Particle in Cell Method for a 2D low density plasma simulation

Algorithm and Parallel Computing

Politecnico di Milano

A.Y. 2018-2019

Baioni Paolo J. – Vallisa Lorenzo

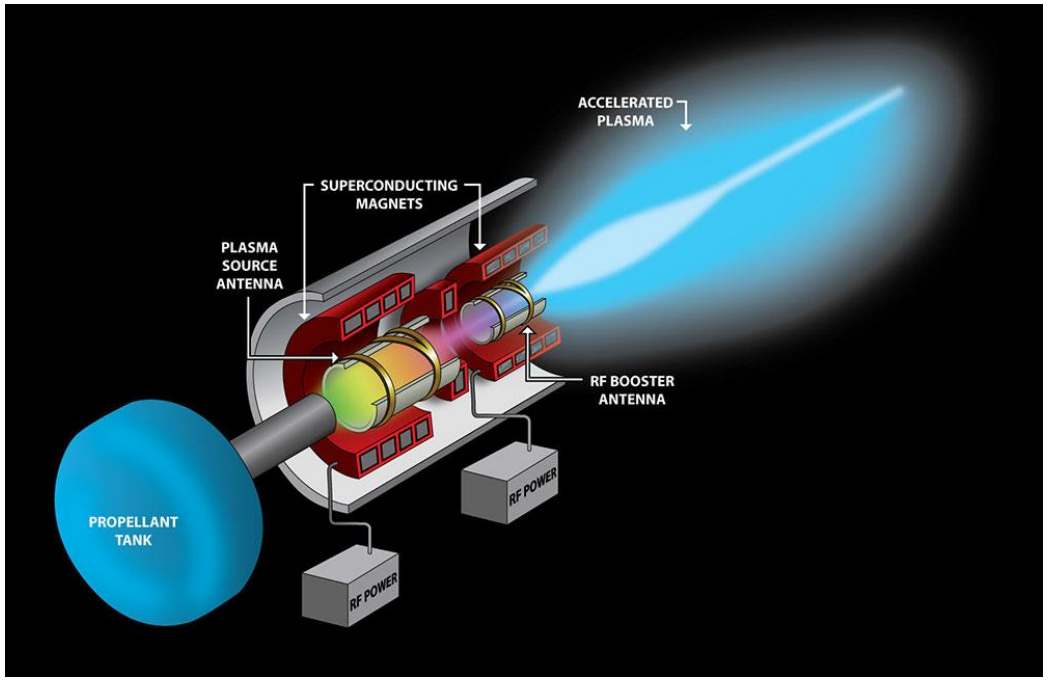
Introduction

Plasma: overall neutral system of particles in which there is a certain grade of ionization.

A plasma is characterized by the presence of a **self-consistend electromagnetic field**: the particles motion gives rise to the field which in turn influences their dynamics.

This system present local gas-like dynamics up to a characteristic length λ_D, beyond which collective modes prevail.

Physics of the simulation



In our simulation we have:

- 2D domain fill with low density plasma with a jet of plasma particles
- Absence of external fields
- Non relativistic particles

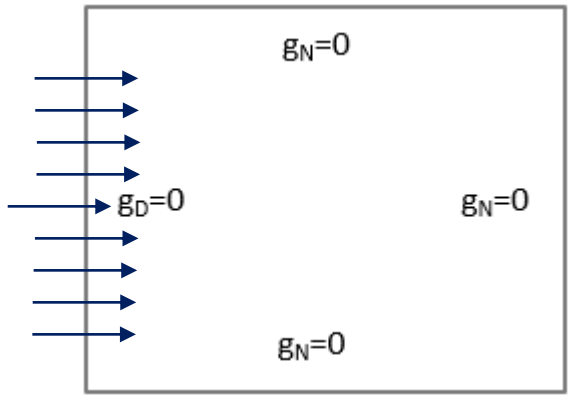
} → **F=qE**

We resolve the system up to the space scale of λ_D , thus focusing on the collective dynamics, and up to the time scale given by the characteristic time of the ions, much greater that the one of the electrons, which are then considered at equilibrium.

