell_iterators()){
2      if (cell->is_locally_owned())
3  {
4      cell_matrix = 0.;
5
6      fe_values.reinit(cell);
7
8      for (q_point = 0; q_point < n_q_points; ++q_point) {
9      for (idof = 0; idof < dofs_per_cell; ++idof) {
10         for (jdof = 0; jdof < dofs_per_cell; ++jdof)
11     cell_matrix(idof, jdof) += fe_values.shape_value(idof, q_point) *
12         fe_values.shape_value(jdof, q_point) * fe_values.JxW(q_point);
13     }
14     }
15
16     cell->get_dof_indices(local_dof_indices);
17     zero_constraints_poisson.distribute_local_to_global(cell_matrix,
18                                                         local_dof_indices,
19                                                         mass_matrix_poisson );
20 }
21 }
22
23 mass_matrix_poisson.compress(VectorOperation::add);
24
25 }
```

The second part of the method is pretty straightforward: for each cell in the `DoFHandler` object, if the cell is owned by the current processor, we perform some computations. First of all, we set to zero the value related to the current owned cell, then we cycle over the quadrature points `q_point`, row index i and column index j. Inside the nested loops we can find the specific computation related to the matrix, in this particular case the user could recognize the integral of the ij-th term of a mass matrix. The method `.shape_value(idof,q_point)` returned the value of the i-th base function on the quadrature point `q_point`, the same goes for the function specifying the j-th component. The last method, `.JxW(q_point)`, represents the Jacobi determinant times the weight of the quadrature node. Finally, we retrieve the indexes related to the current cell, and we apply the boundary condition by exploiting the `distribute_local_to_global()` method.

This function not only impose the specified boundary conditions, it also distributes the computed value in the right position in the global matrix (note that the global matrix is still distributed across the different processors!). The method ends with the call to `compress`, which goes through the data associated with ghost indices and communicates it to the owner process, which can then add it to the correct position. We stress again that both the methods `assemble_poisson_laplace_matrix` and `assemble_drift_diffusion_mass_matrix` follow the same strategy with a different computation inside the nested for loops.

Now, we carefully describe the four methods that build the different algebraic systems.

`assemble_initial_system`

One of the few drawbacks of the Newton method is the necessity of a good initial value in order to converge. The purpose of this method is to assemble an easy algebraic system (both matrix and right-hand-side) to be solved, in order to obtain the first vector for the Newton method. The function follows the philosophy described in the previous section, with a couple of differences:

```
template <int dim>
void CompleteProblem<dim>::assemble_initial_system()
{

initial_poisson_rhs = 0;

Vector<double>     cell_rhs(dofs_per_cell);
```

At the beginning of the method we set to zero also the right-hand-side of the system, moreover, we create a vector taking care of all the computation that we will perform on the local cells.

```

  if (cell->is_locally_owned()){
      cell_matrix = 0.;
      cell_rhs = 0.;
```

As before, if the local cell is owned we enter in the loops, but this time we need to reset also the local RHS.

```

for (jdof = 0; jdof < dofs_per_cell; ++jdof){
                    cell_matrix(idof, jdof) += fe_values.shape_grad(idof, q_point)
                        *
                    fe_values.shape_grad(jdof, q_point) * fe_values.JxW(q_point);
                }
```

This time we compute the local terms of the Laplace matrix, which will be used to find the initial vector.

```

cell->get_dof_indices(local_dof_indices);
constraints_poisson.distribute_local_to_global(cell_matrix,
                                               cell_rhs,
                                               local_dof_indices,
                                               initial_matrix_poisson,
                                               initial_poisson_rhs);
    }
 }

```

```
initial_matrix_poisson.compress(VectorOperation::add);
initial_poisson_rhs.compress(VectorOperation::add);

}
```

Finally, we distribute the boundary condition on the local system and we fill, term by term, the global one, this time creating both the matrix and the right-hand-side. As before we call `compress()` to fix the ghost nodes. In this way we created our first linear system to solve.

`assemble_nonlinear_poisson`

Once the initial system was generated and solved, we worried about creating a function able to build the Newton algebraic system.

```
template <int dim>
void CompleteProblem<dim>::assemble_nonlinear_poisson()
{

// Fix the constants
const double q0 = m_data.electrical_parameters.q0;
const double eps_r = m_data.electrical_parameters.eps_r;
const double eps_0 = m_data.electrical_parameters.eps_0;
const double kB = m_data.electrical_parameters.kB;
const double T = m_data.electrical_parameters.stratosphere ? 217. : 303.;
const double V_TH = kB*T/q0;

//BUILD NEWTON MATRIX

system_matrix_poisson = 0;
density_matrix = 0; // we store in a  matrix the ion density

double new_value = 0;

// Generate the term:  (eta)*MASS_MAT   lumped version
for (auto iter = locally_owned_dofs.begin(); iter != locally_owned_dofs.end(); ++
    iter){

  new_value = mass_matrix_poisson(*iter, *iter) * eta(*iter);
  density_matrix.set(*iter,*iter,new_value);

}

density_matrix.compress(VectorOperation::insert);
```

We started by retrieving the user defined constants. For a matter of clarity and readability we created local `const` variables to store and utilize these values. Subsequently, we initialize to zero all the terms of `system _matrix_poisson`, output of the method, and `density_matrix` object that will be used during the procedure. The first `for` loop is meant to generate the term accounting for the (old) ion density. The idea is to store inside an empty `SparseMatrix` the product between the Poisson mass matrix and the ion density vector `eta`. Actually, we implemented a lumped version, filling only the principal diagonal of the matrix; this choice is meant to reduce the computational cost without losing much accuracy.

```

// SYS_MAT = SYS_MAT +  eps*A
system_matrix_poisson.add(eps_r * eps_0, laplace_matrix_poisson);

// SYS_MAT = SYS_MAT + q0/V_TH * (eta)*MASS_MAT
system_matrix_poisson.add(q0 / V_TH, density_matrix);
```

The generation of the Newton matrix ends by performing the following two operations:

1. add, with a constant in front, the Laplace matrix, namely the matrix accounting for the grad-grad term
2. add the previously filled matrix with a constant

The method proceeds with the creation of the right-hand-side of the system:

```

// BUILDING SYSTEM RHS
poisson_rhs = 0;

PETScWrappers::MPI::Vector temp(locally_owned_dofs, mpi_communicator);
mass_matrix_poisson.vmult(temp,eta);
poisson_rhs.add(q0,temp);
laplace_matrix_poisson.vmult(temp,potential);
poisson_rhs.add(- eps_r * eps_0, temp);


poisson_rhs.compress(VectorOperation::insert);

}
```

First we set it to zero, then the building strategy starts. By exploiting a temporary vector, few matrix-vector multiplications and a couple of sum between vectors, we created the RHS. In particular, it contains contributions related to both the old ion density and the old potential. As always, the method ends with a call to `compress()`. In this way, each process sends updates for the entries in the matrix and vector that it does not own to the respective process that does own them. Once these updates are received from other processes, each process adds them to its local values. This procedure mirrors the integration of contributions from shape functions associated with multiple cells, much like in a serial computation. However, the distinction here is that the cells are distributed across different processes.

`assemble_drift_diffusion_matrix`

This is one of the most important methods of the class, it creates the drift-diffusion system following the procedure described in (3.1.2). Just like the other methods, this one also constructs matrix and right-hand-side looping over the owned cell.

```

template <int dim>
void CompleteProblem<dim>::assemble_drift_diffusion_matrix()
{

  //fix the constants
  const double kB = m_data.electrical_parameters.kB;
  const double q0 = m_data.electrical_parameters.q0;
  const double mu0 = m_data.electrical_parameters.mu0;
  const double T = m_data.electrical_parameters.stratosphere ? 217. : 303.;
  const double V_TH = kB * T / q0;
  const double D = mu0 * V_TH;

  const unsigned int vertices_per_cell = 4;
  std::vector<types::global_dof_index> local_dof_indices(vertices_per_cell);

  FullMatrix<double> cell_matrix(vertices_per_cell, vertices_per_cell);
```

```
  Vector<double>     cell_rhs(vertices_per_cell);

  const unsigned int t_size = 3;

  Vector<double> A_cell_rhs(t_size), B_cell_rhs(t_size);
  FullMatrix<double> A(t_size,t_size), B(t_size,t_size);

  std::vector<types::global_dof_index> A_local_dof_indices(t_size);
  std::vector<types::global_dof_index> B_local_dof_indices(t_size);
```

First of all, we retrieve the user defined values by initializing some constant variables. Then, we set up all the objects that will be used to store the local contribution of each cells. In particular, a `std::vector<types::global_dof_index>` is created to host the four DOFs of the current quadratic cell, instead the subsequent `FullMatrix` and `Vector` will hold the values for the local matrix and the local right-hand-side. The same objects are then introduced for a triangular cell, dividing them into two contributions A and B.

