Universidade de São Paulo

Escola Politécnica

Departamento de Engenharia de Estruturas

Trabalho de Formatura - 2015

Computational modelling of particle flows under the influence of external electromagnetic fields

Orientador:

Prof. Eduardo M.P. Campello

Aluno:

Bruno Maria Calidonna

Catalogação-na-publicação

Calidonna, Bruno Maria
Computational modelling of particle flows under the influence of external electromagnetic fields / B. M. Calidonna -- São Paulo, 2015.
76 p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Estruturas e Geotécnica.

1.Eletromagnetismo 2.Geometria e modelagem computacional I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Estruturas e Geotécnica II.t.

Alla mia famiglia,
Che anche da lontano, con sorrisi e rassicurazioni,
sempre mi supporta e mi aiuta.
Senza di voi, non sarei me stesso.
Ao meu orientador, o Prof. Campelo,
pela confiança, o suporte e a sua guia.
Graças à você, voltei à amar a Engenharia.
À Bruna e Simone,
Que foram meu motor e minha consciência nos momentos
Menos fáceis.

E a me stesso.
Perché me lo merito.

Abstract

The result of this work is a completely working and validated simulator for particle interactions simulation under external electromagnetic fields.

The forces considered for the resolution of the governing equation of the problem, which is a differential equation of second order, are near-field forces, due to the electrical charges of other particles, contact forces, due to the shock with other particles and barriers, and electromagnetic forces, due to the presence of external electromagnetic fields.

Contact is considered to be elastic, modelled by the Herztian equation.
Near-field are modelled by empirically-calibrated formulae.
Electromagnetic forces are modelled straight by the theoretical solutions.

This work features a triangular, finite, planar 3D barrier element. This element requires just three points as input to be defined and allows the representation of basically every possible contact surface.

Sumário

Introduction

Correctly simulating and foreseeing the movements of particles subjected to electromagnetic fields is quite a feature required by industry, nowadays.

In fact, countless industrial process are carried by means of charged particles.
As an examples:

- Industrial painting;
- Electrostatic precipitators;
- Printing processes;

Also, analizing the dynamics of solid particles can be implemented, under other but not-so-far conditions, in other areas, such as:

- Drug delivery processes;
- Filtering processes;
- Separation processes;

As it is possible to see, a simulator who is able to predict the movement of a certain number of particles after a certain kind of input can be of great help in order to calibrate that input itself.

It is an industrial-applicable feature that could help optimize and help succeed countless processes.

This simulator has some key-features implemented:

- Hertzian Elastic particle-particle and particle-barrier contact;
- Near-field forces caused by electric charges of the particles themselves;
- Electromagnetic forces caused by external electromagnetic fields;
- triangular, finite, planar 3D barrier element;
- A space object that collects different barrier elements and can thus potentially identify every possible surface;
- Brute-force approach in contact detection;

Every key-feature will be presented and justified.

This handout will present how the problem was tackled from the mathematical modelling point of view, then to its coding and validation and finally ending with some Case Studies in order to show the potentialities of the simulator.

Problem formulation

In this chapter, basic theoretical foundations of this work will be reviewed.
Particle dynamics will be briefly tackled in order to understand how our physical system is mathematically modelled.
We will then proceed to introduce the chosen models for the forces considered.

Particle dynamics

Let´s introduce a Cartesian coordinate system defined by the mutually orthogonal triad **e₁**, **e₂** and **e₃**. We define as **r** the position vector of a given particle. We can thus define the kinematics of a single particle as:

$$\begin{cases} \boldsymbol{r} = r_1\boldsymbol{e_1} + r_2\boldsymbol{e_2} + r_3\boldsymbol{e_3} \\ \boldsymbol{v} = \dfrac{d\boldsymbol{r}}{dt} = \dot{\boldsymbol{r}} = \dot{r_1}\boldsymbol{e_1} + \dot{r_2}\boldsymbol{e_2} + \dot{r_3}\boldsymbol{e_3} \\ \boldsymbol{a} = \dfrac{d\boldsymbol{v}}{dt} = \ddot{\boldsymbol{r}} = \ddot{r_1}\boldsymbol{e_1} + \ddot{r_2}\boldsymbol{e_2} + \ddot{r_3}\boldsymbol{e_3} \end{cases}$$

Where we identify the bold terms as vectors, while the non-bolded ones are the Cartesian norm of the vector projected onto the *i* direction.

We can identify the force generated by the electromagnetic phenomena as:

- $\boldsymbol{\psi}^e = q\boldsymbol{E}$ due to the electric field E
- $\boldsymbol{\psi}^m = q\boldsymbol{v} \times \boldsymbol{B}$ due to the magnetic field B

In a more general form, we can write:

$$\boldsymbol{\psi}^{em} = \boldsymbol{q}(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B})$$

Which allows us, thanks to Newton's second law, to get to our governing equation:

$$\dot{\boldsymbol{v}} = \frac{q}{m}(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B})$$

Solving in terms of position vectors, we get:

$$m\ddot{\boldsymbol{r}} = q\ (\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B})$$

Or, getting back to the more general form, considering a generic particle i:

$$m_i\ddot{\boldsymbol{r}}_{\boldsymbol{i}} = \ \boldsymbol{\psi_i}^{\boldsymbol{em}}$$

As a matter of fact, we are not yet satisfied with this modelling.
In fact, at the moment, the equation reproduces the dynamics of particles subjected to an external electromagnetic field, but doesn't take into account contact forces or particle-particle interaction.

We thus need to better model the forces acting on our system.
Considering a particle i, we want the equation to look like this:

$$m_i\ddot{\boldsymbol{r}}_{\boldsymbol{i}} = \boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{tot}}$$

Where our forces are defined by three different members:

$$\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{tot}} = \boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{em}} + \boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{nf}} + \boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{con}}$$

- $\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{em}}$ is the force generated by the presence of an external electromagnetic field;
- $\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{nf}}$**,** called "**near-field interaction"** is the force generated by the proximity of other particles;
- $\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{con}}$ is the force generated by contacts with other particles and/or barriers;

While for the electromagnetic fields the implemented model is coincident to the theoretical one, for near-field interaction and for contact we have to dwelve a bit deeper: different models with different hypothesis and consequences, in fact, exist in order to represent these forces.

Near-field interaction

The theoretical interpretation of the interaction force between two particles, i and j, due to their respective charges q is given by Coulomb's law:

$$\boldsymbol{\psi}_{\boldsymbol{ij}}^{\boldsymbol{nf}} = \frac{q_i q_j}{4\pi\epsilon\left\|\boldsymbol{r}_i - \boldsymbol{r}_j\right\|^2}\boldsymbol{n}_{ij}$$

Where q are the respective electric charges for the two particles, ϵ is the permittivity and $\boldsymbol{n}_{ij}$ is the normal direction determined by the difference in the position vectors of the particles' centers, defined by:

$$\boldsymbol{n}_{ij} = \frac{\boldsymbol{r}_i - \boldsymbol{r}_j}{\left\|\boldsymbol{r}_i - \boldsymbol{r}_j\right\|}$$

However, for massive systems, we cannot apply this model, since Maxwell equations' should be applied to each of the components, generating an enormous quantity of equations.
Thus, we will use empirically generated interaction laws.

There are vast numbers of empirical representations that are reported in Literature.
However, we will use the model presented by Zhodi [1]. The resulting force on particle i due to near-field interactions $\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{nf}}$ is given by:

$$\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{nf}} = \sum_{j\neq i}^{N_p}\left(\alpha_{1ij}\left\|\boldsymbol{r}_i - \boldsymbol{r}_j\right\|^{-\beta_1} - \alpha_{2ij}\left\|\boldsymbol{r}_i - \boldsymbol{r}_j\right\|^{-\beta_2}\right)\boldsymbol{n}_{ij}$$

Where α e β are properly defined coefficients.
We opted for this modelling because it can effectively represent different kind of interactive forces, not just the ones of electric nature.
In fact, α e β can be, also:

- Mass based
- Surface area based

- Volume based

This could result helpful in case of future developments of the simulator.
At the moment, we will focus just on charge-based interactive forces.

Now that the near field interactive forces $\boldsymbol{\psi}_i^{nf}$ are defined, we need to focus on the contact modelling.

Contact mechanics

Contact mechanics defines two opposite poles of mechanical interaction between two bodies that are coming into touch:

- Perfectly Elastic shock
- Perfectly Plastic shock

In the first case, the two bodies come in touch and no energy is dissipated during the contact, neither the particles are subjected to any permanent deformation: it's the case, for example, of a tennis ball that shocks with a wall; it just comes back with approximately the same speed with which it shocked against the wall.
If the shock was perfectly elastic, that speed would be exactly, in magnitude, the same as in which it came, but with opposite direction.

On the other hand of the "contact-spectrum", we have the perfectly plastic contacts, in which the two bodies becomes one, joining their masses and respecting the continuity of movement.
It's the case of two lovers who are meeting at the airport: one runs and jumps on the other, hugging him/her; a few steps back are taken, in order to accommodate the new quantity of motion attained.
If it were a perfectly plastic shock, with no friction whatsoever, the two of them would be happily hugging themselves in an endless motion defined by the speed generated in the instant just after the contact.

Defining plastic shocks, besides the necessity of solving continuity equations, involves a better definition of the material and its deformations.
In order to represent a more realistic contact model, we should in fact address the reality of the process of shock, meaning studying the approaching part, the contact part and the release part that all present different material behaviours.

We decided, thus, to create a functioning simulator based on "simpler" mechanics, meaning elastic.

It was thus implemented an Elastic contact law known as Hertzian contact.

We will omit the derivation of this law (that can be found in [8]). It is however important to focus on the hypothesis it considers:

- The strains are small and within the elastic limit.
- The surfaces are continuous and non-conforming
- Each body can be considered an elastic half-space.
- The surfaces are frictionless.

It is obvious that by using this model we are making some simplifications (first of all the no-friction hypothesis), but under our conditions, at the moment, this is not an excessively restricting condition.

We can thus define the governing equation as:

$$\boldsymbol{\psi}_{\boldsymbol{i}}^{\boldsymbol{con}} = \frac{4}{3} E^* \delta^{3/2} \sqrt{R^*} \boldsymbol{n}_{ij}$$

Where R* is the equivalent radius:

$$R^* = \frac{R_i R_j}{R_i + R_j}$$

E* is the equivalent elastic modulus:

$$E^* = \frac{E_i E_j}{E_j(1 - \upsilon_i^2) + E_i\left(1 - \upsilon_j^2\right)}$$

And δ is the compenetration between the two bodies

$$\delta = \left\|\boldsymbol{r}_i - \boldsymbol{r}_j\right\| - \left(R_i + R_j\right)$$

Modelling contact through Hertz equations has another bonus.
Since one of the hypothesis is that we are considering the bodies as half-spaces, nothing implies that we can’t use the same equation in order to define the contact with a barrier.
In that case, all the above equations result unaltered, except the one of the equivalent radius. In fact, we can write that equation in this way:

$$\frac{1}{R^*} = \frac{1}{R_i} + \frac{1}{R_j}$$

If we imagine the body j to be a barrier, a planar surface, we have that its radius R_j is infinite, meaning that its inverse is zero. Thus, in the case of particle-barrier contact, our equivalent radius is simply defined by:

$$R^* = R_i$$

We have hence defined the contact force, meaning that we have completed the mathematical modelling of our problem.