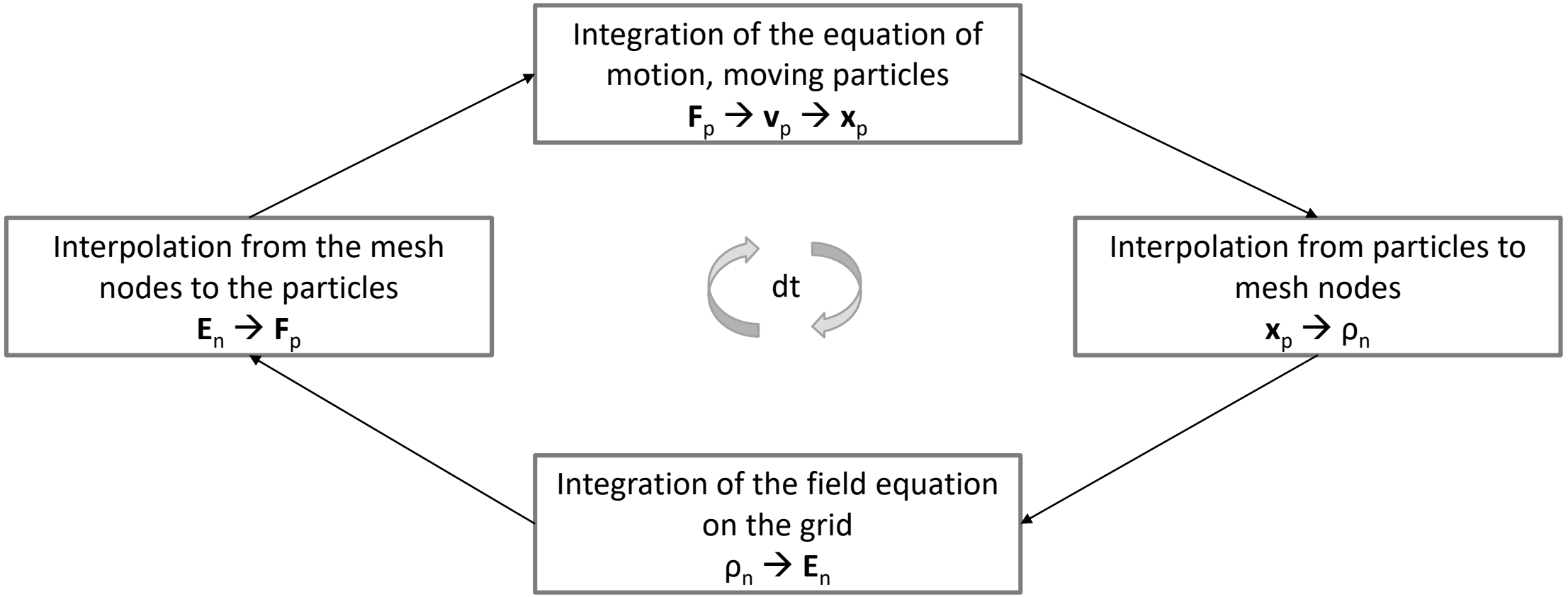
Model

$$\begin{cases} \Delta\varphi = -\frac{1}{\varepsilon_0}\rho(\varphi) & in\ \Omega \\ \varphi = g_D & on\ \Gamma_D \\ \partial_\nu\varphi = g_N & on\ \Gamma_N \\ \mathbf{E} = -\nabla\varphi & in\ \Omega \\ m\dot{\mathbf{v}} = q\mathbf{E} & in\ \Omega \\ \dot{\mathbf{x}} = \mathbf{v} & in\ \Omega \end{cases}$$

By using the Lorentz force instead of the Coulomb one we reduce the computational effort from $O(n^2)$ to $O(n)$, where n is the number of particles in the simulation.



The Particle in Cell Method



Algorithm

Allocate the mesh, the initial ions particles (optional) and the solver

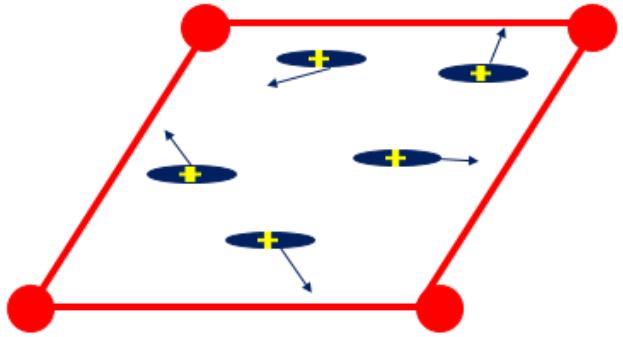
For each time iteration:

- Link particles to mesh cells
- Set nodes charge density, ions density from the particles, electrons density from Boltzmann distribution
- Get the scalar electrostatic potential with the solver, computing the electric field
- Set Lorentz force to the particles, moving the particles
- Remove the lost particles, add the new ones entering the domain through the jet

At 4 equidistant time iterations we print the electric potential and the velocity vector on 2 csv files, in such a way that we can visualize the evolution of the system. To do so we have written a matlab script that creates the countour plot of the electric potential and the vector plot of the velocity.

Macroparticle

Every macro-particle represents a bunch of real particles having a similar position and momentum (standard choice in PIC); the number of particles per macro-particle is called spwt (specific weight) and it is stored in Constant.h.

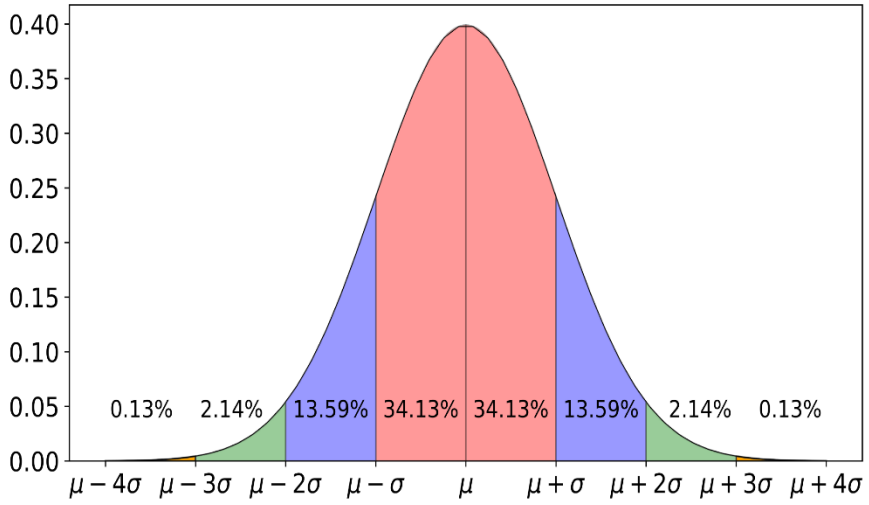


Implementative choices:
Every macroparticle is linked to its current cell through a shared pointer, because many different macro-particle could be in the same cell at the same time

MacroParticle
- double x,y,vx,vy,Ex,Ey - std::shared_ptr<Cell> my_cell
- double rand_pos(const double, const double) - double rand_vel(const double mu, const double sigma) + MacroParticle() + MacroParticle(double vdriftx) + double get_x() const + double get_y() const + double get_vx() const + double get_vy() const + double get_Ex() const + double get_Ey() const + std::pair<double,double> get_E() const + std::shared_ptr<Cell> get_mycell() const + void set_pos(const double xx, const double yy) + void set_pos(const std::pair<double,double> p) + void set_v(const double vxx, const double vyy) + void set_E(const double Exx, const double Eyy) + void set_cell(const std::shared_ptr<Cell> cella) +std::pair<size_t,size_t> indexes() const

The default **constructor** generates the macroparticle for the initial condition, the second one for the input through the jet on the left boundary.