```

  evaluate_electric_field();

  QTrapezoid<dim-1> face_quadrature;

  const unsigned int n_q_points = face_quadrature.size();

  FEFaceValues<dim> face_values(fe, face_quadrature, update_values |
                        update_quadrature_points | update_JxW_values);
```

Before entering into the `for` loop, we computed the electric field using the method `evaluate_electric_field`. From now on, inside the vectors `Field_X` and `Field_Y` we have respectively the X component and the Y component of the electric field. This section of code ends with the introduction of objects that help to perform numerical integration.

```
for (const auto &cell : dof_handler.active_cell_iterators())
   {
       if (cell->is_locally_owned()){

           A = 0;
           B = 0;
           cell_matrix = 0;
           cell_rhs = 0;
           A_cell_rhs = 0;
           B_cell_rhs = 0;


           cell->get_dof_indices(local_dof_indices);
```

Finally, we start looping over the active owned cells of the current processor. As always, we start by setting to zero all the local contributions and by retrieving the degrees of freedom of the current cell.

```

// Lexicographic ordering
const Point<dim> v1 = cell->vertex(2); // top left
const Point<dim> v2 = cell->vertex(3); // top right
const Point<dim> v3 = cell->vertex(0); // bottom left
const Point<dim> v4 = cell->vertex(1); // bottom right

const double u1 = -potential[local_dof_indices[2]]/V_TH;
const double u2 = -potential[local_dof_indices[3]]/V_TH;
const double u3 = -potential[local_dof_indices[0]]/V_TH;
const double u4 = -potential[local_dof_indices[1]]/V_TH;

const double l_12 = side_length(v1,v2);
```

```
const double l_31 = side_length(v1,v3);
const double l_24 = side_length(v4,v2);
const double l_43 = side_length(v3,v4);

const double l_alpha = std::sqrt(l_12*l_12 +l_24*l_24 - 2*((v1 - v2) * (v4 - v2)));
const double l_beta = std::sqrt(l_43*l_43 + l_24*l_24 - 2*((v2 - v4) * (v3 - v4)));
```

From the current cell, following a lexicographical ordering, we retrieved the physical position of the four vertexes. After that, we stored in four different double the quantity $\frac{-\Phi_j}{\beta T_e}$, that will be used for the computation of the fluxes. Finally, we computed the distances between the nodes exploiting the helper function `side_length` and the two diagonals.

```

// velocities
Tensor<1,dim> u_f_1, u_f_2, u_f_3, u_f_4;
u_f_1[0] = Vel_X(local_dof_indices[2]);
u_f_1[1] = Vel_Y(local_dof_indices[2]);
u_f_2[0] = Vel_X(local_dof_indices[3]);
u_f_2[1] = Vel_Y(local_dof_indices[3]);
u_f_3[0] = Vel_X(local_dof_indices[0]);
u_f_3[1] = Vel_Y(local_dof_indices[0]);
u_f_4[0] = Vel_X(local_dof_indices[1]);
u_f_4[1] = Vel_Y(local_dof_indices[1]);

// versors
const Tensor<1,dim> dir_21 = (v1 - v2)/l_12;
const Tensor<1,dim> dir_42 = (v2 - v4)/l_24;
const Tensor<1,dim> dir_34 = (v4 - v3)/l_43;
const Tensor<1,dim> dir_13 = (v3 - v1)/l_31;


const double alpha21 = (u_f_2 * dir_21)/D*l_12 + (u1 - u2);
const double alpha42 = (u_f_4 * dir_42)/D*l_24 + (u2 - u4);
const double alpha34 = (u_f_3 * dir_34)/D*l_43 + (u4 - u3);
const double alpha13 = (u_f_1 * dir_13)/D*l_31 + (u3 - u1);
```

We proceed by retrieving the X and Y velocities in the four vertexes of the cell. Then, four versors `dir_ij` going from i to j are defined. This part of the code ends with the computation of the terms α_{ij} described in section (3.1.2).

```

// l_alpha is the longest diagonal: split by beta
if (l_alpha >= l_beta) {

        const double l_23 = side_length(v2,v3);
        const Tensor<1,dim> dir_23 = (v3 - v2)/l_beta;
        const double alpha23 = (u_f_2 * dir_23)/D*l_23 + (u3 - u2);

        // Triangle A:
        A = compute_triangle_matrix(v2,v1,v3, alpha21, alpha13, -alpha23, D);

        // Triangle B:
        B = compute_triangle_matrix(v3,v4,v2, alpha34, alpha42, alpha23, D);

        // Matrix assemble
        A_local_dof_indices[0] = local_dof_indices[3];
        A_local_dof_indices[1] = local_dof_indices[2];
        A_local_dof_indices[2] = local_dof_indices[0];

        B_local_dof_indices[0] = local_dof_indices[0];
        B_local_dof_indices[1] = local_dof_indices[1];
        B_local_dof_indices[2] = local_dof_indices[3];
```

Now we compute the fluxes for a triangular cell; the result depends on which is the larger diagonal term. First, we compute all the previously presented quantities for the new edge. Then, the local cell matrix is find through the

computation of two dense matrices. Both of them exploit the helper function `compute_triangular_matrix` which computes through the `bernoulli` function, the matrix presented in (3.1.2). The dense matrix A takes into account the terms presented as B_{ij}^{+}, instead B the terms B_{ij}^{-}. The `if` condition ends with the storing of the degrees of freedom related to the three vertexes involved in the two 3x3 matrices.

```
} else { // l_beta is the longest diagonal: split by alpha

    const double l_14 = side_length(v1,v4);
    const Tensor<1,dim> dir_14 = (v4 - v1)/l_alpha;
    const double alpha14 = (u_f_1 * dir_14)/D*l_14 + (u4 - u1);

    // Triangle A:
    A = compute_triangle_matrix(v4,v2,v1, alpha42, alpha21, alpha14, D);

    // Triangle B:
    B = compute_triangle_matrix(v1,v3,v4, alpha13, alpha34, -alpha14, D);

    A_local_dof_indices[0] = local_dof_indices[1];
    A_local_dof_indices[1] = local_dof_indices[3];
    A_local_dof_indices[2] = local_dof_indices[2];

    B_local_dof_indices[0] = local_dof_indices[2];
    B_local_dof_indices[1] = local_dof_indices[0];
    B_local_dof_indices[2] = local_dof_indices[1];
}
```

This section represents the `else` condition, namely what happens if the larger diagonal is the opposite one. The instructions are exactly the same of the previous part, it changes only the value related to the diagonal term, namely α_{14}.

```

for (unsigned int i = 0; i < t_size; ++i) {
    for (unsigned int j = 0; j < t_size; ++j) {
        A(i,j) = A(i,j)*timestep;
        B(i,j) = B(i,j)*timestep;
    }
}


ion_constraints.distribute_local_to_global(A, A_cell_rhs,  A_local_dof_indices,
    ion_system_matrix, ion_rhs);
ion_constraints.distribute_local_to_global(B, B_cell_rhs,  B_local_dof_indices,
    ion_system_matrix, ion_rhs);


}
}

ion_system_matrix.compress(VectorOperation::add);
ion_rhs.compress(VectorOperation::add);

}
```

Since the contribution of the flux in the final system is multiplied by the time step, we looped over all the elements of the local matrices to perform this adjustment. The method ends with the imposition of the boundary conditions and the distribution from local matrices and local RHS to the global ones. Note that the final matrix takes into account both matrices A and B.

`assemble_NS`

Since this assemble method follows the direction of the assembling procedure described in the `deal.II` tutorial n.57, here we report only the main difference with the original one: the effect that the electrical system has on the Navier-Stokes equations. In particular, from chapter three we know that the coupling resides in the right-hand-side of the system.