Before going over the solution implemented in the simulator, let’s make a brief detour looking at some other solutions that can be found in literature.

Proposed Solution

In this chapter, the adopted solution will be explained.
Main algorithm and code main structures, functions and dynamics will be hereby on presented.

Main algorithm

```
i)    If first time-step: Reading input datas;
      a. Else: resetting forces and auxiliary variables to null values;
ii)   Computing particles positions;
iii)  Computing near-field forces;
iv)   For each particle:
      a. Checking for contacts with other particles;
      b. Checking for contacts with barriers;
            i. If contact is detected, resulting force is stored as barrier element value too.
      c. Evaluation of speed and position increments;
      d. Convergence check:
            i. If tolerance is met, move to next particle
           ii. If tolerance is not met, iterate calculations
v)    Transfer time-step final data set to output file;
vi)   Move to next time step
```

As an important note to point iv.d.ii, the code is engineered so that the iteration in the same time-step happens through a local and delimited time-step refining.
Iterations are made on the single particle, preserving physical (real) time and not interfering with the general time step.

Objects implemented

This code has three main physical objects:

a) Particle;
b) Barrier;
c) Space;

It was also defined a fourth, virtual one, which gathers some external and computing conditions:

d) Fields;

We will now discuss basic elements and operations of each object.

Particle

Each particle is defined by a set of datas:

1) An integer which identifies the particle;

2) An array of floats which defines the position of the particle relative to a global Cartesian coordinate system;
3) An array of floats which defines the velocity of the particle, referred to the three Cartesian coordinates;
4) An array of floats which defines the acceleration of the particle, referred to the three Cartesian coordinates;
5) A float which identifies its charge;
6) A float which identifies its radius;
7) A Young modulus;
8) A Poisson modulus;
9) A material density value;

Besides that, each particle has a set of auxiliary variables which help with the iteration and solution of the system.
To be more specific, it has float arrays for accellerations and forces and float values for the computation of errors in the evaluation of new positions and speeds.

Barrier

The input for each barrier triangular element, whose modelling will be presented later in this chapter, is composed solely by an integer which identifies the element and the spatial coordinates (in the three main Cartesian coordinates) of three points.

After that, each barrier element is computed and defined by:

1) A float array defining its center;
2) Three float arrays defining its normal and two perpendicular directions on the surface of the element, computed as versors;
3) A float array identifying the forces acting on the element;
4) Three float array vectors identifying the vectors connecting the center to each of the three vertexes;
5) Three floats defining the norm of such vectors;
6) A float array identifying the progressive angles between the arrays.

As an important remark, the direction of the vector connecting center and the first point inputed is defined as tangent, while its rotation of ¾ Pi is defined as its perpendicular, thus making the normal face outside the element.

The code works in a way that the barrier is initialized just at the beginning of the simulation, retaining its datas until the end of it.

Space

The object space is the collection of all the barriers present in the simulation.
It is defined by:

1) A Barrier-type array containing all the inputed barriers;
2) An integer defining the total number of barriers;

It also has an auxiliary integer which helps the computation of the correct effects of contact in case a particle should simultaneously hit more than one barrier element.
This is needed because, as we said in “Problem Formulation” we are computing the effects of the

impact on the barrier as the one between a sphere and a half-space, in Hertzian conditions.
This integer will basically divide the effects of the force applied by and on each element.
This naturally implies that there's no actual difference if the particle hits the center or the border of a single Barrier element.
This is perfectly consistent with the fact that we are treating the element as a rigid half-space, defined basically by its normal.
For small enough elements, this hypothesis is completely sound.

Fields

This virtual object contais:

1) A float which defines the time-step of the simulation;
2) A float which defines the time integration scheme we want to use;
3) Floats which define the magnitude of electric and magnetic field in each of the three Cartesian coordinates componets;

Objects communication and interaction

For the sake of better understanding the dynamics of the simulator, we will briefly present the internal dynamics between objects.

As we said, the object Space is composed solely by all the barriers inputed. Basically, we can say that Space sees Barriers, but Barriers don't see Space.
However, Space acts on Barrier, since it updates the force values generated by the impact with a Particle.
Barrier is basically a passive object, which define itself by the means of some routines and then is included into another object, which takes care of its communication with the experimental environment.

The object Particle sees all the others Particles and the Space.
By this means, Particle is an active object, which interacts with all the other objects of the experimental environment simultaneously and at all times.

Contact dynamics are implemented when contact is detected. To do so, a brute force approach is implemented, meaning that at each step of the simulation, the position difference vector between a Particle and all the others and the barrier elements contained in Space is computed. If this difference (the normal component of it in case of Particle-Space interaction) should be negative (and tangential distance should be included in the boundaries of the element, in case of Particle-Space interaction) such dynamics are triggered.

Barriers and Space are *optional* objects, meaning that is completely possible to run a simulation without any barriers present.

Also, it is possible to nullify some values in the Fields object, like the electromagnetic field, leaving it a pure mechanical simulation of solid spheres and barriers.

Barrier element modeling

The barrier element implemented is a three-dimensional planar triangular surface.

The only input needed for our barrier are its three points, defined in the space of the three Cartesian coordinates, an identifier, a Young and a Poisson modulus.

Having that, the code performs once and at the very beginning of the simulation, a geometric initialization of the barrier element.

After that, each element is assigned to the object Space.

Before diving into the mathematical simulation of the barrier, let's have a general overview of its communication with the particle.

Two main checks are performed, when trying to identify contact with a particle:

1) First, a *normal distance* check: the distance between particle and barrier is projected onto the normal. Since this distance is defined considering the center of the particle, we subtract to this value the value of the Radius of the particle. Then, if and just if the normal distance is lesser or equal than zero, we proceed to the second check;
2) A *tangential distance* check: given the normal distance value, the distance between particle and barrier is projected onto the tangential plane of the element; it is then checked if the distance lies between the boundaries of the element, once discounted by the *half-chord* obtained "cutting" the particle at the plane of the element.

If these two checks are positive, we apply contact dynamics.

Let's dive a bit deeper into the formulation of this element.

Geometry definition

Given three points in space, the center of the element is computed as:

$$C_x = \frac{x_1 + x_2 + x_3}{3}$$

$$C_y = \frac{y_1 + y_2 + y_3}{3}$$

$$C_z = \frac{z_1 + z_2 + z_3}{3}$$

$$C = (C_x, C_y, C_z)$$

The center of the element will be used as the reference point for the barrier element for the Particle-Space interaction. Also, normal, tangent and the perpendicular to the tangent will have their origin here.

Once the center is known, we compute the three vectors connecting the center to the three vertexes as:

$$c_{ij} = C_j - p_{ij}$$

$$c_i = (\, c_{ix}, c_{iy}, c_{iz})$$

Where i is the number of the vertex (1,2 or 3) and j is the considered direction (x, y or z).

The direction defined by c1, meaning the vector connecting the center to the first vertex will be considered as the main tangential one for the element.

The cross product between c1 and c2, the vector connecting the center to the second vertex inputed, will be then performed, identifying the normal vector:

$$\boldsymbol{N} = \boldsymbol{c_1} \, X \, \boldsymbol{c_2}$$

The normal versor will be thus computed through the formula:

$$\boldsymbol{n} = \frac{\boldsymbol{N}}{\|\boldsymbol{N}\|}$$

The norm of the three vector c1,c2 and c3 is also performed. A Cartesian norm is used in this simulator.

The tangential versor is also defined. As it was already stated, the direction of the vector c1 is chosen as the one for the tangential versor, so that the tangential versor is calculated through:

$$\boldsymbol{t} = \frac{\boldsymbol{c_1}}{\|\boldsymbol{c_1}\|}$$

Now, the angle between defined between the three vectors are computed, through:

$$\theta_{ij} = \operatorname{acos}\left(\frac{c_i \, c_j}{\|c_i\|\|c_j\|}\right)$$

Given the angles between the vectors, we define the incremental angles of the element as a three-dimensional column vector, defined as:

$$\Delta\theta = \begin{matrix} 0 \\ \theta_{12} \\ \theta_{12} + \theta_{32} \end{matrix}$$

The need of this vector will soon be clear as it will be explained in the next paragraph.

Finally, we compute the perpendicular to the tangent versor, performing a 3/2 Pi rotation of the tangent.

With this, the geometric definition of the barrier element can be considered as concluded.
As a note, it is remembered that this computations happens just once at the beginning of the simulation.
As a further development of the code, is envisioned the possibility to perform these computations and store them in proper data-sheet in order to re-use them for known geometries and multiple simulations.

Contact Detection

As introduced before, the contact detection happens through a two-step verification:

1) Normal condition
2) Tangential condition

Normal condition

We define the vector distance between a generic Particle P and our barrier element B as:

$$\boldsymbol{d} = r_P - C$$

As a matter of fact, we can project this distance onto the normal direction to the barrier element by scalar product with the normal versor:

$$\boldsymbol{d_n} = \boldsymbol{d} \cdot \boldsymbol{n}$$

The norm of this resulting vector is then performed. Subtracting the particle radius to this value leads to:

$$\delta = \|\boldsymbol{d_n}\| - R$$

If this value is lesser or equal than zero, the tangential check routine starts.

Tangential condition

First, the half-chord defining the r', radius of the particle cut at ("R - δ"), is computed as:

$$r' = R\sin(\mathrm{acos}(\frac{R-\delta}{R}))$$

The angle between the distance vector and the normal versor is computed as:

$$\theta_n = \mathrm{acos}(\frac{\boldsymbol{d} \cdot \boldsymbol{n}}{\|\boldsymbol{d}\|\|\boldsymbol{n}\|})$$

We can thus project the distance onto the barrier element plane through:

$$\boldsymbol{d_{surf}} = \boldsymbol{d}\, sin\theta_n$$

Now, we can compute the angle between the distance and the tangential vector with:

$$\theta_t = \mathrm{acos}(\frac{\boldsymbol{d_{surf}} \cdot \boldsymbol{t}}{\|\boldsymbol{d_{surf}}\|\|\boldsymbol{t}\|})$$

At this stage, however, it is not possible to know from which side of the triangle the particle is approaching, since we can compute, with this formula, just angles included between 0 and PI.
However, we can use the tangent perpendicular versor to sort this problem out.
In fact, it is know that if two vectors are pointed in the opposite direction, their scalar product will be negative. Thus, we perform the scalar product with both the tangent and the tangent perpendicular versors.
Since they are perpendiculars, we can define four different regions.

Based on that, it is now possible to define the physical angle, incrementing going from the tangent versor to the vector c2, between the tangent versor and the distance projected onto the barrier element plane.