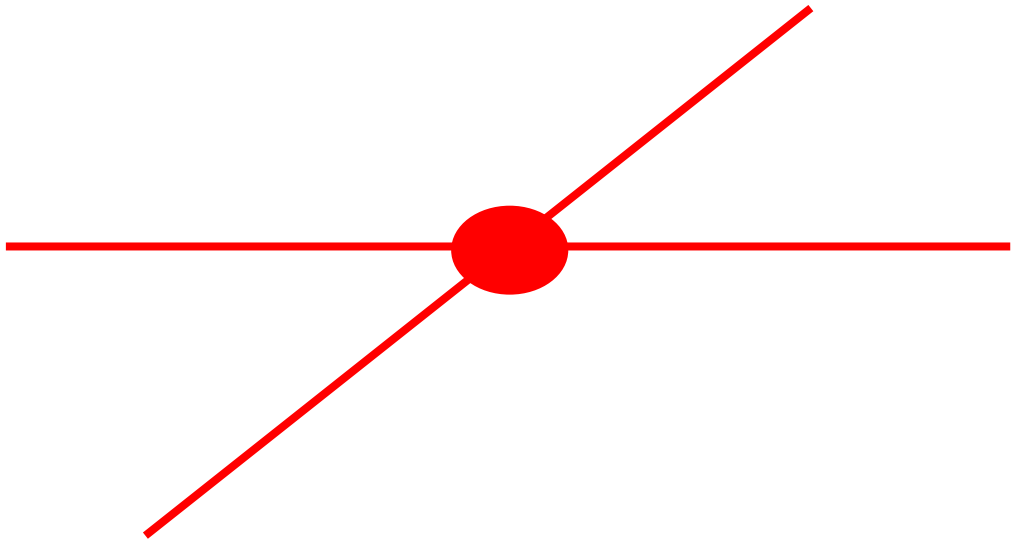
Both constructors rely on the STL random functions, in particular the velocity follows the Maxwell-Boltzmann distribution, that is a Gaussian N(mu,sigma) for each component of the vector.



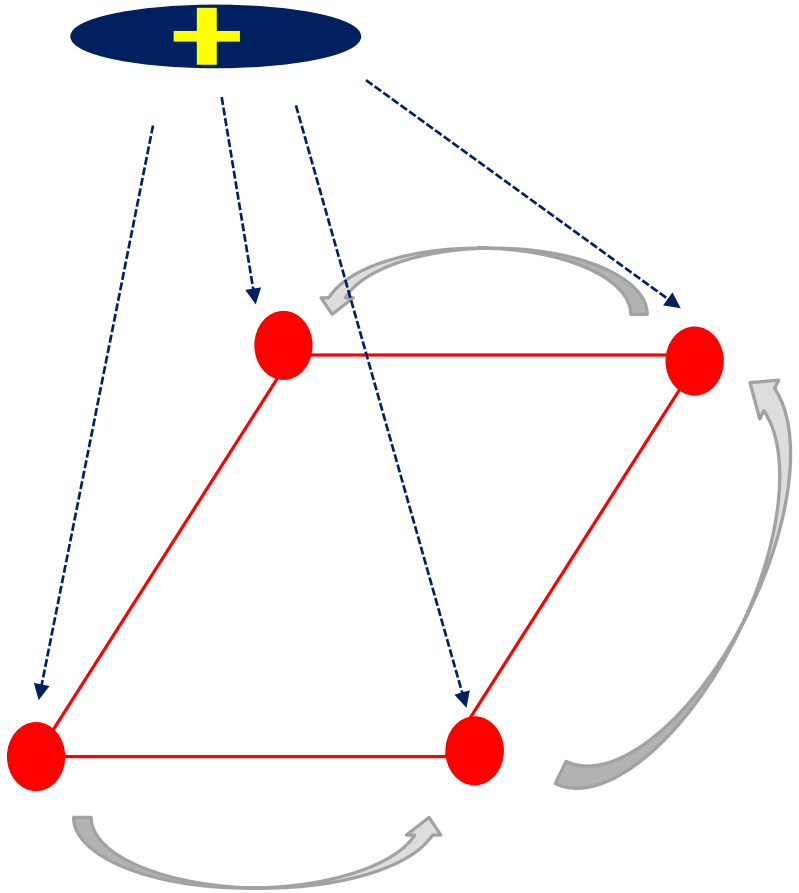
Implementative choices:

- the class has 2 private methods that generate the random values and that are called, with different parameters, by the two constructors.

Node



Node
- std::size_t i,j - double rho,Ex,Ey,vx,vy
+ Node(std::size_t ii, std::size_t jj):i(ii),j(jj){} + Node(std::pair<std::size_t,std::size_t> ind_pair):i(ind_pair.first),j(ind_pair.second) {} + reset_charge() + get_x()const + get_y()const + size_t get_i()const + size_t get_j()const + get_density_charge()const + get_Ex()const + get_Ey()const + get_vx()const + get_vy()const + reset_velocities() + operator<(const Node & l)const

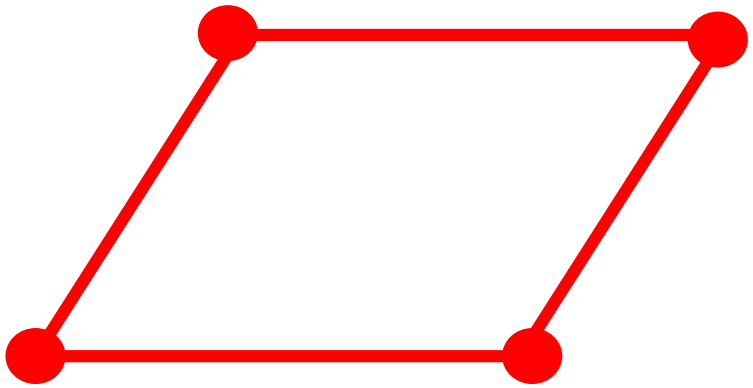


+ set_electric_field(double Exx, double Eyy)
+ set_density_charge(double new_rho)
+ next_first()const
+ next_second()const
+ next_third()const