```
auto ion_cell = dof_handler.begin_active();  // DD dofs
const auto ion_endc = dof_handler.end();     // DD dofs

const unsigned int ion_dofs_per_cell = 4;
std::vector<types::global_dof_index> ion_local_dof_indices(ion_dofs_per_cell);

const double q0 = m_data.electrical_parameters.q0;
const double rho = m_data.electrical_parameters.stratosphere ? 0.089 : 1.225;

FEValues<dim> fe_values_ion(fe,
                            quadrature_formula_ion,
                            update_values | update_quadrature_points |
                                update_JxW_values);
```

We started by retrieving electrical iterators and constants. The challenge of this method was the fact that we needed to match (in parallel) the information of two different `DoFHandler` objects. The Navier-Stokes system, is described through `NS_dof_handler`, instead the electrical problem through `dof_handler`. A `FEValues<dim>` object was necessary to operate with electrical numerical vectors.

```
for (auto cell = NS_dof_handler.begin_active(), ion_cell = dof_handler.begin_active
    ();
   cell != NS_dof_handler.end() && ion_cell != dof_handler.end();
   ++cell, ++ion_cell)
{
  Assert(cell->index() == ion_cell->index(), ExcMessage("Mismatch between NS and
      ion cells!"));

  if (cell->is_locally_owned()) {

      fe_values.reinit(cell);
      fe_values_ion.reinit(ion_cell);
```

Like the other assemble methods we need to loop over the active owned cells. This time though, we need to satisfy a double condition, one for the navier-stokes cells, the other for the electrical cells.

```

std::vector<double> field_x_values(n_q_points);
std::vector<double> field_y_values(n_q_points);
std::vector<double> ion_density_values(n_q_points);

fe_values_ion.get_function_values(Field_X, field_x_values);
fe_values_ion.get_function_values(Field_Y, field_y_values);
fe_values_ion.get_function_values(ion_density, ion_density_values);

if (ion_cell->is_locally_owned()) {
ion_cell->get_dof_indices(ion_local_dof_indices);
}
```

Subsequently, we initialized vectors for both electric field (X and Y) and ion density. Moreover, we stored the number of the degrees of freedom of the current electrical cell.

```
for (unsigned int q = 0; q < n_q_points; ++q) {
```

```
2
3   Tensor<1, dim> f;
4   f[0] = q0 * field_x_values[q] / rho * ion_density_values[q];
5   f[1] = q0 * field_y_values[q] / rho * ion_density_values[q];
6
7   local_rhs(i) -=
8               (viscosity * scalar_product(current_velocity_gradients[q], grad_phi_u
                    [i]) -
9               current_velocity_divergences[q] * phi_p[i] -
10              current_pressure_values[q] * div_phi_u[i] +
11              gamma * current_velocity_divergences[q] * div_phi_u[i] +
12              current_velocity_gradients[q] * current_velocity_values[q] * phi_u[i]
                     -
13              scalar_product(phi_u[i], f)) * fe_values.JxW(q);
```

Finally, this is the extract of the code that describes the effect of the electrical phenomena in the fluid problem. With a `for` loop we cycled over all the quadrature points, and then we computed the electrohydrodynamic (EHD) force term in each of these nodes. The force is then added in the local right-hand-side of the fluid problem. As noted before, we skip the description of all the other terms since standard.

Solve methods

Once the systems are assembled, we need to solve them in order to retrieve a numerical solution. As before, we deal with different solve methods that will be explained in this subsection.

```
1
2
3  void solve_initial_poisson();
4  double solve_poisson();
5  void solve_nonlinear_poisson(const unsigned int max_iterations,const double tol);
6  void solve_drift_diffusion();
7  std::pair<unsigned int, double> solver_NS(bool use_nonzero_constraints, bool
       assemble_system, double time_step);
8  void solve_navier_stokes();
```

As in the case of the assemble methods, the functions above share a common strategy with different variations that make them unique. In the following, we will try to describe the general case and all the peculiarities of the methods.

`solve_initial_poisson`

This is the first solve method used in the code and also the easier one; it follows a very general strategy, thus, it is perfect to introduce this kind of function.

```
1  template <int dim>
2  void CompleteProblem<dim>::solve_initial_poisson()
3  {
4
5    PETScWrappers::MPI::Vector temp;
6    temp.reinit(locally_owned_dofs, mpi_communicator);
7
8    //Solve system problem
9    SolverControl sc_p(dof_handler.n_dofs(), 1e-10);
10   PETScWrappers::SparseDirectMUMPS solverMUMPS(sc_p);
11   solverMUMPS.solve(initial_matrix_poisson, temp, initial_poisson_rhs);
12
13   constraints_poisson.distribute(temp);
14   temp.compress(VectorOperation::insert);
15
16   //Initialize the potential with the solution of this first system
17   potential = temp ;
```

```
18
19 }
```

We start by creating a temporary vector, of the right size, that will be used to store the solution of the algebraic system. Then, the `SolverControl` object is introduced. This is a control class to determine convergence solvers, it requires only the number of the degrees of freedom and the desired tolerance as inputs. This class constitutes the argument for the real solver: `PETScWrappers::SparseDirectMUMPS`. In this particular case, the sparse direct solver `MUMPS` is employed, and a 1e-10 tolerance is set. Subsequently, the algebraic system is solved by exploiting the method `solve` of the solver. We stress the necessity of a temporary vector for this implementation: the auxiliary vector does not hold ghost values, unlike the majority of the class vectors, hence we can use it for storing the output of the previous function. The method ends with the application of the boundary conditions on the solution through the utility `.distribute()`, and the consequent initialization of the class vector potential. We now have a good starting point for the Newton method.

`solve_poisson`

This method is a bit more involved, and it starts by retrieving some user defined constants:

```
template <int dim>
double CompleteProblem<dim>::solve_poisson()
{

// Fixing costants
const double q0 = m_data.electrical_parameters.q0;
const double kB = m_data.electrical_parameters.kB;
const double T = m_data.electrical_parameters.stratosphere ? 217. : 303.;
const double V_TH = kB*T/q0;
```

By following a different procedure with respect to the last solve method, here we first apply boundary conditions on the whole system, and then we solve it. Since the boundary conditions change accordingly to the mesh geometry, we exploited the same procedure presented before based on a `unordered_map` and a `switch` case:

```
std::unordered_map<std::string, int> stringToCase{
  {"NACA", 1},
  {"WW", 2},
  {"CYL", 3}
  };

const std::string input = m_data.simulation_specification.mesh_TAG;
auto iter = stringToCase.find(input);

if (iter != stringToCase.end()) {
switch (iter->second) {

case 3:{

    std::map<types::global_dof_index, double> emitter_boundary_values,
                                              collector_boundary_values;

    VectorTools::interpolate_boundary_values(mapping, dof_handler,3, Functions::
        ZeroFunction<dim>(), emitter_boundary_values);
    MatrixTools::apply_boundary_values(emitter_boundary_values,
        system_matrix_poisson, poisson_newton_update, poisson_rhs);
```

```
20
21      VectorTools::interpolate_boundary_values(mapping, dof_handler,4, Functions::
            ZeroFunction<dim>(), collector_boundary_values);
22      MatrixTools::apply_boundary_values(collector_boundary_values,
            system_matrix_poisson, poisson_newton_update, poisson_rhs);
23
24      break;
25  }
```

We reported only a specific case, but the procedure is exactly the same for the other geometries: we define different `map` objects containing pairs made of DOFs and desired value. and then we impose these values on the system through `MatrixTools::apply_boundary_values`.

```
1
2   //Solve poisson system problem
3   const double coeff = 1e-3;
4   SolverControl sc_p(dof_handler.n_dofs(),coeff *poisson_rhs.l2_norm());
5   PETScWrappers::SparseDirectMUMPS solverMUMPS(sc_p);
6   solverMUMPS.solve(system_matrix_poisson, poisson_newton_update, poisson_rhs);
7
8   //Compute the residual
9   double residual = poisson_newton_update.linfty_norm();
```

At this point we follow the exact same procedure described in the first solve method, with the only difference that the tolerance change every time since it is based on the L^2-norm of the right-hand-side of the system. We then compute the output of the function: L^∞-norm of the newton update. However, the methods is not completed yet.