Once this is done, a check is performed to see which of the sides of the elements is facing the particle, through the "Incremental angles vector".
If the value of the actual angle is included between the first and the second value of said vector, c1 will be used to check the tangential condition, if it's between the second and the third c2 will be used.

In fact, we can compute the maximum tangential distance possible as:

$$d_{tmax,i} = c_i\, cos\theta_{ti}$$

Finally, it has to be:

$$\left(\|\boldsymbol{d_{surf}}\| - r'\right) \leq d_{tmax,i}$$

If this condition is true, contact dynamics are considered and computed.

Time-integration schemes

At the moment, it is possible to use whichever time-scheme that can be expressed by the formula below:

$$\int_t^{t+\Delta t} \boldsymbol{f}_i \; dt \approx \left[\phi \boldsymbol{f}_i \;\; (t+\Delta t) + (1-\phi)\boldsymbol{f}_i \;\; (t)\right] \Delta t$$

For our validation, three different time-schemes where considered:

1) Φ=0, identifying a Forward Euler integration;
2) Φ=1, identifying a Backward Euler integration;
3) Φ=0.5, identifying a Trapezoidal Rule integration;

Introducing an implicit time-integration scheme, we found ourselves in the need of defining a convergence criterion.
As for my case, I decided to implement a criterion based on solution's successive refination.
With this I mean that, whenever an implicit time-integration scheme is used, the simulator keeps iterating until the successive iteration generates a refination of the solution that is below the imposed tolerance. Meaning we are under the acceptable error.
It is possible to define two different tolerances for positions and for velocities, thus meaning that the computation of the errors is independent.
However, being the positions calculated based on the velocities, the overall precision of the program is decided by the tolerance on them.
Meaning that if a tolerance of 10^{-7} is imposed on positions while the tolerance on velocities is 10^{-3}, our positions will be precise to the 7th decimal case on a position that was calculated with a velocity error on the 4th one, thus implying that our position would be accurate just to the 4th decimal.

We define the errors as:

$$err = \frac{\left\|r_i^{K+1} - r_i^K\right\|}{\left\|r_i^K - r_i^0\right\|} \leq TOL_r$$

$$erv = \frac{\left\|v_i^{K+1} - v_i^K\right\|}{\left\|v_i^K - v_i^0\right\|} \leq TOL_v$$

Where the K indicates the iteration number and 0 is indicating the value at the beginning of the time step.

Solution Validation

In this chapter are collected the validation tests performed in order to identify and detect eventual bugs or formulation errors.
Three different time-integration schemes will be used: Forward Euler, Backward Euler and Trapezoidal.
Different time-steps will be considered, but the total number of steps will be adjusted in order to have the total physical integration time equal to 1 s.

From here on, the values X,Y,Z will refer to the coordinates given the global coordinate system, with the values rx,ry and rz referring to the positions and the values Vx,Vy,Vz will define the velocities corresponding to each coordinate.

Each of the validation will be presented with a *input*, meaning the initial state of our system, and an *expected output*, meaning the solution we would be likely to obtain according to the theoretical solutions.

From now on, we will refer to an unique kind of material for the particle, which characteristics are summarized in the table below:

Material density ρ	2500	Kg/m^3
Young Modulus E	10^7	Pa
Poisson Modulus ν	0,3	

Radius and eventual electric charges will be changing in some simulations.

Also, all these simulations were conducted with imposed tolerances on positions and velocities in the order of 10^{-4}.
Tolerances are kept constant for all the time-steps used, in order to better appreciate the effects of the diminishing time-intervals.

A note to input and output tables: values expressed in parenthesis are intended to be presented in vectorial notation. Thus, a generic vector **v** expressed as (a,b,c) is meant to have value a along the direction X, b in the direction Y and c in the Z direction.

All values will be expressed in SI units, meaning:

Position	m
Velocity	m/s
E (electric field)	Coulomb/ N
B (magnetic field)	Kg/s Coulomb
Charge	Coulomb
Radius	m

Single particle validation

Input

Number of particles	1
Initial position	
Particle 0	(0,0,0)

Initial speed	
Particle 0	(1,0,0)
External fields	
E	(0,0,0)
B	(0,0,0)

Output

Theoretical solution by time integration provides that position is given by:

$$\boldsymbol{r} = \boldsymbol{r_0} + \mathbf{v_0}\,\mathrm{t}$$

For constant speed (as such is our case).
Being the initial speed $\mathbf{v_0}$ **(1,0,0)** and the initial position vector $\boldsymbol{r_0}$ **(0,0,0)** we expect to have after one second the final position vector r(1,0,0).

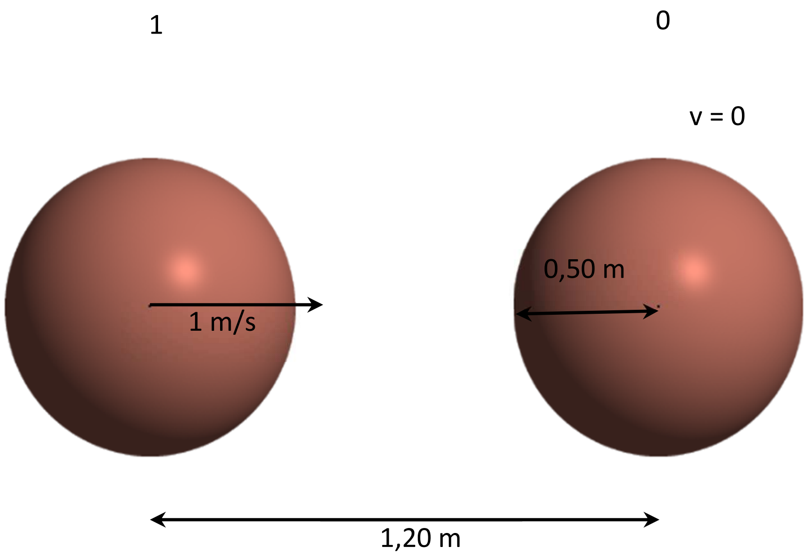
We collected these final values for the simulation:

	e-04	**e-05**	**e-06**
Rx	0.9999	0.99999	0.999999

As we can see, the error is equal to our tolerance, so the simulator is validated.

Two particles contact validation

Input



Number of particles	2
Initial position	
Particle 0	(0,0,0)
Particle 1	(-1.2, 0, 0)
Initial speed	

Particle 0	(0,0,0)
Particle 1	(1,0,0)
External fields	
E	(0,0,0)
B	(0,0,0)
Radius	0,5

Output

The expected outcome of this experiment comes from the theory of elastic contact.
Imagining the shock is completely elastic, the outcome should be decided by this system of equations:

$$\begin{cases} \frac{1}{2}m_1v_{1i}^2 + \frac{1}{2}m_2v_{2i}^2 = \frac{1}{2}m_1v_{1f}^2 + \frac{1}{2}m_2v_{2f}^2 \\ m_1v_{1i} + m_2v_{2i} = m_1v_{1f} + m_2v_{2f} \end{cases}$$

Where the pedix i indicates the initial values and f finals.
As for 1 and 2 we are referring to two generic particles.
First equation comes from the conservation of kinetic energy, while the second from the conservation of quantity of movement.
Being, in our case:

$$\begin{cases} v_i^{(0)} = 0 \\ m^{(0)} = m^{(1)} \\ v_i^{(1)} = 1\,m/s \end{cases}$$

We can write the system as:

$$\begin{cases} {v_i^{(1)}}^2 = {v_f^{(1)}}^2 + {v_f^{(0)}}^2 \\ v_i^{(1)} = v_f^{(1)} + v_f^{(0)} \end{cases}$$

That leads to

$$\begin{cases} v_f^{(1)} = 0 \\ v_f^{(0)} = 1\,m/s \end{cases}$$

Being this our theoretical results, we can compare them with the numerical ones, obtained through a mid-point time integration scheme:

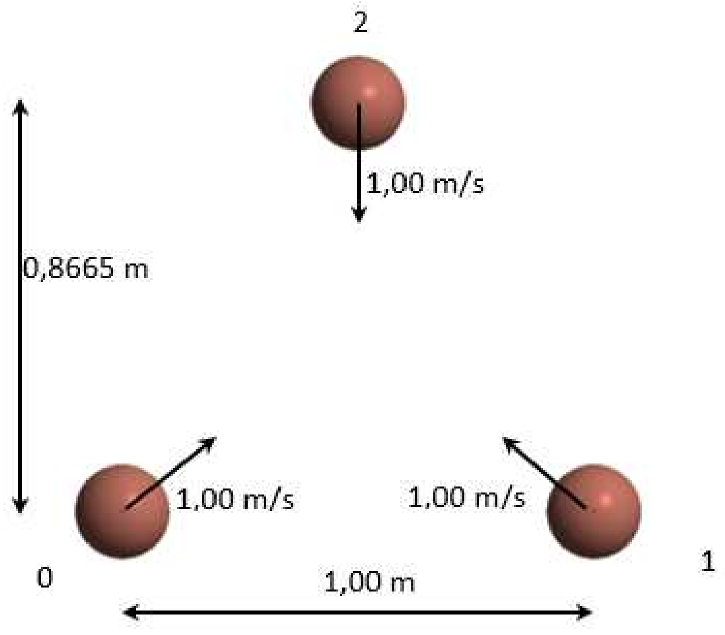
	x Position (m)				x Speed (m/s)			
	Theory	e-04	e-05	e-06	Theory	e-04	e-05	e-06
0	0.8	0.733081	0.73355	0.733597	1	0.999537	0.999954	0.999995
1	-1	-0.93459	-0.9337	-0.93361	0	-0.00139	-0.00014	-1.39E-05

As it is possible to see, numerical results are not perfectly close to the theoretical ones and it is not a matter of approximation (tolerance was set to 10^{-4}).

Problem is that our model requires the two particles to compenetrate in order to generate force. Thus, since the Young modulus in not so high, we need quite a big compenetration in order to generate the necessary force.
If E were bigger, the compenetration would be smaller and the result closer to the theoretical one, meant for infinitely rigid bodies.