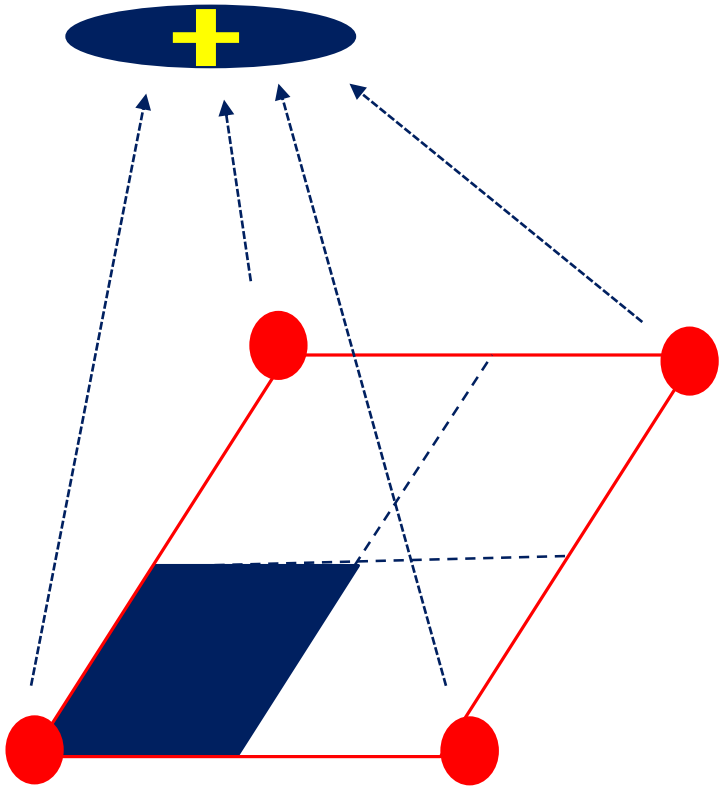
Implementative choices:

- Single class object instead of built-in variable: flexibility in data management
- Need of inequality operator

Cell



Cell
- array<std::shared_ptr<Node>,4> cell_nodes
- compute_surface(double x1, double x2,double y1, double y2)const + Cell(std::array<std::shared_ptr<Node>,4> nodes):cell_nodes(nodes) + void print_my_nodes() + void update_velocity(double x,double y,double vx,double vy)

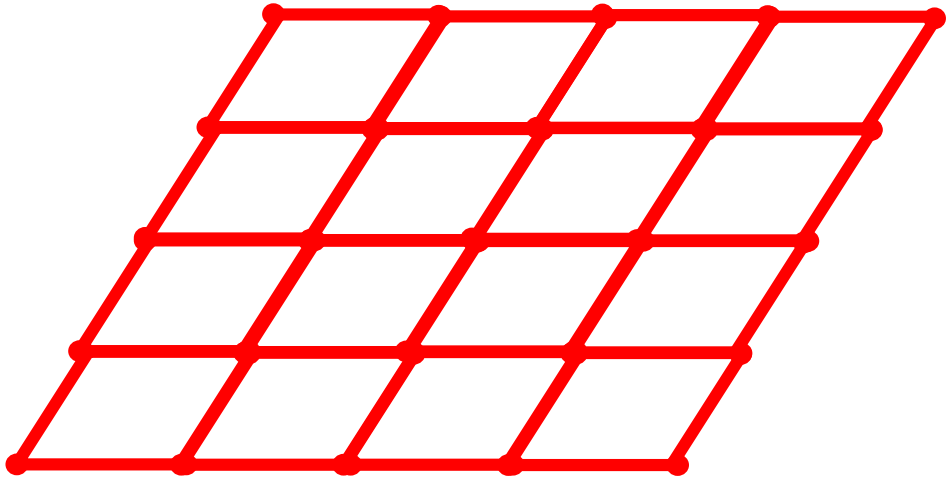


+ update_field(const double x, const double y)const
+ update_charge(const double x, const double y)

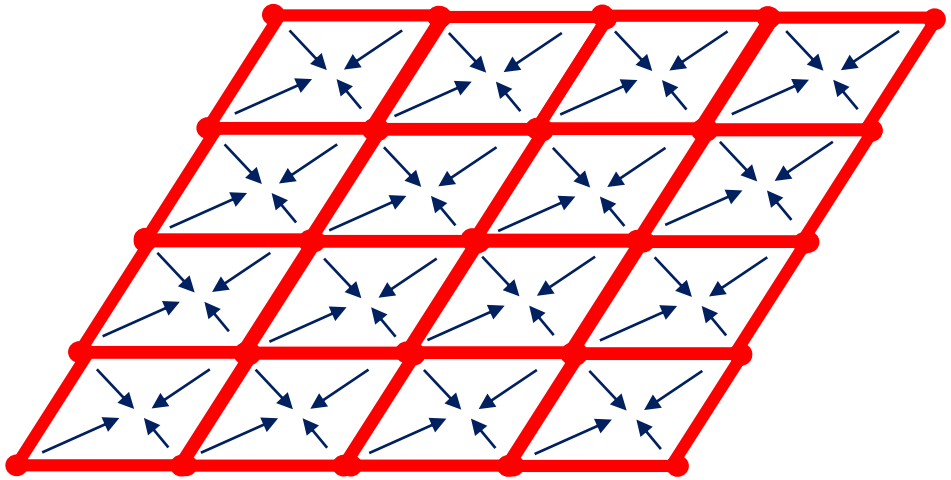
Implementative choices:

- ***shared_ptr*** needed since Node is shared within Cell object, and Cell can have up to two Node in common: every method calls the pointer to Node to act on the field
- Cell is key feature in algorithm: connects Macroparticle functions with domain variables needed for Electrostatic field computation (i.e. collects electric fields on its node, sum the values and give it to Macroparticle)