```
1
2   //Clamping
3   for (auto iter = locally_owned_dofs.begin(); iter != locally_owned_dofs.end(); ++
        iter){
4
5   if (poisson_newton_update[*iter] < -V_TH) { poisson_newton_update[*iter] = -V_TH; }
6   else if (poisson_newton_update[*iter] > V_TH) { poisson_newton_update[*iter] = V_TH
        ; }
7
8   // update ion density
9   eta[*iter] *= std::exp(-poisson_newton_update[*iter]/V_TH);
10  }
11
12  poisson_newton_update.compress(VectorOperation::insert);
13  eta.compress(VectorOperation::insert);
14
15  ion_constraints.distribute(eta);
16  eta.compress(VectorOperation::insert);
17
18  //Update current solution
19  PETScWrappers::MPI::Vector temp;
20  temp.reinit(locally_owned_dofs, mpi_communicator);
21
22  temp = potential;
23  temp.add(1.0, poisson_newton_update);
24
25  potential = temp;
26
27
28  return residual;
29
30  }
```

First we perform the clamping described in section (3.1.1) by spanning all the values stored inside the `poisson_newton_update` vector (`if-else` condition inside the `for` loop). Then, we update the ion density with the correct value of

the newton update. In conclusion, the new potential is computed by summing up the newton solution. The residual is finally returned.

`solve_nonlinear_poisson`

The goal of this method is to completely define the Newton algorithm. The input data are the usual two: maximum number of iterations allowed and the tolerance.

```
1
2  template <int dim>
3  void CompleteProblem<dim>::solve_nonlinear_poisson(const unsigned int
       max_iter_newton, const double toll_newton){
4
5  unsigned int counter = 0; // it keeps track of newton iteration
6
7  double increment_norm = std::numeric_limits<double>::max();
8
9  while(counter < max_iter_newton && increment_norm > toll_newton){
10
11 //NB: Mass and Laplace matrices are already build
12
13 assemble_nonlinear_poisson();
14
15 increment_norm = solve_poisson();  //residual computation,
16                                    //clamping on newton update,
17                                    //BCs and update of the charges
18
19 counter ++;
20
21 if(counter == max_iter_newton){
22   pcout<< "   MAX NUMBER OF NEWTON ITERATIONS REACHED!"<<std::endl;
23 }
24
25 }
26
27 }
```

The method starts by initializing the counter and the error. Subsequently, a `while` loop considering the current iteration and the current error starts. Inside this loop the Newton system is both created and solved until the conditions are satisfied. Note that only `assemble_nonlinear_poisson()` is called, neither mass and Lapalce matrices are being reconstructed since they do not vary! They are both filled in the `run()` method.

`solve_drift_diffusion`

For the sake of completeness we also report the method which solve the drift-diffusion problem, even if the structure does not introduce any peculiarity.

```
1
2  template <int dim>
3  void CompleteProblem<dim>::solve_drift_diffusion()
4  {
5
6  const double coeff = 1e-3;
7  PETScWrappers::MPI::Vector temp(locally_owned_dofs, mpi_communicator);
8  SolverControl sc_ion(dof_handler.n_dofs(), coeff *ion_rhs.l2_norm());
9  PETScWrappers::SparseDirectMUMPS solverMUMPS_ion(sc_ion);
10 solverMUMPS_ion.solve(ion_system_matrix, temp, ion_rhs);
11
12 ion_constraints.distribute(temp);
13 temp.compress(VectorOperation::insert);
14
15 ion_density = temp;
```

```
16
17 }
```

`solver_NS`

The following method describe the setup for the solver of the navier-stokes system of equations. Since the matrix of the fluid problem is very ill conditioned, we need to employ a preconditioner in order to converge in a acceptable time.

```
1
2  template <int dim>
3  std::pair<unsigned int, double>
4  CompleteProblem<dim>::solver_NS(bool use_nonzero_constraints, bool assemble_system,
       double time_step)
5  {
6
7  if (assemble_system)
8  {
9    preconditioner.reset(new BlockSchurPreconditioner(timer,
10                                                     gamma,
11                                                     viscosity,
12                                                     timestep_NS,
13                                                     owned_partitioning,
14                                                     NS_system_matrix,
15                                                     NS_mass_matrix,
16                                                     pressure_mass_matrix));
17 }
18
19 // to avoid to have a tolerance too small
20 double coeff = 0.0 ;
21 if (time_step < 2) {
22     coeff = 1e-2;
23 } else {
24     coeff = 1e-4;
25 }
```

If the boolean `assemble_system` is true, the preconditioner is created through an instance of the class `BlockSchurPreconditioner`. In the opposite case, the function move on towards the definition of the coefficient that establish the tolerance of the solver.

```
1
2  SolverControl solver_control(10*NS_system_matrix.m(), coeff * NS_system_rhs.l2_norm
       (), true);
3
4  SolverBicgstab<PETScWrappers::MPI::BlockVector> bicg(solver_control);
5
6  bicg.solve(NS_system_matrix, NS_solution_update, NS_system_rhs, *preconditioner);
7
8  const AffineConstraints<double> &constraints_used =
9  use_nonzero_constraints ? nonzero_NS_constraints : zero_NS_constraints;
10
11 constraints_used.distribute(NS_solution_update);
12
13 return {solver_control.last_step(), solver_control.last_value()};
14 }
```

Like every other `solve` method, an instance of a `SolverControl` is created to control the solver parameters. Finally, the system is solved and the solution stored inside the NS solution update vector. For the fluid problem we decided to employ the stabilized iterative bicg solver instead of a direct mumps. Then, based on the input, some boundary condition are imposed and a pair of values returned.

solve_navier_stokes

This method combines the assemble of the Navier-Stokes system with the solver which actually solves it. Moreover, it manages the updates of the fluid quantities.

```
template <int dim>
void CompleteProblem<dim>::solve_navier_stokes()
{

  evaluate_electric_field();

  NS_solution_update = 0;

  bool apply_nonzero_constraints = (time_NS == 1);
  bool assemble_system = true;

  pcout << "   ASSEMBLE NAVIER STOKES SYSTEM ..." ;
  assemble_NS(apply_nonzero_constraints, assemble_system);
  pcout << "   Done !"<<std::endl;

  pcout << "   SOLVE NAVIER STOKES SYSTEM ..."  << std::endl;
  auto state = solver_NS(apply_nonzero_constraints, assemble_system, time_NS);


  PETScWrappers::MPI::BlockVector tmp;
  tmp.reinit(owned_partitioning, mpi_communicator);
  tmp = NS_solution;
  tmp += NS_solution_update;
  NS_solution = tmp;
```

After calculating the electric filed, we set to zero the update of the current solution. Then, we call the assemble and the solver of the fluid problem. Both of them are characterize by the boolean introduced few lines above. Finally, the solution is updated.

```
PETScWrappers::MPI::Vector temp_X;
PETScWrappers::MPI::Vector temp_Y;
PETScWrappers::MPI::Vector temp_pressure;

temp_X.reinit(locally_owned_dofs,  mpi_communicator);
temp_Y.reinit(locally_owned_dofs,  mpi_communicator);
temp_pressure.reinit(owned_partitioning[1], mpi_communicator);

const unsigned int dofs_per_cell = fe.n_dofs_per_cell();
std::vector<types::global_dof_index> ion_local_dof_indices(dofs_per_cell);

const unsigned int dofs_per_NS_cell = NS_fe.n_dofs_per_cell();
std::vector<types::global_dof_index> NS_local_dof_indices(dofs_per_NS_cell);

auto cell = dof_handler.begin_active();
auto NS_cell = NS_dof_handler.begin_active();

const auto endc = dof_handler.end();
const auto NS_endc = NS_dof_handler.end();
```

Since the physical problem we want to solve is coupled, we needed to find a way to pass information between the two sub-problems. That is why the second part of the method has the purpose to bridge the solutions who live on the `DoFHandler` of the fluid problem with the ones living on the electric `DoFHandler`. In particular, we want to split the navier-stokes solution into three different vectors defined over the electrical degrees of freedom: `temp_X`, `temp_Y` and `temp_pressure`. We defined the usual iterators that will be use to loop over all the active owned cells.

```

std::vector<int> index_x = {0, 3, 6, 9};

```

```
std::vector<unsigned int> block_component(dim + 1, 0);
block_component[dim] = 1;
DoFRenumbering::component_wise(NS_dof_handler, block_component);
dofs_per_block = DoFTools::count_dofs_per_fe_block(NS_dof_handler, block_component);
unsigned int dof_u = dofs_per_block[0];

for (auto NS_cell = NS_dof_handler.begin_active(), ion_cell = dof_handler.begin_active();
NS_cell != NS_dof_handler.end() && ion_cell != dof_handler.end();
++NS_cell, ++ion_cell)
{

  Assert(NS_cell->index() == ion_cell->index(), ExcMessage("Mismatch between NS and ion cells!"));

  if (NS_cell->is_locally_owned() && ion_cell->is_locally_owned()) {

      NS_cell->get_dof_indices(NS_local_dof_indices);
      ion_cell->get_dof_indices(ion_local_dof_indices);
```

Before entering in the loop we defined two vectors: `index_x` that contains the local DOF of the X component of the velocity and `block_component`, which will be used to distinguish velocity from pressure solution. Once inside, as always, we retrieve the degrees of freedom of the current cell from both the `DoFHandler`.