Three particles contact validation

Input



Number of particles	3
Initial position	
Particle 0	(-0.5,0,0)
Particle 1	(0.5, 0, 0)
Particle 2	(0,0.866025,0)
Initial speed	
Particle 0	(0.866025,0.5,0)
Particle 1	(-0.866025,0.5,0)
Particle 2	(0, -1, 0)
External fields	
E	(0,0,0)
B	(0,0,0)
Radius	0,1

Output

This contact validation is stronger and more restrictive than the previous one.
Mathematical and physics principles applying to this one are the same as the previous, thus allowing us to omit the mathematical resolution of the problem.
The comparative table will be thus straightforwardly presented:

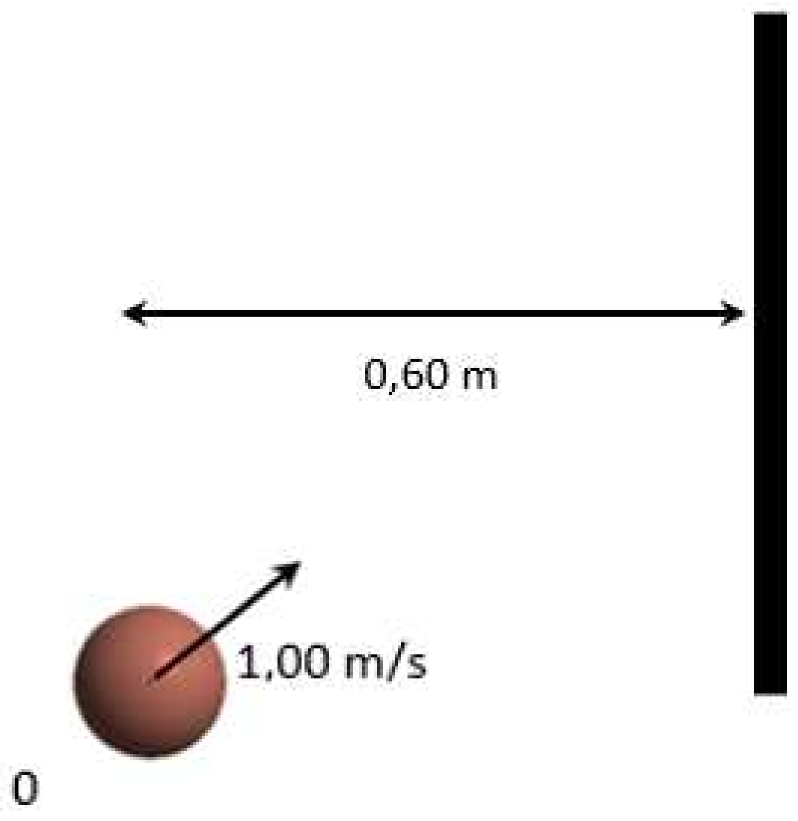
		Theory	**e-01**	**e-02**	**e-03**	**e-04**	**e-05**	**e-06**
0	vx	-0.866025	0.866025	0.866025	-0.96176	-0.87575	-0.86697	-0.8661
	vy	0.5	0.5	0.5	-0.54362	-0.50439	-0.50045	-0.50006
1	vx	0.866025	-0.86601	-0.86601	0.961783	0.875765	0.866989	0.866115
	vy	-0.5	0.5	0.5	-0.52096	-0.50219	-0.50023	-0.50004
2	vx	0	0	0	0.010962	0.001086	0.000109	1.09E-05
	vy	1	-1	-1	1.06952	1.00659	1.00068	1.0001

We can see that these results are much, much closer to the theoretical ones: this happens because the chosen radius is smaller, meaning volume, meaning mass, thus requiring less force, meaning compenetration, to move the particle.

Again, our simulator results validated.

Single barrier validation

Input



Number of particles	1
Initial position	
Particle 0	(-1.2,0,-0.6)
Initial speed	
Particle 0	(0.5, 0 , 0.5)
External fields	
E	(0,0,0)
B	(0,0,0)
Radius	0,1
Number of barriers	1
Plane of impact	(-0.7, 0, 0)

Output

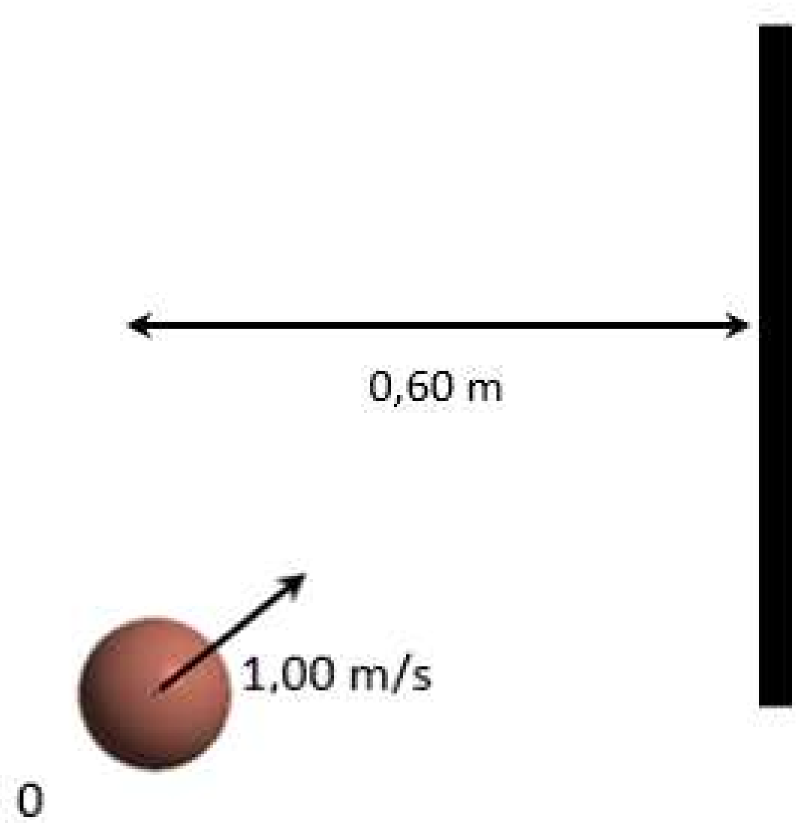
	e-04	**e-05**	**e-06**
Vx	-0.4999	-0.49967	0.499999
Vy	0.5000	0.5000	0.5000

Again, we can see that our simulator is validated, since the particle inverted its speed just in the direction x, which was normal to the barrier.

Adjacent barriers validation

Input

This simulation reproduces the one before, with the difference that the barrier is no longer defined by only one triangle, but by two, positioned in a way that the point of impact of the particle coincides with the division of the two barriers, thus activating both.
This test is run in order to assess if in case of contact with multiple barriers the particles shows abnormal behaviour.



Number of particles	1
Initial position	
Particle 0	(-1.2,0,-0.6)
Initial speed	
Particle 0	(0.5, 0 , 0.5)
External fields	
E	(0,0,0)
B	(0,0,0)
Radius	0,1
Number of barriers	2
Plane of impact	(-0.7, 0, 0)

Output

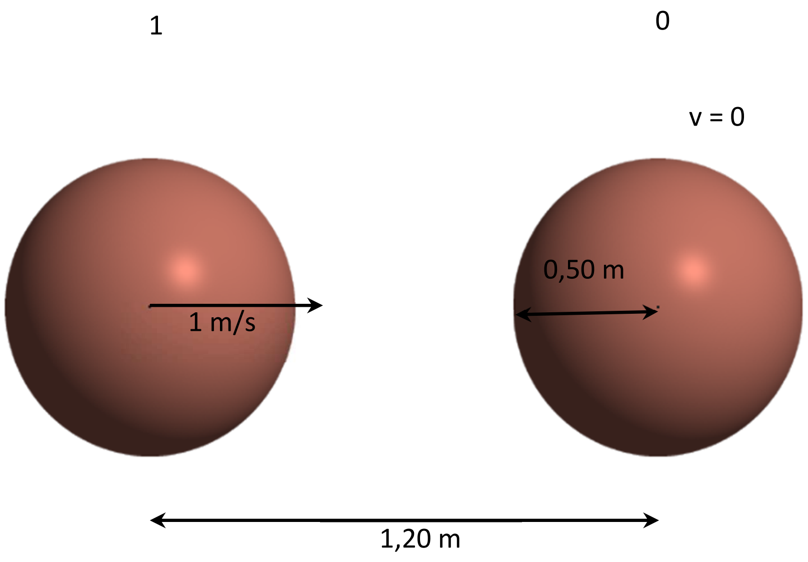
	e-04	e-05	e-06
Vx	-0.4999	-0.49967	0.499999
Vy	0.5000	0.5000	0.5000

Output was identical to the previous case, thus we consider the simulator, again, validated.

Hole validation

Input

This test is run on the same particles seen for the two particles contact validation.
However, this time we set 4 triangle barriers defining a square hole of 1.02 m of side, positioned in a way that it would allow the passage of the particle.
This test was run in order to assess if the particles were able to correctly identify the precence of spacially limited barriers.



Number of particles	2
Initial position	
Particle 0	(0,0,0)
Particle 1	(-1.2, 0, 0)
Initial speed	
Particle 0	(0,0,0)
Particle 1	(1,0,0)
External fields	
E	(0,0,0)
B	(0,0,0)
Radius	0,5
Number of barriers	4

Plane of impact	(-0.6, 0, 0)

Output

The results obtained were identical as the ones obtained withouth barrier, thus validating the simulator also in this situation.

	x Position (m)				x Speed (m/s)			
	Theory	e-04	e-05	e-06	Theory	e-04	e-05	e-06
0	0.8	0.733081	0.73355	0.733597	1	0.999537	0.999954	0.999995
1	-1	-0.93459	-0.9337	-0.93361	0	-0.00139	-0.00014	-1.39E-05

Electromagnetic validation

Once done the mechanicals validations, we will now perform some validations over the electromagnetic forces. We will present three cases, one where the electric field in the z direction is different than 0, one that the magnetic field in the direction z is different than 0 and one where both (electric and magnetic fields in z directions are non-nulls) are applied onto a single particle.

Input data will be the same in all the three cases for the particle, changing just the external environmental variables.

Number of particles	1
Initial position	
Particle 0	(0,0,0)
Initial speed	
Particle 0	(1,0,0)

All simulations were run with four different time-steps: 10^{-3}, 10^{-4}, 10^{-5} and 10^{-6}.
Tolerances were constant in all the cases and set to 10^{-4}.

As another general remark, since these comparisons had to be made manually, it was necessary to choose a time interval in order to pick and record the values of the various simulations. This time interval was chosen to be 0,05 s.
For this reason it could appear that the solutions are not “smooth enough”, but this is just a direct consequence of this choice.
Were the time-interval smaller, we would get a much smoother representation of the solution.
But it was not the focus of this tests, which were to assess the accuracy of the simulator towards electromagnetic problems.

Finally, the solving theoretical system will be introduced.
Is important to note that Ez and Bz are decoupled, meaning that the general case solution is valid for the other ones, nulling the non-interesting values.