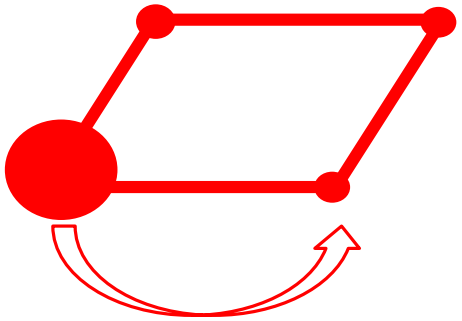
Mesh



Mesh
typedef std::pair<std::size_t,std::size_t> indexes typedef std::array<std::shared_ptr<Node>,4> cell_nodes - **unordered_map**<indexes,std::shared_ptr<Cell>,my_hash> mesh_map - std::**array**<std::shared_ptr<Node>,constant::nodes_number > nodes
+ find_my_cell(const indexes & p) const + print_my_cell()const + reset_velocities_nodes() + reset_mesh_charge() + *get_rho_ions()* const + *update_electric_field*(const std::array<std::pair<double,double>,constant::nodes_number> &) + ofstream & print_nodes_velocity(std::ofstream & os)



Cycling every Cell and finding lower left index



Using *next* methods of Node to find adjacent one

```
+    Mesh()
+    add_element(const cell_nodes & nodi,const indexes & p)
-    get_lower_left_indexes_pair(const std::size_t & xx)const

struct my_hash{
  size_t operator()(const std::pair<std::size_t,std::size_t> &
coppia)const{
    return (coppia.first*constant::x_nodes+coppia.second);
  }
};
```

Implementative choices:

- Use of both ***array*** and ***unoredered_map:*** need to cycle over all Node but at same time optimium algorithm to find Cell, given a macroparticle position, within the Mesh (use of array since fixed Mesh size)
- Constructor of Mesh: when creating Node, assigning a shared_ptr to each of them, and saving them anticlockwise four by four to each Cell

System

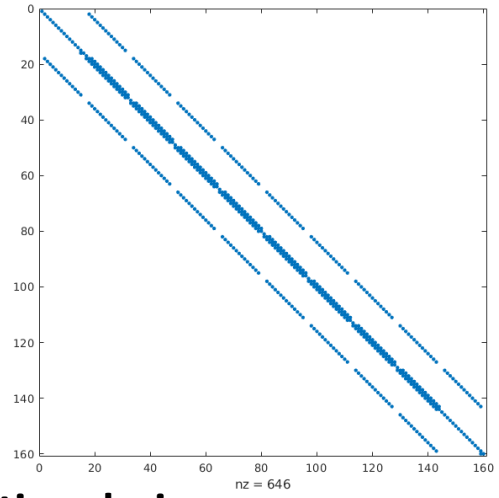
Implementative choices:

- use of forward_list to contain the particles since:
  - we need to scan the whole container multiple times
  - we need to easily remove particles that escape from the domain (std::forward_list.remove_if(…))
  - it’s cheaper than a std::list

System
- std::forward_list<MacroParticle> mplist - Mesh mesh
+ System() + void print_particles() const + void assign_cells() + void update_charge() + std::array<double,constant::nodes_number> get_rho_ions() + void set_nodes_field(std::array<std::pair<double,double>, constant::nodes_number> E) + void update_particles_field() + void initial_velocity(const double& dt) + void motion_equation(Solve & solref, const double& dt) + void interp_vel() + void print_vel(std::ofstream& os) + void reset_node_velocities() + void check_position() + void emplace_front(std::size_t n) + void reset_ion_charge()

Matrix

We use the class Matrix to allocate the matrix A that solves the Poisson problem.

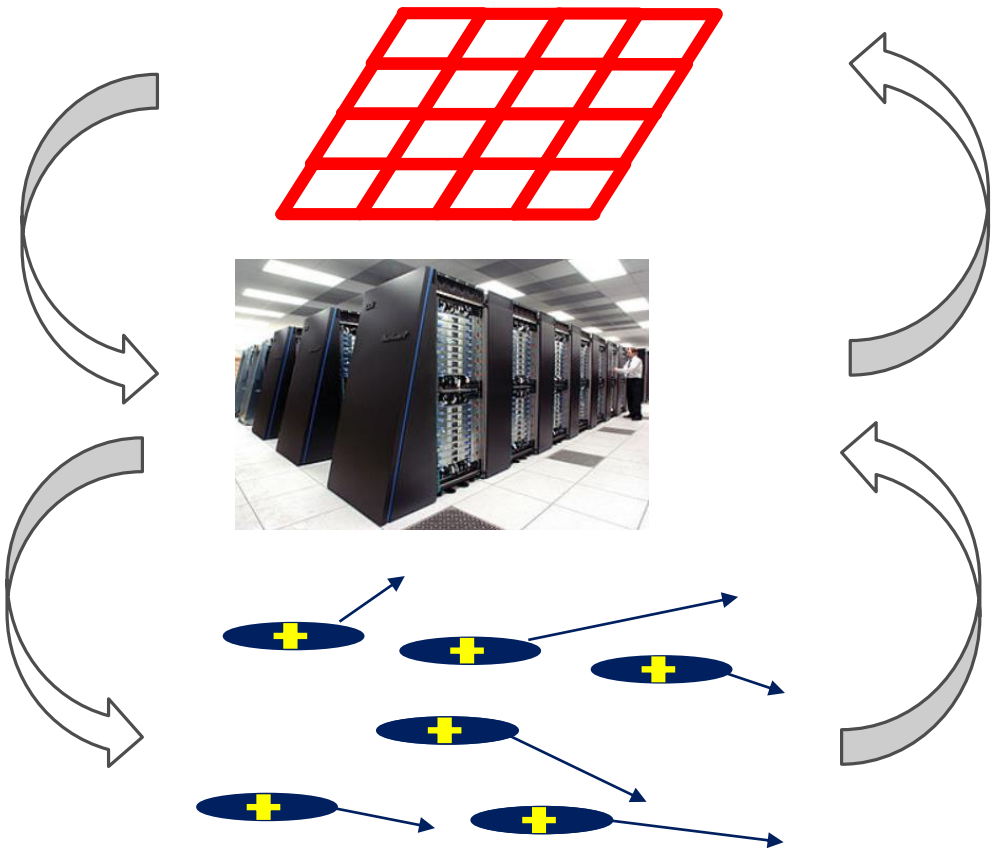


Implementative choices:

Even if the matrix is sparse, we use std::vector instead of a map because our matrix contains 160x160 doubles, and thus it takes up only ~ 0.2 MB. Moreover, due its small size, it fits in the cache, so std::vector results very well performing.