```

for (unsigned int i = 0; i < 4; ++i) {

  // vel x index
  const unsigned int dof_x = NS_local_dof_indices[index_x[i]];

  // vel y index
  const unsigned int dof_y = NS_local_dof_indices[index_x[i] + 1];

  // pressure index
  const unsigned int dof_p = NS_local_dof_indices[3 * i + 2];

  if (dof_x < NS_solution.block(0).size() && dof_y < NS_solution.block(0).size() &&
      dof_p < NS_solution.block(1).size() + dof_u) {

      temp_X(ion_local_dof_indices[i]) = NS_solution.block(0)[dof_x];
      temp_Y(ion_local_dof_indices[i]) = NS_solution.block(0)[dof_y];
      temp_pressure(ion_local_dof_indices[i]) = NS_solution.block(1)[dof_p-dof_u];
  } else {

      std::cerr << "Errore: Indice DoF fuori dai limiti. DoF X: " << dof_x << ", DoF Y: " << dof_y << ", DoF P: " << dof_p << std::endl;
  }
}

}
}

  temp_X.compress(VectorOperation::insert);
  temp_Y.compress(VectorOperation::insert);
  temp_pressure.compress(VectorOperation::insert);

  Vel_X = temp_X;
  Vel_Y = temp_Y;
  pressure = temp_pressure;

}
```

We cycle over the vertexes of the electric cell. Inside the `for` loop we retrieve the degrees of freedom of the fluid variables that live on the current vertex. Finally, inside the `if` condition we associate the new vectors with these values. The method ends with the usual `compress` and the passage of information from the

temporary vectors to the ones that hold also the ghost cells.

evaluate_electric_field

The name of this method is pretty explanatory, it computes the electric field starting from the current potential vector. It divide the contributions into two vectors, one for each physical dimension. It also initialize a third vector that tracks the contribution of each degrees of freedom.

```
template <int dim>
void CompleteProblem<dim>::evaluate_electric_field()
{

  const unsigned int dofs_per_cell = fe.n_dofs_per_cell();

  PETScWrappers::MPI::Vector global_dof_hits(locally_owned_dofs, mpi_communicator);

  PETScWrappers::MPI::Vector el_field_X(locally_owned_dofs, mpi_communicator);
  PETScWrappers::MPI::Vector el_field_Y(locally_owned_dofs, mpi_communicator);

  QTrapezoid<dim-1> iv_quadrature;
  FEInterfaceValues<dim> fe_iv(fe, iv_quadrature, update_gradients);

  const unsigned int n_q_points = iv_quadrature.size();
  std::vector<Tensor<1,dim>> iv_gradients(n_q_points);

  std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
```

We exploit `QTrapezoid<dim-1>` and `FEInterfaceValues<dim>` to perform numerical integration on the owned cells, moreover, we initialize a tensor which will take into account the gradient of the potential on the DOFs.

```

// Loop on active cells
for (auto &cell : dof_handler.active_cell_iterators())
{
if (cell->is_locally_owned())
{
  for (const auto face_index : GeometryInfo<dim>::face_indices())
  {
    fe_iv.reinit(cell, face_index);
    local_dof_indices = fe_iv.get_interface_dof_indices();
    fe_iv.get_average_of_function_gradients(potential, iv_gradients);

    for (const auto q : fe_iv.quadrature_point_indices())
    {
      for (const auto i : fe_iv.dof_indices())
      {
        // Incrementiamo global_dof_hits per tenere traccia del numero di
            contributi
        global_dof_hits[local_dof_indices[i]] += 1.0;
```

We loop over the active owned cells; for each of them we store their local degrees of freedom and gradient contribution. Then we loop over each quadrature node and DOF.

```
            for (unsigned int d = 0; d < dim; ++d)
            {
              if (d == 0)
                el_field_X(local_dof_indices[i]) += -iv_gradients[q][d];
              else if (d == 1)
                el_field_Y(local_dof_indices[i]) += -iv_gradients[q][d];
              else
                Assert(false, ExcNotImplemented());
            }
          }
        }
```

```
      }
    }
  }
```

Finally, the electric field is computed in both direction.

```

  el_field_X.compress(VectorOperation::add);
  el_field_Y.compress(VectorOperation::add);
  global_dof_hits.compress(VectorOperation::add);


  el_field_X.update_ghost_values();
  el_field_Y.update_ghost_values();
  global_dof_hits.update_ghost_values();


  for (auto iter = locally_owned_dofs.begin(); iter != locally_owned_dofs.end(); ++
      iter)
  {

    const double hit_count = global_dof_hits[*iter];

    el_field_X[*iter] /= std::max(1.0, hit_count);
    el_field_Y[*iter] /= std::max(1.0, hit_count);
  }

  el_field_X.compress(VectorOperation::insert);
  el_field_Y.compress(VectorOperation::insert);


  Field_X = el_field_X;
  Field_Y = el_field_Y;
}
```

The method ends with the compression and the normalization of the new vectors. As the reader can see, through the function we exploited two temporary vector which store the gradient of the potential `el_fiel_X` and `el_field_Y`. We are forced to define this two auxiliary vector since a ghost vector, namely a vector which store both owned and ghost cells like `Field_X` and `Field_Y`, can not accede to its own values by element. We needed to operate element-wise in order to compute the electric field, thus we introduced these vectors.

perform_drift_diffusion_fixed_point_iteration_step

This method creates and solve the algebraic system related to the drift-diffusion equation. First of all it sets to zero both the system matrix and the right-hand-side, then it starts generating all the terms. Notice that when the method `assemble_drift_diffusion_matrix` is called, it builds starting from the ion system matrix, hence the contribution of the mass matrix is not lost.

```
template <int dim>
void CompleteProblem<dim>::perform_drift_diffusion_fixed_point_iteration_step()
{
  //(MASS  +  DELTA_T * DD_MATRIX)*ION_DENS = MASS*OLD_ION_DENSITY

  PETScWrappers::MPI::Vector temp;
  temp.reinit(locally_owned_dofs, mpi_communicator);

  ion_rhs = 0;
  ion_system_matrix = 0;

  // generate RHS: MASS_ION * OLD_DENSITY
  ion_mass_matrix.vmult(temp, old_ion_density);
  ion_rhs.add(1.0,temp);
```

```
15
16    // generate LHS: (MASS + DELTA_T * DD_MATRIX)
17    ion_system_matrix.add(1.0, ion_mass_matrix);
18    assemble_drift_diffusion_matrix();
19
20    solve_drift_diffusion();
21
22  }
```

update_ion_boundary_condition

The algorithm that solves the drift-diffusion system could present numerical oscillations, hence, we need to stabilize it. The strategy that we implemented through this method is extensively described in [2]. The idea is to relax the value imposed on the boundaries of both emitter and collector, with a convex combination of the exponential diode condition and the value of the ion density computed in last iteration. This combination is tuned by a parameter θ that is dynamically computed through the simulation.

```
1  template <int dim>
2  void CompleteProblem<dim>::update_ion_boundary_condition(){
3
4  double theta;
5  const double k_min = 0.995;
6  const double k_max = 1.005;
7
8  const double ion_norm = ion_density.l2_norm();
9  const double old_ion_norm = old_ion_density.l2_norm();
10 const double condition = ion_norm / old_ion_norm;
11
12 if(condition <= k_max && condition >= k_min){
13     theta = 1;
14 }
15
16 if(condition > k_max){
17    theta = (k_max -1)/(condition -1);
18 }
19
20 if(condition < k_min){
21    theta = (k_min -1)/(condition -1);
22 }
23
24 pcout<<" theta is: "<<theta<<std::endl;
```

First of all, the method calculates theta by comparing the L^2 norm of the current ion density with the old ion density, adjusting theta if this ratio deviates from a specified range (0.995 to 1.005). The range (k_{min}, k_{max}) was tuned by hand performing different simulations.