We will thus have:

$$\begin{cases} v_x(t) = v_0\cos(\omega t) \\ v_y(t) = -v_0\sin(\omega t) \\ v_z(t) = \dfrac{q}{m}E_z^{ext}t \end{cases}$$

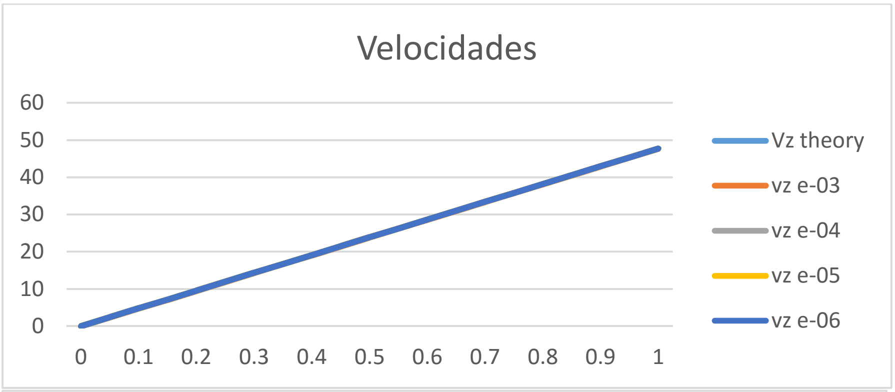
$$\begin{cases} r_x(t) = \dfrac{v_0}{\omega}\sin(\omega t) \\ r_y(t) = \dfrac{v_0}{\omega}(\cos(\omega t) - 1) \\ r_z(t) = \dfrac{q}{2m}E_z^{ext}t^2 \end{cases}$$

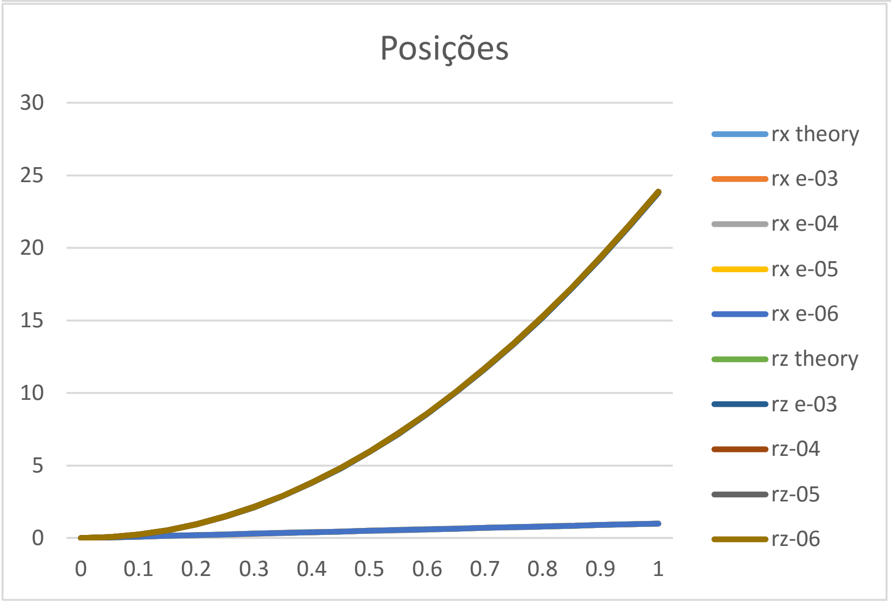
Where ω is the cyclotron frequency defined by:

$$\omega = \frac{qB_z^{ext}}{m}$$

Ez = 500 N/ Coulomb , Bz=0

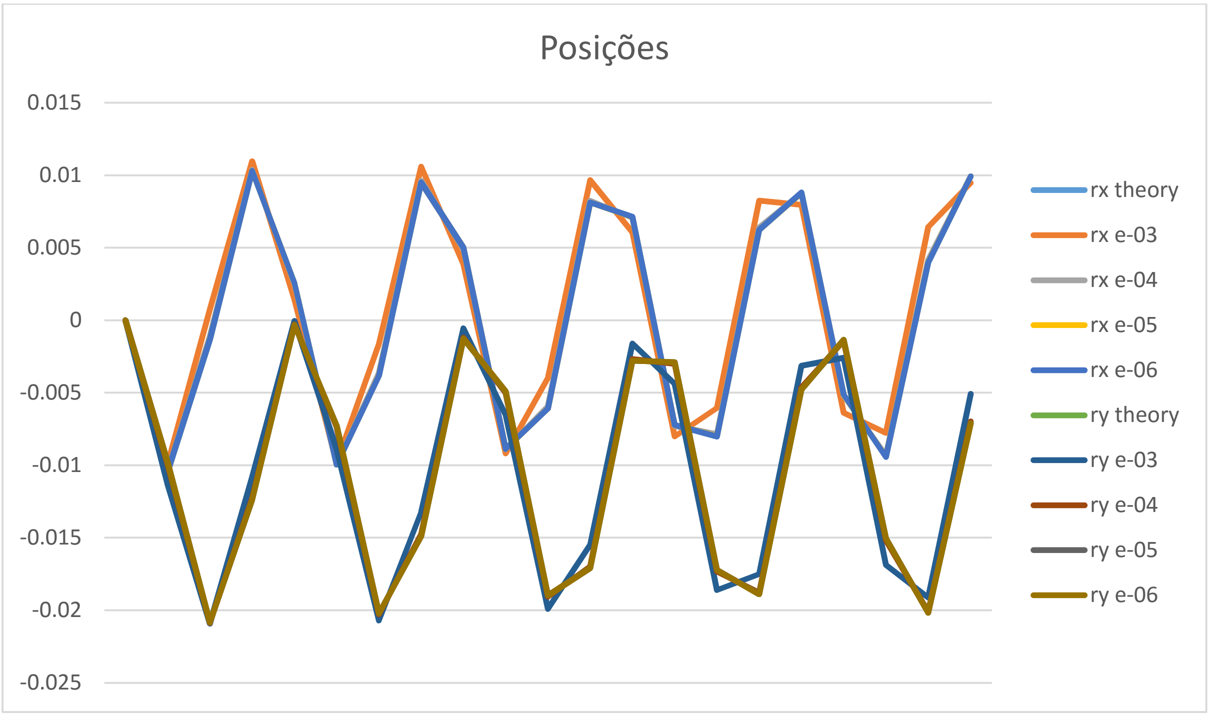
Here are represented the graphs resuming the values computed on speeds and positions.

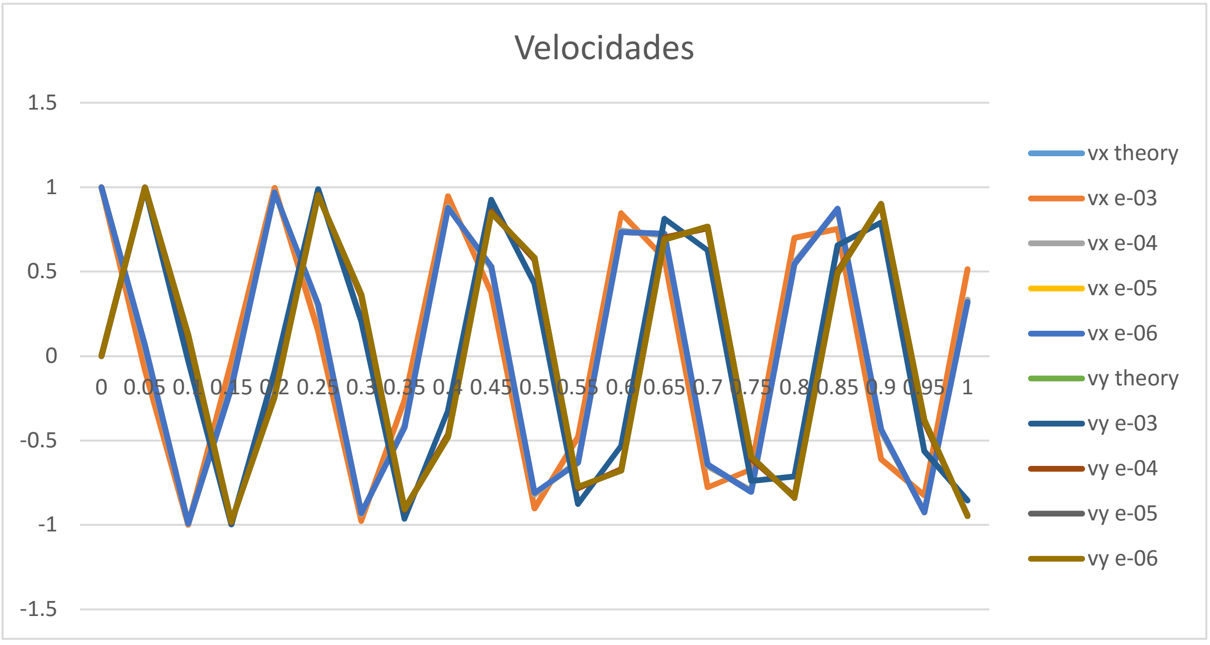




As we can see, the solutions generated by all time-steps converge without problem to the theoretical solution: all the values are superposing with the theoretical values.