Matrix
- std::vector<double> elements - const std::size_t rows,cols
- inline std::size_t sub2ind (const std::size_t i, const std::size_t j) const - double & index (std::size_t i, std::size_t j) - const double & const_index (std::size_t i, std::size_t j) const + Matrix(std::size_t sz) + Matrix (std::size_t r, std::size_t c) + Matrix (Matrix const &) + std::size_t get_rows () const + std::size_t get_cols () const + double & operator() (std::size_t i, std::size_t j) + const double & operator() (std::size_t i, std::size_t j) const + double * get_elements () + const double * get_elements () + std::ofstream & print(std::ofstream & os) + double scalar_product(size_t i, size_t j1, size_t j2, const std::array<double,constant::nodes_number>::iterator ptr)const

Solve



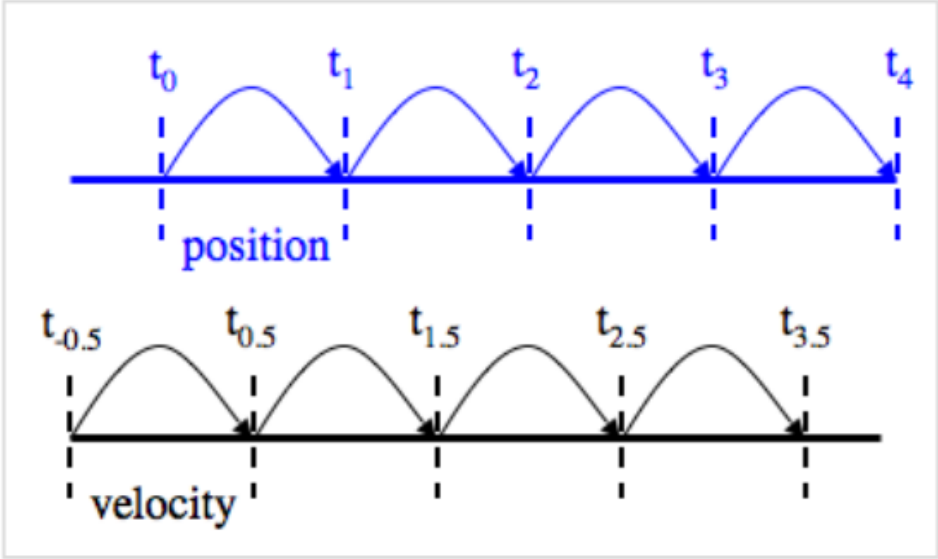
Solve.h
- tol = 1e-2 - Matrix A
- get_nodes_index(std::size_t i,std::size_t j)const - compute_residual(…)const + **Solve**() + **leapfrog**(…); + eulero_avanti(…); + **compute_electric_field**(const std::array<double,constant::nodes_number> & phi) + **poisson**(const std::array<double,constant::nodes_number> & rho_ions, const std::array<double,constant::nodes_number> & phi_start)

Leapfrog algorithm

The function tuple<doubles...> leapfrog(doubles...) implements the leapfrog method for the solution of the Newton-Lorentz equation.

The method is stable since $\Delta t_x \omega_p \sim 0.1$, of order 2 and symplectic, but it requires to set v(t)=v(t-½) at the very first iteration, since it does a shifted integration of x and v. This is done by calling System::initial_velocity in the main, so that line has to be commented out if System::motion_equation calls another solver, like Solve::eulero_avanti.

These solvers share the same signature but for the different name, so changing the time integrator requires only a 1 word change in System::motion_equation.



Non-Linear Poisson equation

$$\Delta\varphi = \frac{e}{\varepsilon_0}\left(Zn_{ions} - n_0 e^{\frac{\varphi}{K_B T_e}}\right)$$

Equation to compute electrostatic field given ions and electrons charge, the latter one according to Maxwell-Boltzmann distribution. Goal is hence to discretize the Laplacian following a centered finite difference method:

$$\frac{\varphi_{i-1,j} - 2\varphi_{i,j} + \varphi_{i+1,j}}{\Delta^2 x} + \frac{\varphi_{i,j-1} - 2\varphi_{i,j} + \varphi_{i,j+1}}{\Delta^2 y} = \frac{e}{\varepsilon_0}\left(Zn_{ions} - n_0 e^{\frac{\varphi}{K_B T_e}}\right)$$

Getting eventually to the final algebraic non-linear system of equations:

$$\mathbf{A}\varphi = \mathbf{b}(\varphi)$$

Gauss-Seidel algorithm

Main strategy now is to implement an iterative method, such as the fixed point algorithm (proved to be fast enough convergent for this problem), in order to reduce it to a linear system of equation:

$$\mathbf{A}\varphi = \mathbf{b}(\varphi)$$

After having implemented the boundary condition in A (done while constructing the A matrix) and in b (done at every iteration, since it depends on the previous value of the electrostatic potential), the final step is to implement an algorithm for solving linear system, in this case the Gauss Seidel algorithm has been chosen:

$$\mathbf{A} = \mathbf{P} - \mathbf{N} \qquad \mathbf{P} = \mathbf{D} - \mathbf{E} \qquad \mathbf{N} = \mathbf{F}$$