```
1
2  PETScWrappers::MPI::Vector temp;
3  temp.reinit(locally_owned_dofs, locally_relevant_dofs, mpi_communicator);
4  temp = old_ion_density;
5
6  // Corona inception condition: boundary condition for ion density
7  Functions::FEFieldFunction<dim, dealii::PETScWrappers::MPI::Vector>
       solution_as_function_object_1(dof_handler, ion_density, mapping);
8  Functions::FEFieldFunction<dim, dealii::PETScWrappers::MPI::Vector>
       solution_as_function_object_2(dof_handler, temp, mapping);
9
10
11 auto boundary_evaluator = [&] (const Point<dim> &p)
12 {
13
14 const double ion_value = solution_as_function_object_1.value(p);
```

```
const double old_ion_value = solution_as_function_object_2.value(p);

const double value = ion_value;

return value;

};


std::unordered_map<std::string, int> stringToCase{
  {"NACA", 1},
  {"WW", 2},
  {"CYL", 3}
  };

const std::string input = m_data.simulation_specification.mesh_TAG;
auto iter = stringToCase.find(input);
```

Using the `boundary_evaluator` lambda function, which retrieves ion density values at boundary points, the function applies boundary conditions based on the simulation's mesh type (NACA, WW, or CYL).

```

ion_constraints.close();

DynamicSparsityPattern ion_dsp(locally_relevant_dofs);
DoFTools::make_sparsity_pattern(dof_handler, ion_dsp, ion_constraints, false);

SparsityTools::distribute_sparsity_pattern(ion_dsp,
                                           dof_handler.locally_owned_dofs(),
                                           mpi_communicator,
                                           locally_relevant_dofs);

ion_mass_matrix.clear();
ion_mass_matrix.reinit(locally_owned_dofs, locally_owned_dofs, ion_dsp,
    mpi_communicator);

ion_system_matrix.clear();
ion_system_matrix.reinit(locally_owned_dofs, locally_owned_dofs, ion_dsp,
    mpi_communicator);
}
```

Finally, the updated boundary conditions are applied following the usual rule.

output_results

The goal of this method is to generate .pvtu and .vtu files that represent the results of the simulations.

```

template <int dim>
void CompleteProblem<dim>::output_results(const unsigned int cycle)
{

// Base directory for output
std::string base_directory = "../output";

// Directory to store the results of this simulation
std::string output_directory = base_directory + "/NS_DD_Simulation/";

// Ensure the output directory is created (if it doesn't exist)
if (!std::filesystem::exists(output_directory))
{
    std::filesystem::create_directory(output_directory);
}

DataOut<dim> data_out;

data_out.attach_dof_handler(dof_handler);
```

```
21
22 data_out.add_data_vector(potential, "potential");
23 data_out.add_data_vector(pressure, "pressure");
24 data_out.add_data_vector(Vel_X, "Vel_X");
25 data_out.add_data_vector(Vel_Y, "Vel_Y");
26 data_out.add_data_vector(ion_density, "Ion_Density");
27 data_out.add_data_vector(Field_X, "Field_X");
28 data_out.add_data_vector(Field_Y, "Field_Y");
```

This method is invoked for every cycle of the complete algorithm, that is why we pass the integer "cycle" as input. The function is fairly easy; it creates the output directory and then it groups all the results inside an instance of `DataOut<dim>`.

```
1
2 Vector<float> subdomain(triangulation.n_active_cells());
3 for (unsigned int i = 0; i < subdomain.size(); ++i)
4     subdomain(i) = triangulation.locally_owned_subdomain();
5
6 data_out.add_data_vector(subdomain, "subdomain");
7 data_out.build_patches();
8
9 data_out.write_vtu_with_pvtu_record(output_directory, "solution", cycle,
      mpi_communicator, 2, 1);
10
11
12 }
```

It also shows the different subsets of the domain that each processor takes into account.

run

Second, and last, public method of the class; through this method the user can start the numerical simulation. The method follows exactly the steps of the algorithm described in section (3.3).

```
1
2 template <int dim>
3 void CompleteProblem<dim>::run()
4 {
5   // fix the constants
6   const double N_0 = m_data.electrical_parameters.stratosphere ? 2.2e-3 : 0.5e-3;
7
8   pcout << "   SETUP POISSON PROBLEM ... ";
9   setup_poisson();
10  pcout << "   Done !"<<std::endl;
11
12  pcout << "   ASSEMBLE INITIAL SYSTEM ... ";
13  assemble_initial_system();
14  pcout << "   Done !"<<std::endl;
15
16  pcout << "   SOLVE INITIAL SYSTEM (INITIALIZE POTENTIAL) ... ";
17  solve_homogeneous_poisson();
18  pcout << "   Done !"<<std::endl;
19
20  pcout << "   ASSEMBLE MASS AND LAPLACE POISSON MATRICES ... ";
21  assemble_poisson_laplace_matrix();
22  assemble_poisson_mass_matrix();
23  pcout << "   Done !"<<std::endl;
24
25  pcout << "   SETUP DRIFT-DIFFUSION PROBLEM ... ";
26  setup_drift_diffusion();
27  pcout << "   Done !"<<std::endl;
28
29  pcout << "   ASSEMBLE MASS DRIFT DIFFUSION MATRIX ... ";
30  assemble_drift_diffusion_mass_matrix();
31  pcout << "   Done !"<<std::endl;
32
```

```
  pcout << "   INITIALIZE OLD_ION_DENSITY ... ";
  VectorTools::interpolate(mapping, dof_handler , Functions::ConstantFunction<dim>(
      N_0), old_ion_density);
  pcout << "   Done !"<<std::endl;

  pcout << "   SETUP NAVIER STOKES PROBLEM ... "<< std::endl;
  setup_NS();

  pcout << "   STORE INITIAL CONDITIONS ... ";
  output_results(0);
  pcout << "   Done !"<<std::endl<<std::endl;
```

The method starts by calling all the setups and by assembling all the matrices which will no longer change afterwards, it also compute the first potential and creates the output of the initial condition.

```

  // SET ERRORS AND TOLERANCES
  int step_number = 0;                    // time algorithm
  const unsigned int max_it = 1e+3;       // DD fix point algorithm
  const unsigned int max_steps = 1100;    // time algorithm
  const double time_tol = 1.0e-3;         // time algorithm
  const double tol = 1.e-9;              // tol poisson newton
  double time_err = 1. + time_tol;        // time algorithm error

  // INITIALIZE ETA
  eta = old_ion_density; // basically eta = N_0 function
```

Subsequently, we set tolerances, errors and final time instant for the algorithms that will be solved.

```

// START THE ALGORITHM
pcout << "   START COMPLETE PROBLEM ... "<< std::endl;

while (step_number < max_steps){

  ++step_number;

  // adjust time stepping
  if (step_number == 200 && timestep < 1.e-1){
  timestep*= 10.;
  timestep_NS*=10.;
  }

  int it = 0;                          // internal gummel iterator
  const double gummel_tol = 6.e-4;     // tol gummel algorithm
  double err = gummel_tol + 1.;        // error gummel algorithm

  PETScWrappers::MPI::Vector previous_density;
  previous_density.reinit(locally_owned_dofs, mpi_communicator);
```

Finally, the time dependent algorithm starts, it will stop when a certain time instant will be reached. By performing various simulations we realized that at a certain point the solution reaches a stationary solution, so we increase the time step.

```

pcout << "   GUMMEL ALGORITHM N. "<<step_number;

while (err > gummel_tol && it < max_it) {

solve_nonlinear_poisson(max_it,tol); // UPDATE potential AND eta (loop over k)

previous_density = ion_density; // save the previously computed ion_density

perform_drift_diffusion_fixed_point_iteration_step(); // UPDATE ion_density

```

```
previous_density -= ion_density;

err = previous_density.linfty_norm()/ion_density.linfty_norm(); // compute error

eta = ion_density;

it++;

}


if (it >= max_it){
pcout << "WARNING! DD achieved a relative error " << err << " after " << it << "
    iterations" << std::endl;
}
```

Once entered in the time loop we solve the electric problem in a nested while loop.