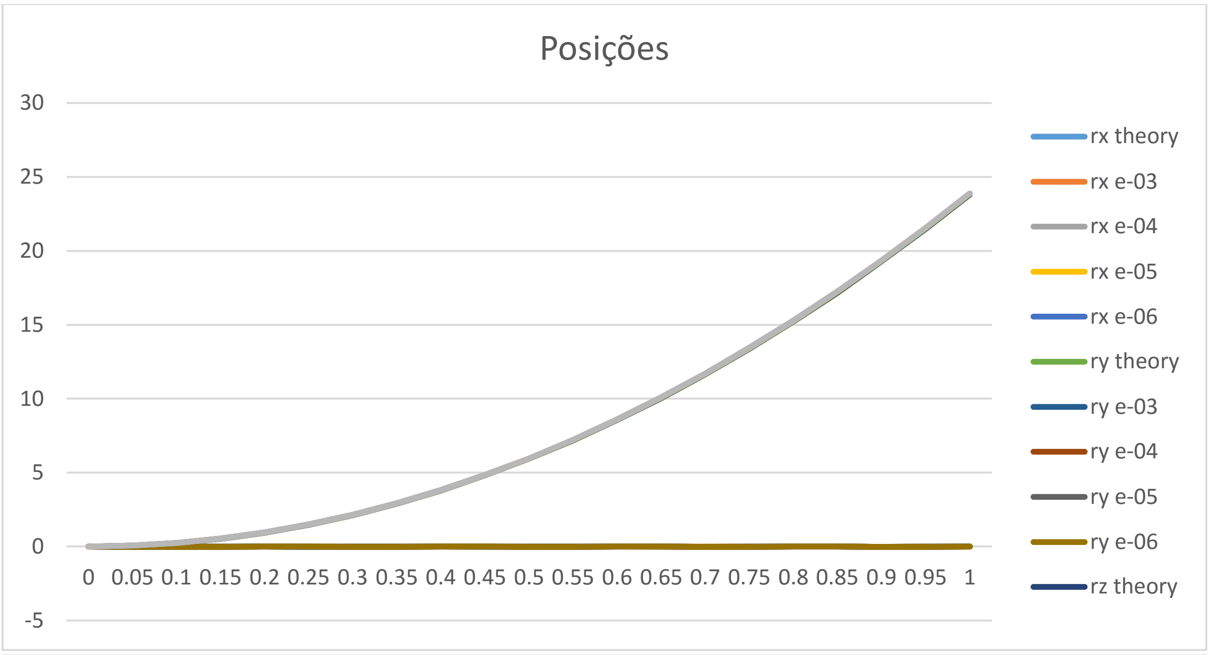
Ez = 0 , Bz=1000 Kg/s Coulomb

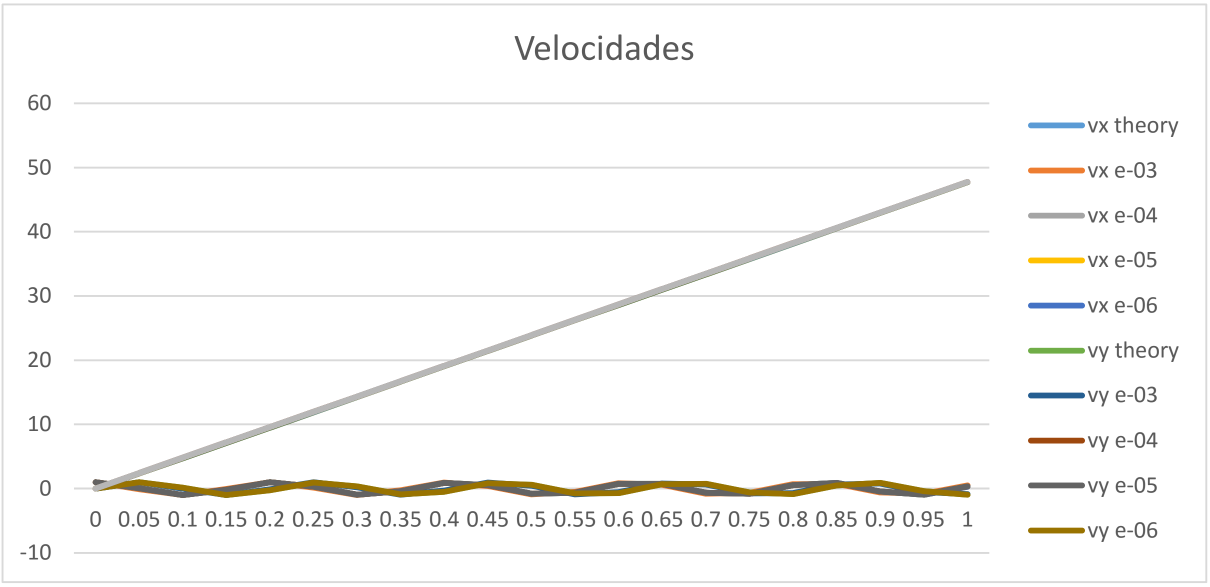




Here, as we can see, we can see that for the time-step 10^{-3} we have some inaccuracies.
As a matter of fact, this time-step is the only one that is not superposing with the values computed theoretically.

Ez = 500 N/ Coulomb , Bz=1000 Kg/s Coulomb





Here, again, we can see how the numerical results converge pretty smoothly to the theoretical solutions.

Final considerations

Some statistical evaluations were run on the various values obtained and average and standar deviation of errors with the theoretical solution were computed.

These datas are summarized in the table below.
We define Mag R as the magnitude of the error towards the position R and as Mag V the magnitude of the error towards the theoretical velocity V.

Caso 1 is referring to the first case here analized (Ez ≠ 0, Bz = 0), Caso 2 is referring to the second (Ez = 0. Bz ≠0) and finally Caso 3 to the third (Ez≠0 and Bz≠0).
The errors were computed time-interval after time-interval, having an average of the errors and a standard deviation for each case and for each time-step.

All the three cases where then analized toghether, getting an average of the averages and an average of the average standard deviations.

Results are summarized in the table below:

		Caso 1		Caso 2		Caso 3		Averages	
		Mag R	Mag V	Mag R	Mag V	Mag R	Mag V	Mag R	Mag V
Average	e-03	0.037569	0.071626	0.001949	0.180823	0.037629	0.195117	0.025716	0.149189
	e-04	0.003763	0.007164	0.00016	0.014708	0.003767	0.01636	0.002564	0.012744
	e-05	0.000399	0.000745	1.62E-05	0.001487	0.0004	0.001663	0.000272	0.001298
	e-06	3.86E-05	7.6E-05	1.59E-06	0.000145	3.86E-05	0.000166	2.63E-05	0.000129
Sigma	e-03	0.004614	5.72E-06	8.84E-05	0.00465	0.004606	0.004376	0.003102	0.003011
	e-04	0.000461	5.83E-06	7.54E-06	4.72E-05	0.000461	4.31E-05	0.00031	3.2E-05
	e-05	6.05E-05	2.32E-05	8.49E-07	4.69E-05	6.04E-05	5.21E-05	4.06E-05	4.07E-05
	e-06	6.44E-06	5.7E-06	7.58E-08	2.88E-07	6.43E-06	2.66E-06	4.32E-06	2.88E-06

As we can see, e-03, indicating time-step 10^{-3} generates the worsts results (as expected) both in term of averages and SD.
As an important remark, these values are not adimensional, but are actually describing error in the SI.
Meaning, for Mag R all errors are in m and for Mag V all average errors are in m/s.
This means that for a 1 second simulation and the forces before identified, 10^{-3} gives an average error in position of 2,5 cm and of 14,91 cm/s in velocities.
We define this, by no means, accettable.

As we can see, clearly e-05 and e-06 are time-steps which are soundly convergent: the differences they have with the theoretical solution are dominated by the chosen tolerance (10^{-4}).

e-04, corresponding to a 10^{-4} time-step, is an intermediate case.
Depending on the dimension of the problem and the accuracy required, it could or could not be a sound decision.
However, we should remember that electromagnetic forces are not the only active ones in our final case studies.

Contact dynamics, in fact, require a minum time-step in order for the contact to be detected. It is defined by the contact time that can be easily computed given the material characteristics and the relative velocities of the colliding bodies.

The minor of these time-steps will thus define the time-step of the whole simulation.

Considerations on computation time

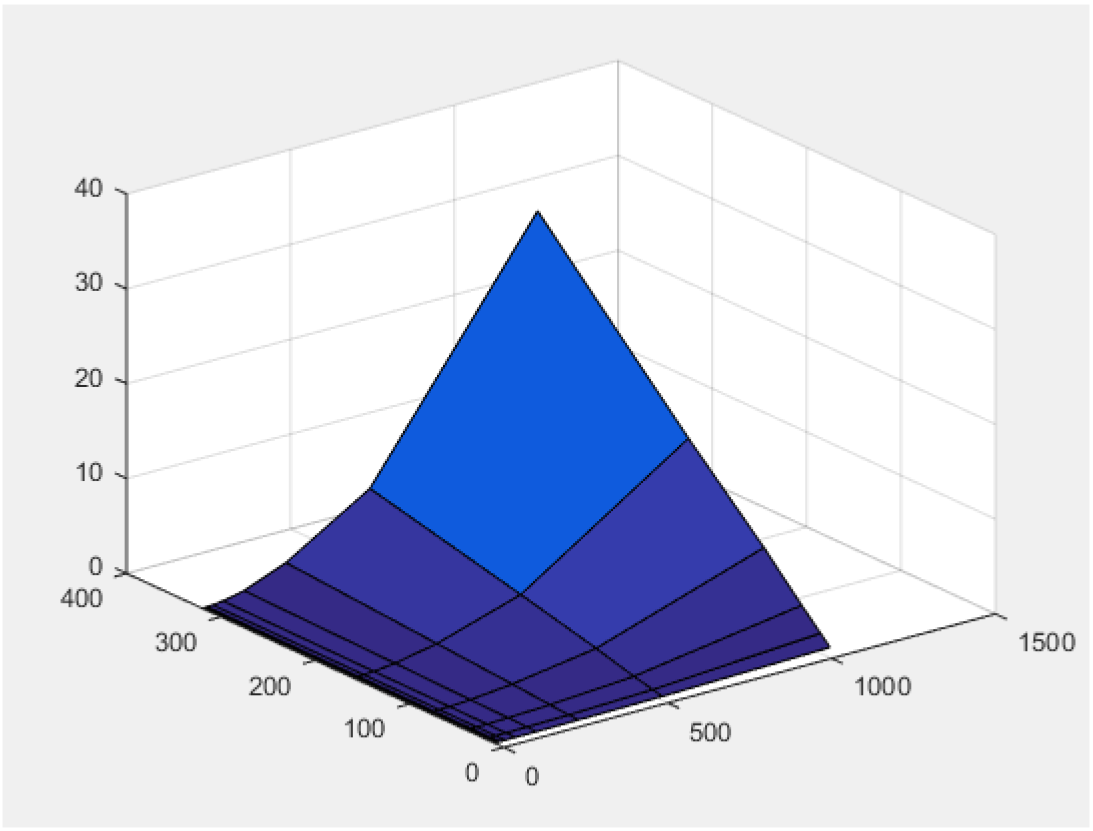
Up until now, we presented just the results obtained with the mid-point time integration.
However, these tests were carried with the other time-integration schemes implemented, validating all the three of them.
It was decided to omit these validations for a matter of redundancy.

However, another analisys was run, in terms of computation times.

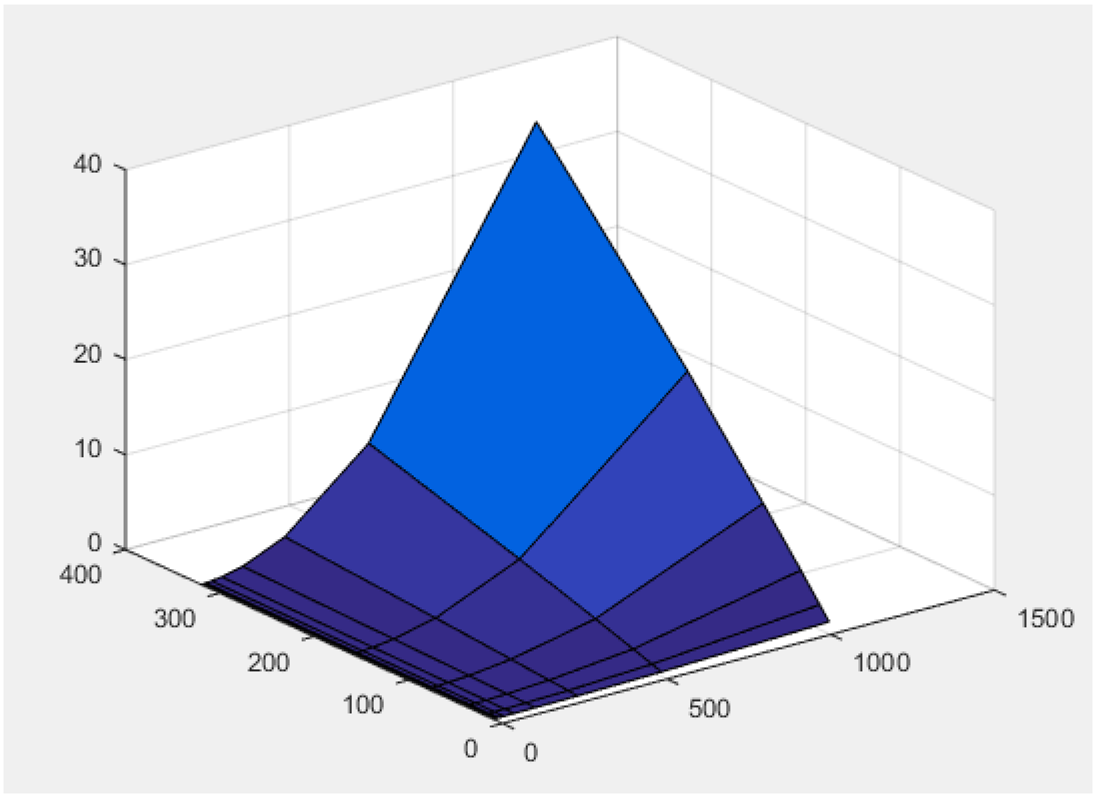
In fact, identical simulations, increasing in order of total integration time and number of particles were run in the three different time-schemes, in order to evaluate the relative speed of the different integrations.