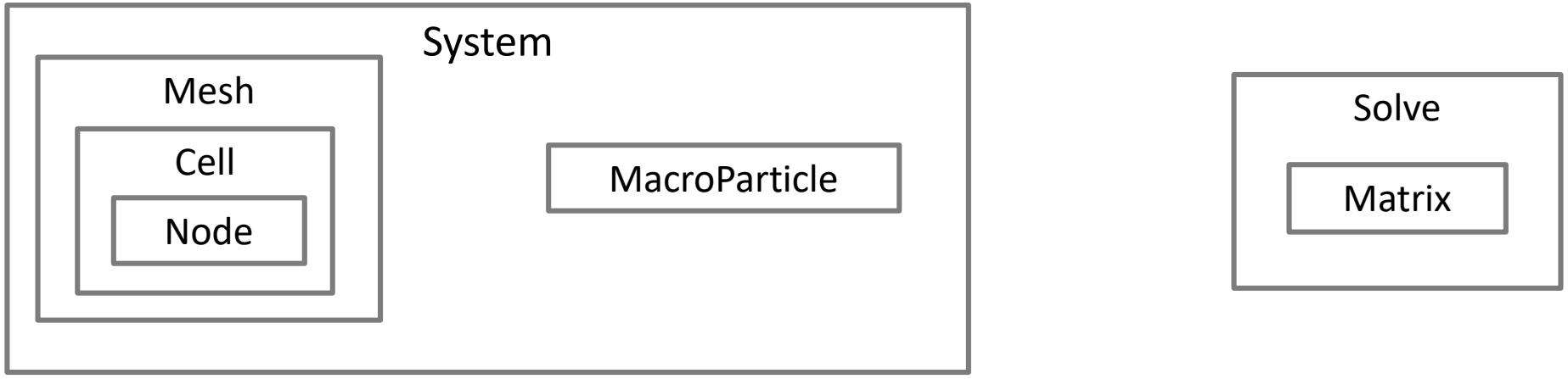
$$\mathbf{P}\mathbf{x}^{(k+1)} = \mathbf{N}\mathbf{x}^{(k)} + \mathbf{b}, k \geq 0$$

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right], \quad i = 1 : n$$

D diagonal matrix of A, E upper triangular matrix of A, F lower triangular matrix of A

Structure of the code

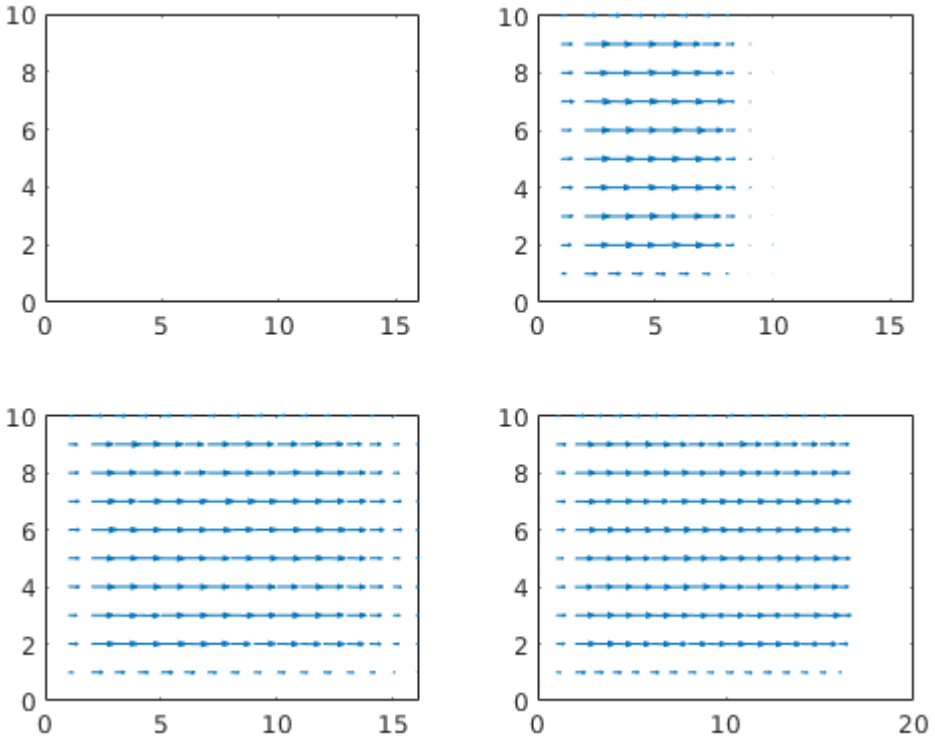
Classes:

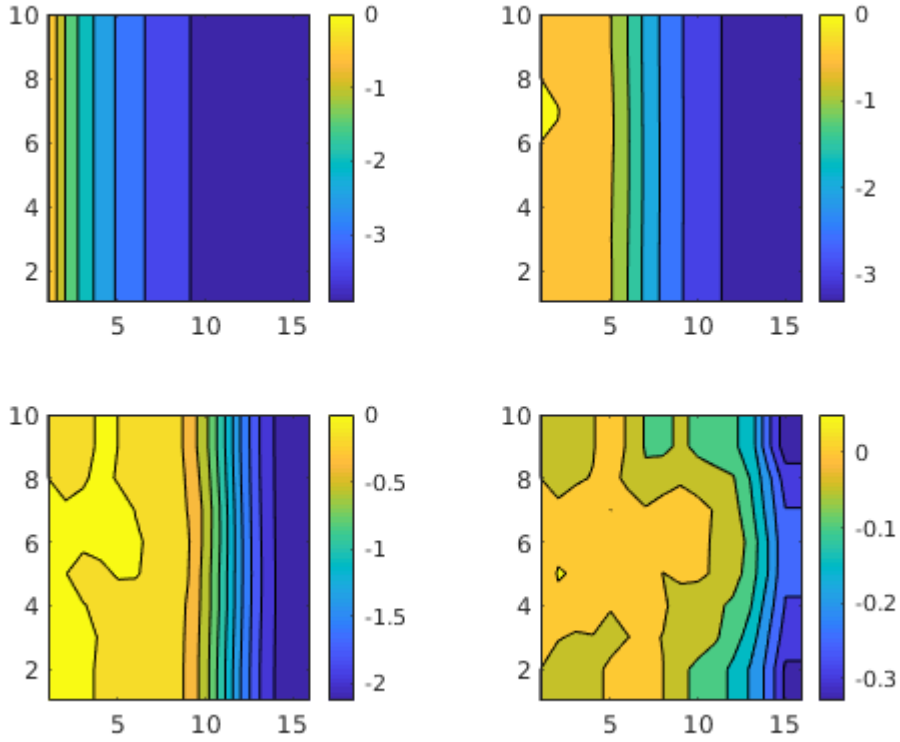


Other files: (the .csv files are created during the execution)