```

  pcout << "   UPDATE ION BOUNDARY CONDITIONS ... ";
  update_ion_boundary_condition();
  assemble_drift_diffusion_mass_matrix();
  pcout<<"   Done !"<<std::endl;

  previous_density = old_ion_density;
  previous_density -= ion_density;

  time_err = previous_density.linfty_norm()/old_ion_density.linfty_norm();
  pcout << "   Density change from previous time-step is: " << time_err*100. << " %
      " << std::endl;
  pcout << "   time error is:  "<<time_err << std::endl<<std::endl;

  old_ion_density = ion_density;


  if (step_number % 40 == 1){ // NS solution update every 40 timesteps
    time_NS += 1;
    pcout << "   START NAVIER STOKES PROBLEM ... "<< std::endl;
    solve_navier_stokes();

  }


  output_results(step_number);

 }
}
```

After that, the boundary conditions and the ion density are updated. Subsequently, the time error is computed and the fluid problem solved (if and only if the step number is correct). Finally, the numerical results are stored in .vtu and .pvtu files. This conclude the description of the main class of our code.

Helper functions

Along with the declaration and definition of the class methods presented above, we exploited few helper functions to simply the implementation. We just describe them since they are very intuitive:

bernoulli

```
void bernoulli (double x, double &bp, double &bn)
```

Given an input x it computes the values of the bernoulli functions b^+ and b^-; these values are used for the computation of current densities.

side_length

```
double side_length (const Point<2> a, const Point<2> b)
```

Compute the distance of the two input points.

triangle_denom

```
double triangle_denom(const Point<2> a, const Point<2> b, const Point<2> c)
```

It uses the coordinates of the input points to compute a value based on the determinant of a matrix that represents the triangle's vertices. It calculates the denominator of an area-related expression for a triangle.

face_normal

```
Tensor<1,2> face_normal(const Point<2> a, const Point<2> b)
```

It calculates the normal vector for a line segment defined by two point.

compute_triangle_matrix

```
FullMatrix<double> compute_triangle_matrix(const Point<2> a,
                                           const Point<2> b,
                                           const Point<2> c,
                                           const double alpha12,
                                           const double alpha23,
                                           const double alpha31,
                                           const double D)
```

It computes the contribution for the drift diffusion flux matrix of a triangular element with diffusivity equal to the input data D. This function is employed for the assemble method of the drift diffusion flux matrix.

get_emitter_normal