The presented graphs will present a surface which elevation values are the total computation time (in minutes), while the independent variables will be the number of particles (from 1 to 1000) and the time-steps (from 1 to 320).

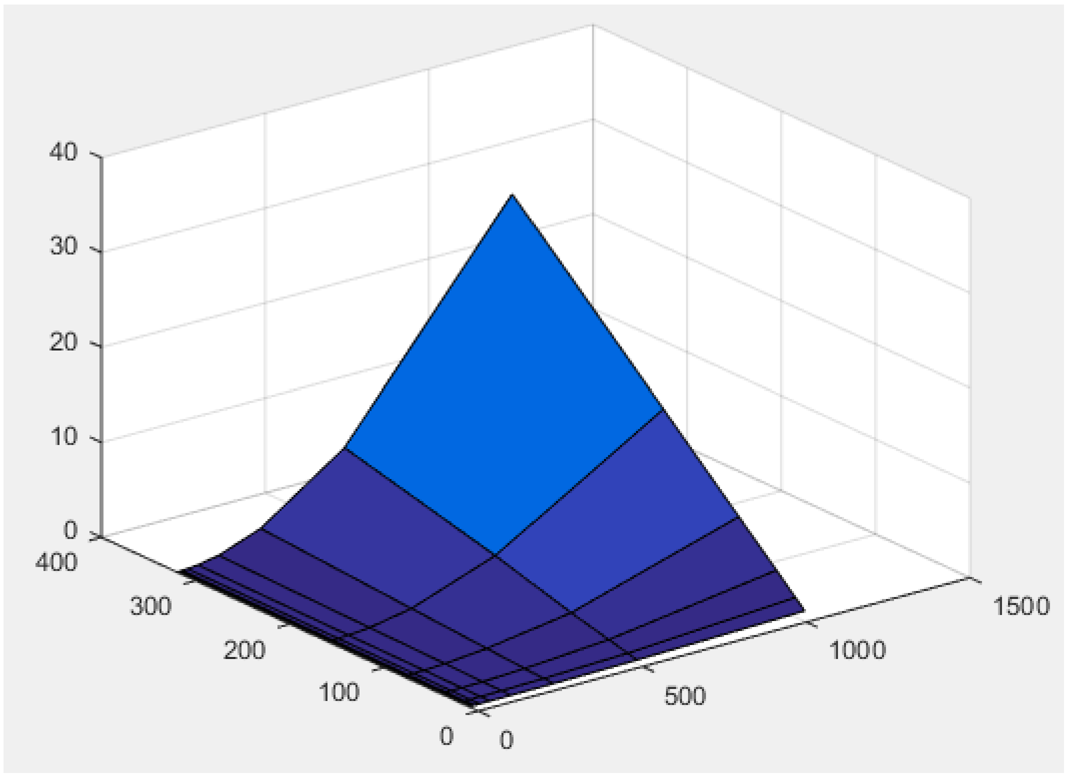
This were the result collected for Explicit Euler method:



This were the results collected for the Implicit Euler method:



And finally, the results for mid-point implicit integration scheme (trapezoidal rule).



As we can see, the latter is by far faster than the completely implicit (Backward Euler) and give a much better accuracy and precision (by definition) of the Forward Euler (explicit).
This is why we primarily focused on the trapezoidal time integration.

Another interesting observation is collectable from these graphs.

We can in fact notice that the total computational time increases quadratically in time increasing the number of particles, while it increase linearly increasing the total integration time.

This happens due to the fact that the more are the particles, the more contacts we will have, causing the simulator to do more computations to get to convergence.

Case studies

Finally, some case studies will be presented. Mainly, we will have a case of 500 particles which will be shocking with a wall, not subjected to any electromagnetic field.

Then, we will have a set of 150 particles that will be sent towards the barrier and will be subjected to electromagnetic fields.

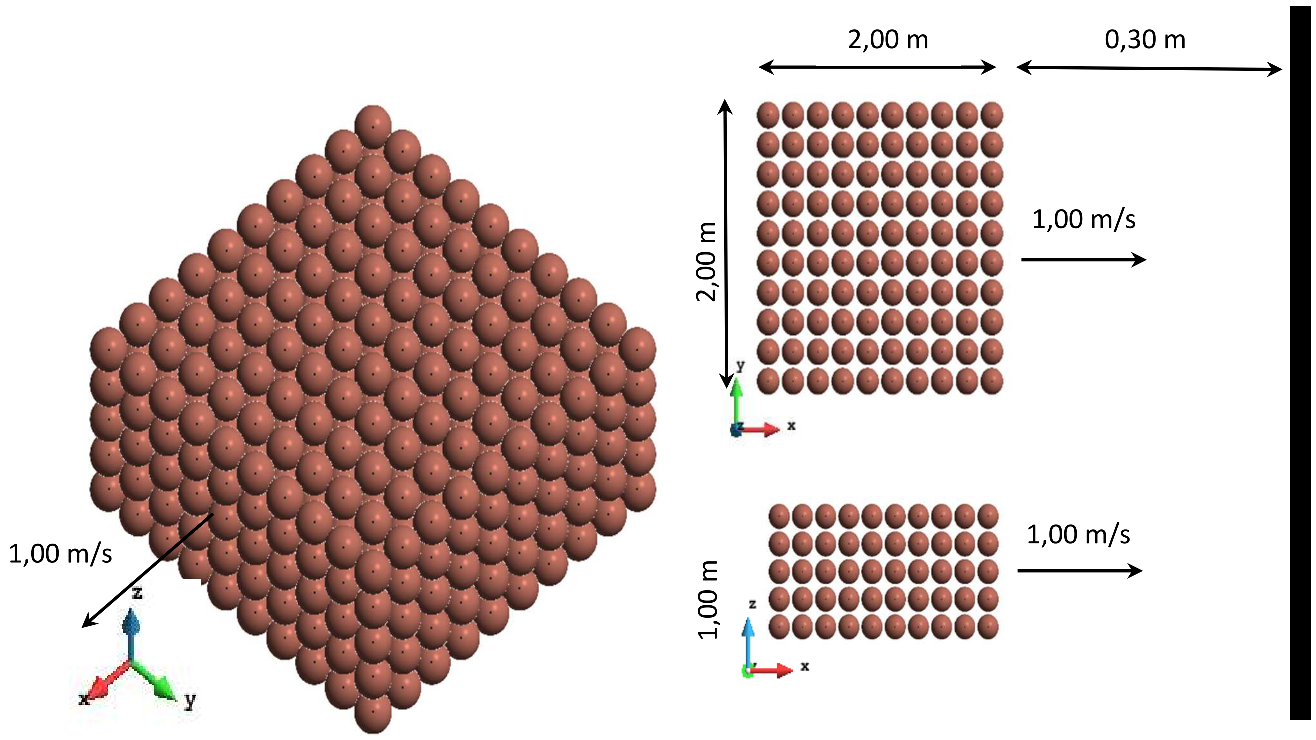
As a remark, all the particles in the simulations are identical:

Material density ρ	2500	Kg/m^3
Young Modulus E	10^7	Pa
Poisson Modulus ν	0,3	
Radius	0,1	m
Charge	± 1	Coulomb

The particles will have alternated charges distributed randomly but respecting a 50/50 proportion.

500 particle barrier shock

Input



The problem set-up is exemplified in the Picture: we have 500 identical particles evenly distributed in a block of 2*2*1 m dimensions, all of them having an initial velocity of (1,0,0).

Output

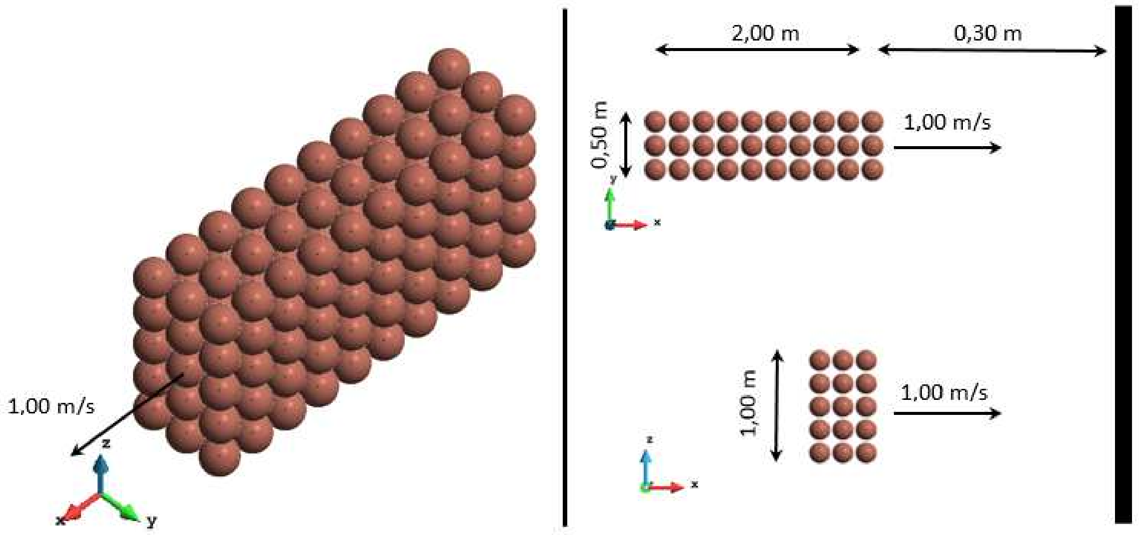
As we can see, the simulator detects the contact with the barrier and the resulting shock-wave that propagates from particle to particle thanks to the contacts particle-particle.

It is also clearly visible the effect of the near-field interaction force in the block: in fact, as it moves towards the barrier we can see that there is the tendencies of the extremes of the block to open-up.

This happens because the particles at the boundaries of the block are receiving non-null total forces from the near-field interactions. The one in the center, on the other hand, since are completely immersed in the block, are in the center of a perfectly balanced system of forces, thus appearing unaffected by the presence of the other particles.

150 particles barrier shock under electromagnetic fields

Input



Conditions are similar to the case studied before, but in this case we have less particles (150) distributed in a smaller block (2,00*0,50*1,00 m).
Initial velocity Is the same (1,0,0) and the same conditions apply for charge.

External field condition 1

External fields	
E	(15,15,15)
B	(0,0,0)

The electric field applied accelerates the particles, creating thus more shocks. In comparison with the example with no electric field applied, we can see that the entropy is much higher, meaning a more disordered system. In this case, in fact, detecting the impact shock-wave is not as immediate as it was in the previous case.