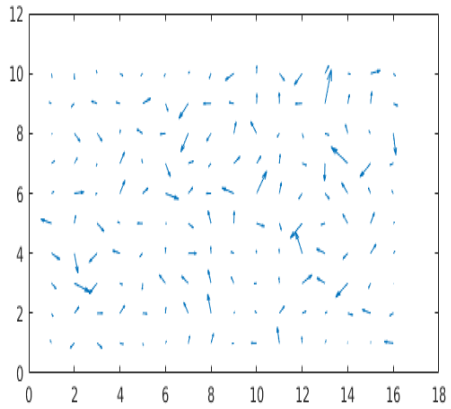
main.cpp | Makefile | Constant.h | graphs.m | Phi.csv | NodesVelocity.csv | README

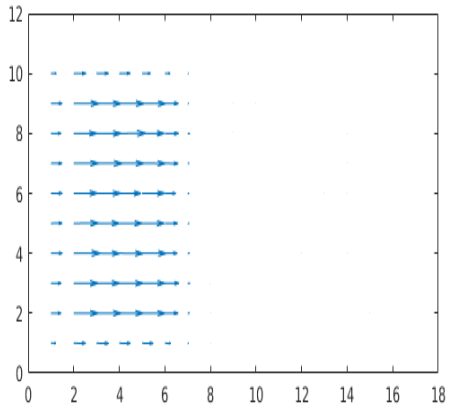
Results (empty domain)

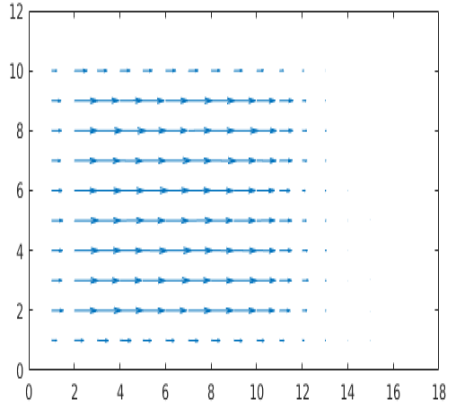


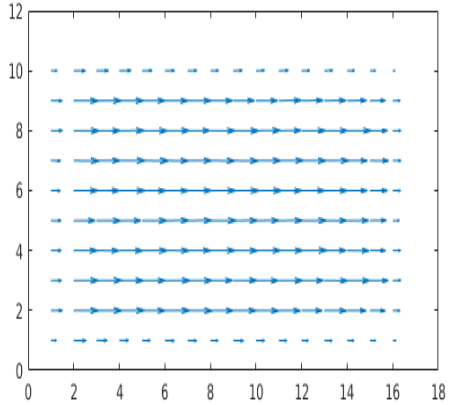


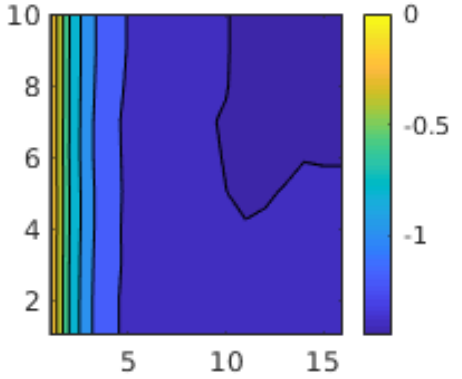
Results (full domain)

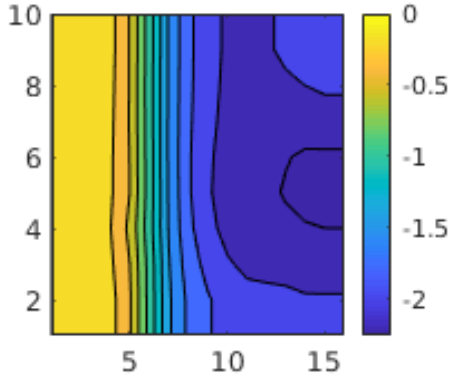


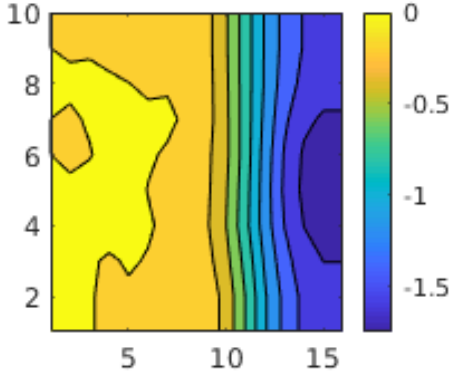


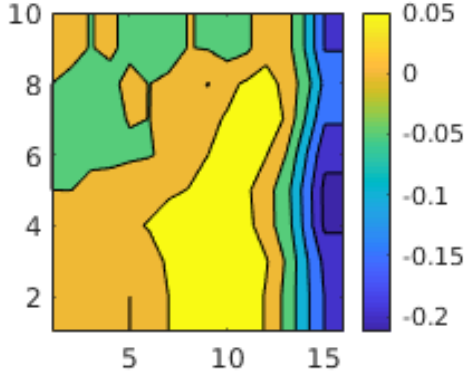












References

- C. K. Birdsall, A. B. Langdon, Plasma Physics via Computer Simulation, Institute of Physics – Series in Plasma Physics, 2004
- A. Quarteroni, R. Sacco, F. Saleri, P. Gervasio, Matematica Numerica, Springer, 2014