```
Tensor<1,2> get_emitter_normal(const Point<2> &a,
                               const Point<2> &emitter_center)
```

It computes the normal to the surface of the emitter centered in `emitter_center` in a specified point a.

Numerical Results

This chapter is subdivided into two main sections, the first one discuss the tests that we performed to validate our algorithms, the second one present the results of a complete simulation.

6.1 Validation

We decided to validate separately electrical and fluid dynamic codes. The former is tested against the provided sequential code, the latter with literature data.

6.1.1 Pn-junction

The validity of the solver for the electrical part of the problem is tested considering a semiconductor problem, with a well-known solution. The problem consists of solving for the potential, charge, and hole densities in a pn junction. The domain is a simple rectangular junction of arbitrary length L. The first 45% of the length has positive doping D, and the final 45% has negative doping A. The mobilities are $\mu_p = 0.1\,[\mathrm{m^2 s^{-1} V^{-1}}]$ for positive charges and $\mu_n = 0.03\,[\mathrm{m^2 s^{-1} V^{-1}}]$ for holes. The temperature in volts is $V_E = 0.026\,[\mathrm{V}]$ for both. On the left boundary, the values

$$n_1 = \frac{D}{2} + \frac{\sqrt{D^2 + 4n_0^2}}{2}$$

for the holes and

$$p_1 = \frac{n_0^2}{p_2}$$

for the positive charges. On the right boundary, the values

$$n_2 = n_0^2 n_1$$

for the holes,

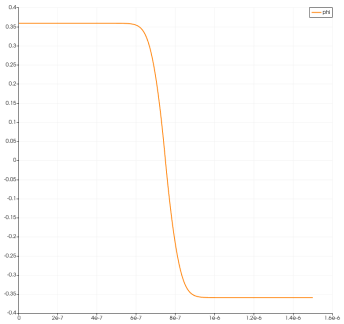
$$\phi_2 = V_E \log\left(\frac{n_2}{n_0}\right)$$

for the potential, and

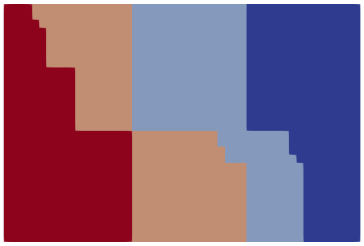
$$p_2 = \frac{A}{2} + \frac{\sqrt{A^2 + 4n_0^2}}{2}$$

for the positive charges. The constants A, D, n_0 are set to $A = D = 10^{22}$ and $n_0 = 10^{16}$. The problem is solved using the Gummel map presented before, hence solving the Non-linear Poisson problem followed by Drift Diffusion for as many fixed point iterations as necessary to achieve a relative error between the infinity norm of the densities lower than the tolerance 10^{-10}.

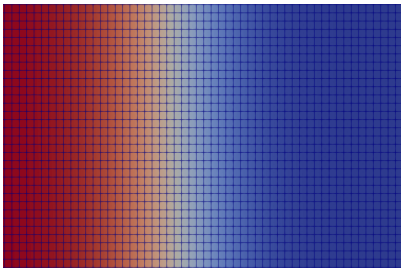
(a) Potential



(b) Plot Potential



(c) subdomains



(d) grid zoom

Figure 6.1: pn-junction results

6.1.2 Time dependent Navier-Stokes

The solver validation for the time dependent Navier-Stokes system is very straightforward. As we mentioned in chapter 2, we took a good part of the implementation from a `deal.II` tutorial; then, we decided to change the treatment of the convective term, moving from a Newton's problem to a sequence of Oseen's problems. Hence, to validate our solver we simply compared our solution with the one from the tutorial. We decided to use both NACA and wire-wire geometries as computational domain. We imposed an incremental inlet velocity ranging from 0.1 to 1 in 100 time steps. The Figures 6.2 and 6.3 are referrend to an inlet velocity of 0.6. The other parameters are hard-coded in `TimeDependentNS/InsIMEX_impl`.

The aim of this validation was to test the code with various geometries within the range of Reynolds numbers relevant to real tests. We were not focused on obtaining exact numerical results, as these were not useful for the project's development; rather, we aimed to qualitatively assess whether our code functions correctly. This was made easier by starting with a well-tested framework, as demonstrated in the deal.II tutorial. We obtained smooth and coherent solutions, allowing us to proceed with the development of the complete problem.

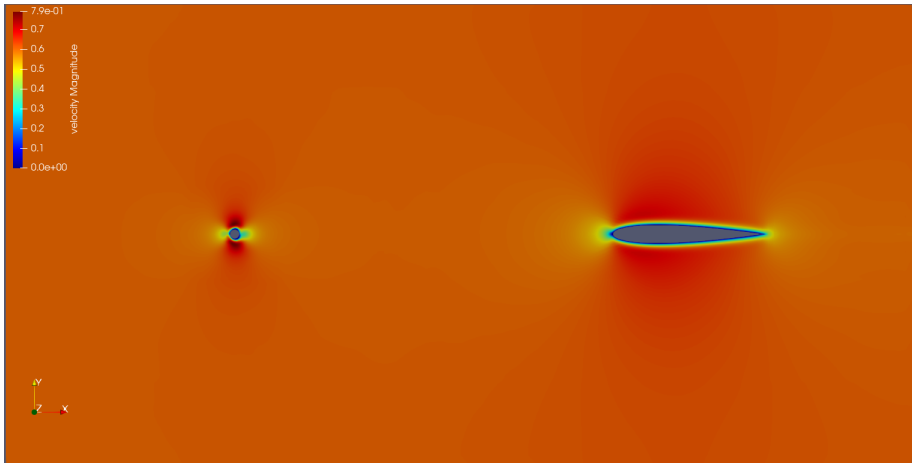


Figure 6.2: Velocity magnitude, NACA geometry

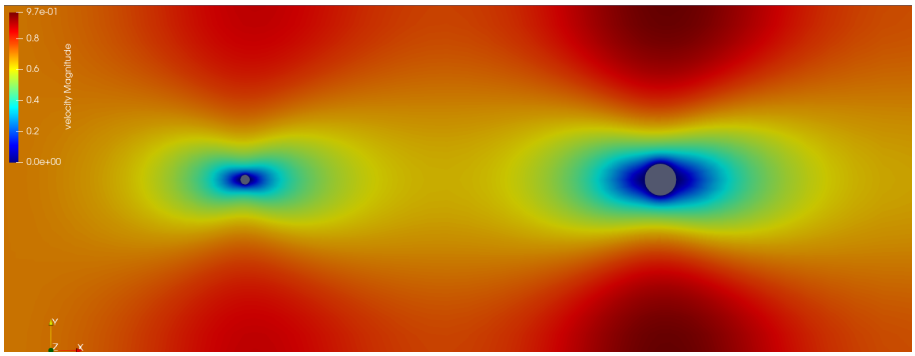


Figure 6.3: Velocity magnitude, wire-wire geometry

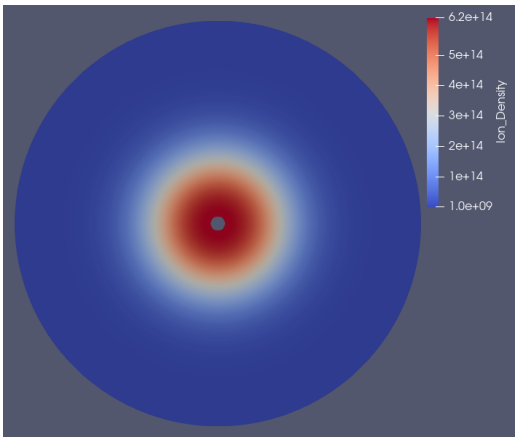
6.2 Coupled Problem Simulation

As we mentioned before, we present our coupled code on the geometry characterized by two concentric circles (tag CYL). The inner circle represents the emitter, while the outer one the collector. The computational domain in the region of the 2D space in between the electrodes. The boundary and initial conditions implemented are the one presented in both chapter 2 and 3, we report here the values we employed for the numerical simulation:

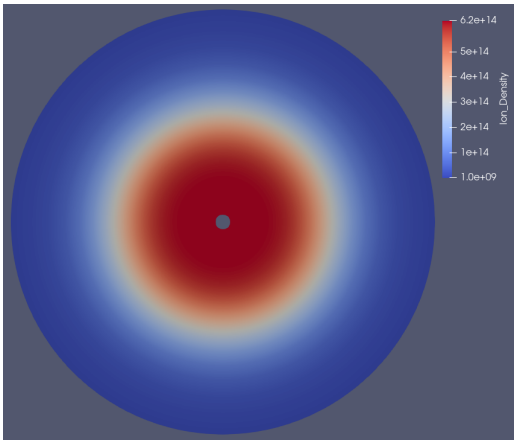
Category	**Parameter**	**Value**
Fluid Parameters	viscosity	1.8e-5
	gamma	1.0
Electrical Parameters	eps_0	8.854e-12
	eps_r	1.0006
	q0	1.602e-19
	kB	1.381e-23
	mu0	2e-4
	stratosphere	false
	E_0N	1.114e+6
	E_ref	4.1436e+5
	N_ref	1e+14
	N_min	1.0e+9
	Mm	29.0e-3
	Vv	2e-4
Geometrical Parameters	emitter_center_X	0.0
	emitter_center_Y	0.0
	emitter_radius	0.7e-3
	collector_radius	2e-2
	distance_emitter_collector	2.5
	distance_trailing_edge_outlet	1.0
	distance_emitter_inlet	1.5
	distance_emitter_up_bottom	1.5
Simulation Specification	mesh_name	concentric_cylinders.msh
	mesh_TAG	CYL

Table 6.1: Simulation Parameters

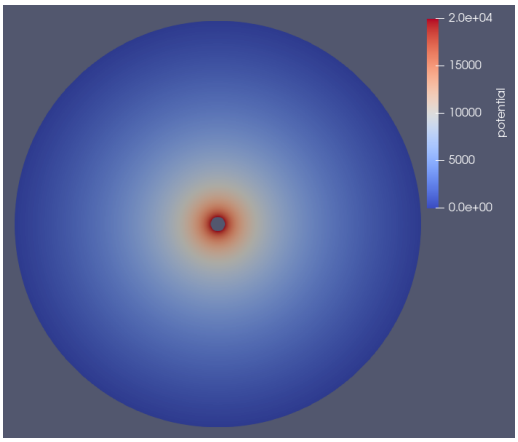
We share some plots of the obtained solution:



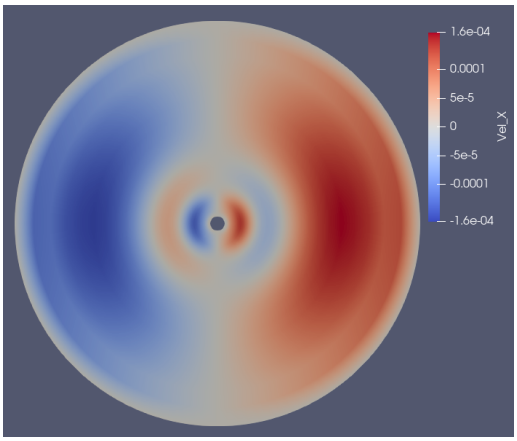
(a) Ion density at second timestep



(b) Ion density at 5^{th} timestep



(c) Potential

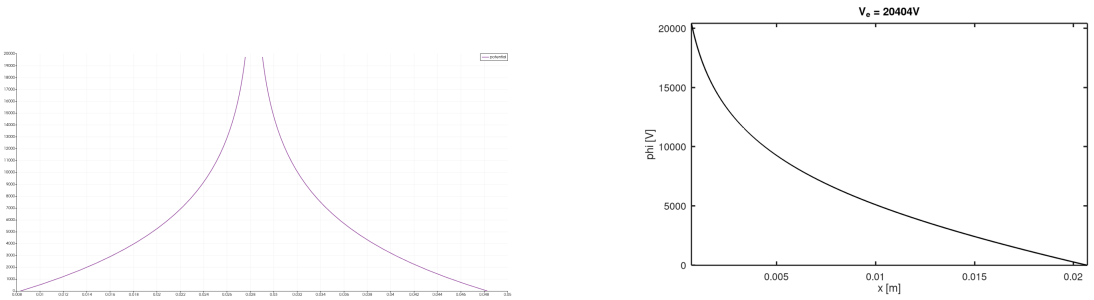


(d) X component of the velocity at 12^{th} timestep

Figure 6.4: Comparison of on density, potential, and velocity components across different timesteps

The solution follows the desired physical behavior: thanks to the difference of potential the corona discharge effect is triggered and hence the drift of positive ions starts. Since the charges drift towards the collector they move the surrounding air particles. Thus, the fluid (air) trapped inside the two electrodes start moving. As the reader can see, the magnitude of the air remains very low, this is perfectly in line with the experimental data. The behavior of the velocity could seems un-physical, but this is actually related to the very low magnitude. We think that this low-velocity phenomena is also the reason why our coupled algorithm works; we find out that higher velocities bring numerical instabilities in the implemented code, hence the algorithm does not converge. We think that the problem is related to the numerical method that solve the Navier-Stokes system.

By comparison with a validated 1D `matlab` version of the code, (actually the 1D version solves only the electrical problem with a steady drift-diffusion system) we find out that our potential presents the correct behavior:



(a) Our potential profile

(b) Validated Potential profile

Figure 6.5: Comparison of potential profile between our case and a validated one

To the best of our knowledge, there are no further numerical results and/or implementations so a comparison is informative.

6.3 Further Improvements and Conclusion

In this report, we have detailed the implementation of our specialized code designed to perform numerical simulations of a coupled problem between fluid dynamics and electrical phenomena. Through a systematic approach, we explained the mathematical model, its numerical discretization, our implementation and finally how to download and use our scripts. We also stressed about the objectives achieved and also those left incomplete. We conclude the report with the possible further improvements for the IPROP project:

1. Adapt the coupled algorithm to perform simulations on the other geometry of interest
2. Extension to 3D case
3. Inclusion of an adaptive grid refinement algorithm
4. Compute physical quantities of interest such as the ion thrust expectation

Bibliography

[1] Michael A. Lieberman and A. J. L. *Principles of Plasma Discharges and Materials Processing*. Wiley, 2 edition, 2005.

[2] Davide Cagnoni, Francesco Agostini, Thomas Christen, Nicola Parolini, Ivica Stevanović, and Carlo de Falco. Multiphysics simulation of corona discharge induced ionic wind. *Journal of Applied Physics*, 114(24):243302, 2013.

[3] Matteo Menessini. Numerical simulation of ion electrodiffusion and advection in atmospheric ionic thrusters. Master's thesis, Politecnico di Milano, 2023-24.

[4] H. K. Gummel. A self-consistent iterative scheme for one-dimensional steady state transistor calculations. *IEEE Transactions on Electron Devices*, 11(10):455–465, 1964.

[5] Jie Cheng. The 'time-dependent navier-stokes' code gallery program. `https://www.dealii.org/current/doxygen/deal.II/code_gallery_time_dependent_navier_stokes.html`, Contributed to deal.II, 2023.

[6] Timo Heister. *A Massively Parallel Finite Element Framework with Application to Incompressible Flows*. PhD thesis, University of Göttingen, 2011. Doctoral dissertation.

[7] W. Bangerth et al. The deal.ii tutorial step-22: Introduction to the incompressible navier-stokes equations. deal.II Library, 2016.