External field condition 2

External fields	
E	(0,0,0)
B	(0,500,1000)

The magnetic field, in this case, traps the particles, which now are cycling around themselves. Since the particles are well ordered with enough space between each other, even if we have opposed rotations, we do not have any contacts: the particles stay in that steady state for all the simulation.

External field condition 3

External fields	
E	(500,250,0)
B	(0,500,1000)

The input of the extremely strong electric fields, in this case, generates, after a while, the “explosion” of the solution.
This happens because the simulator does not have, yet, a time-step adapter implemented.
A time-step adapter would allow the simulator to reduce the integration time-step if speeds are approaching critical values, thus avoiding the numerical instability of the solution.

Conclusions and future steps

Working on this simulator was an utterly magnificent experience.

I discovered myself intrigued and thrilled by the world of computational mechanics. In fact, creating a working simulator from scratch it is not just a matter of actually "coding", but is a matter of deeply understanding the dynamics of the problem, with a deep and complete understanding of the physical model.
And having the chance to create something really mine, meaning not thought in the same way by anybody else, like when I thought, conceived, modelled and programmed my barrier element, was the apex of my satisfaction during this work.
And after countless hours of continuous debugging, getting the feeling that the things work is something like I never experienced before.

It wasn't easy, but now it works. Every line of code I wrote was a forward step not just as me as an Engineer, but also as a man.
After these months, I could say I found myself a passion worth working for.

And that's what I'll do.

Some future steps are already planned and designed:

- Contact search optimization through binning or verlet lists;
- Adaptative time-step frames;
- Friction forces implementation;
- Angular moments implementation;
- Impulse contact mechanics;

And these are the planned developments for the near future.

Some other "dreams" and ideas are present in the back of my mind, but I will keep focusing on every small objective and implementation, getting my code more and more complex.

I saw a future, I am creating it.

Bibliography

[1] 2012, Dynamics of Charged Particulate Systems: Modeling, Theory and Computation, Tarek I. Zohdi, Springer

[2] 2015, A description of rotations for DEM models of particle systems, Eduardo M. B. Campello

[3] 2014, A computational framework for simulation of the delivery of substances into cells, Eduardo M. B. Campello and Tarek I. Zohdi

[4] 2004, The Combined Finite-Discrete Element Method, Ante Munjiza, John Wiley & Sons, Ltd

[5] 2013, The C++ Programming Language, 4th Edition, Bjarne Stroustrup, Addison—Wesley

[6] 2015, Mathlab Primer, 25th Edition, The MathWorks, Inc.

[7] 2015, Mathlab Programming Fundamentals, 25th Edition, The MathWorks, Inc.

[8] 1984, Contact Mechanics, K.L. Johnson, Cambridge University Press

Appendixes

Code

Main.cpp

```
#include <iostream>
#include <sys/time.h>
#include <vector>
#include <unistd.h>
#include <math.h>
#include <fstream>
#include "Particle.h"
#include "Space.h"
#include "Barrier.h"
#include <string>
#include <sstream>
#include "Fields.h"

using namespace std;

// ---------------------------------------------------------------------
// ---------------------------------------------------------------------

void GiDheaderbarr(ofstream& output);
void initializertxt(vector<Particle>* particles, Space& s, string csvFile);
void initializer(vector<Particle>* particles, Space& s, int num);
void initializerCSV(vector<Particle>* particles, Space& s, string csvFile);
void initializerCSVbarrier(Space& space, string csvFile);
int numcontacts(vector<Particle>& particles);
void reset(vector<Particle>& particles);
void computeNf(vector<Particle>& particles);
void interact(vector<Particle>& particles);
void pre_iteration(vector<Particle>& particles);
void move(vector<Particle>& particles);
void descrition(vector<Particle>& particles);
void reset(vector<Particle>& particles);
void csvParticles(vector<Particle>& particles, int iteration, int caso, int tstep);
void csvTime(long totaltime, int caso, int tstep);

void printParticle(vector<Particle>& particles, int t, int step, int l);
void printBarrier(Space space, int t, int step);

void GiDheader(ofstream& output);
void GiDcontentpre(ofstream& output, int step, int* counterprint, float timer);
void printGiD (ofstream& output, int nparticles, int step, vector<Particle> particles, int time, int*
counterprint, float timer);
void printmeshGiD (ofstream& output, vector<Particle> particles, int nparticles, Space space, int*
numer);
```

```
void mesher (ofstream& output, int i, int j, int k, int nummesh);

void printmshGiDbarr(ofstream& output, Space space, int numer);
void printGiDbarr (ofstream& output, Space space, int step, int time, int* counterprint,float timer,
int numer);
void GiDcontentprebarr(ofstream& output, int step, int* counterprint, float timer);

// ----------------------------------------------------------------------
// ----------------------------------------------------------------------

int main(){

        srand(unsigned(time(NULL)));

   ofstream results;
   ofstream resultsbarr;
   results.open("log/GiD/resultsdata.post.res",ios::out);
   resultsbarr.open("log/GiD/resultsdatabarr.post.res",ios::out);
        GiDheader(results);
        GiDheaderbarr(resultsbarr);
        int counterprint = 1;
        int counterprint2 = 1;

        float physicaltime = 1.000;
   int totaltime=physicaltime/DELTA_T;
   float timer=0.001;
   int time = timer/DELTA_T;

   int numer = 1;

   Space space;

/*
   initializerCSVbarrier(space, "datazero/barrieraTCC.csv");

   for (int l=0;l<space.numbarr();l++){
      space.barriers[l].normal_def();
   }
*/
   vector<Particle> particles;

        int l=150;
        int nparticles=l;


   //initializertxt(&particles, space, "datazero/input.txt");
   initializerCSV(&particles, space, "datazero/part_inv04.csv");



   ofstream mesh;
```

```
    printmeshGiD (mesh, particles, nparticles, space, &numer);
    ofstream meshbarr;
    numer = 0;
    printmshGiDbarr(meshbarr,space,numer);

    int step=totaltime;
    for(int t=1; t<step; t++){

        reset(particles);
        //   computeNf(particles);
        pre_iteration(particles);
        move(particles);

                        /*
                        if((counterprint*time - 1 == t) || (t == step-1)){
        printParticle(particles, t, step, l);
          //printBarrier(space, t, step);
        }
        */


        printGiD (results, nparticles, t, particles, time, &counterprint, timer);
        printGiDbarr (resultsbarr, space,t,time, &counterprint2,timer,numer);

        cout << t << endl;


    }

            results.close();
            resultsbarr.close();


    return 0;
}



void computeNf(vector<Particle>& particles)
{
    for(int i=0; i<particles.size(); i++){
        for(int j=0; j<particles.size(); j++){
            if( i!=j ){

                particles[i].computeNf(particles[j]);

            }
        }
    }
}
```

```
void interact(vector<Particle>& particles)
{
   for(int i=0; i<particles.size(); i++){
      for(int j=0; j<particles.size(); j++){
         if( i!=j ){

            particles[i].interact(particles[j]);

         }
      }
   }
}

void move(vector<Particle>& particles)
{
   for(int i=0; i<particles.size(); i++){

      particles[i].move(particles);
   }
}

void descrition(vector<Particle>& particles)
{
   for(int i=0; i<particles.size(); i++){
      particles[i].description();
   }
}

void reset(vector<Particle>& particles)
{
   for(int i=0; i<particles.size(); i++){
      particles[i].reset();
   }
}

void pre_iteration(vector<Particle>& particles)
{
   for(int i=0; i<particles.size(); i++){
      particles[i].pre_iteration();
   }
}


void initializer(vector<Particle>* particles, Space& s, int num)
{
   for(int i=0; i<num; i++){

      Particle p(s);
      p.identifier = i;

      float r_x = (float)(rand()%1000 - 500) / 1000;
```

```
        float r_y = (float)(rand()%200 - 100) / 1000;
        float r_z = (float)(rand()%1000 - 500) / 1000;
        p.r = {r_x, r_y, r_z};
/*
        float v_x = (float)(rand()%4000 - 2000) / 100;
        float v_y = (float)(rand()%4000 - 2000) / 100;
        float v_z = (float)(rand()%4000 - 2000) / 100;
        p.v = {v_x, v_y, v_z};
*/
                    float v_x = 30.00000;
        float v_y = 0;
        float v_z = 0;
        p.v = {v_x, v_y, v_z};

        p.N = {0, 0, 0};
        p.rho = 2500;

        int q = (int)(rand()%5);
        p.q = pow(-1,q);

        p.R = 0.1;
        p.E = 100000000;
        p.poisson = 0.3;
        particles->push_back(p);

    }
}

// ---------------------------------------------------------------------
// ---------------------------------------------------------------------

void initializerCSV(vector<Particle>* particles, Space& s, string csvFile)
{
    ifstream csv;
    string line;
    csv.open(csvFile,ios::in);
    if( csv.is_open() ){
        while(!csv.eof()){
            getline(csv,line);
            if(line != ""){
                //cout << line << endl;

                std::istringstream ss(line);
                std::string token;
                std::vector<string> tokens;

                while(std::getline(ss, token, ',')) {
                    tokens.push_back(token);
                }

                Particle p(s);
                p.identifier = atoi(tokens[0].c_str());
                float r_x = stof(tokens[1]);
                float r_y = stof(tokens[2]);
                float r_z = stof(tokens[3]);
                p.r = {r_x, r_y, r_z};

                float v_x = stof(tokens[4]);
                float v_y = stof(tokens[5]);
                float v_z = stof(tokens[6]);
                p.v = {v_x, v_y, v_z};

                p.N = {0, 0, 0};
                p.rho = 2500;

                float q = stof(tokens[8]);
                p.q = q;

                p.R = stof(tokens[9]);
                p.E = 10000000;
                p.poisson = 0.3;
                p.a={0,0,0};
                particles->push_back(p);

            }
        }
        csv.close();
    }
}

int numcontacts(vector<Particle>& particles)
{
    for(int i=0; i<particles.size(); i++){
        for(int j=0; j<particles.size(); j++){
            if( i!=j ){

                particles[i].numcontacts(particles[j]);

            }
        }
    }
    return 1;
}

void csvParticles(vector<Particle>& particles, int iteration, int caso, int tstep)
{
    ofstream output;

output.open(("datazero/particles_"+to_string(caso)+"_iteration_"+to_string(iteration)+"_over_"+to
_string(tstep)+".csv").c_str(),ios::out);
    if(output.is_open()){
        for(int i=0;i<particles.size();i++)
```

```
            particles[i].csv(output);
        output.close();
    }
}

void csvTime(long totaltime, int caso, int tstep)
{
    ofstream output;

output.open(("zz_Total_Ingretation_time_"+to_string(caso)+"_iteration_"+to_string(tstep)+".csv").c
_str(),ios::out);
    output << totaltime << endl;
    output.close();

}



void initializerCSVbarrier(Space& space, string csvFile)
{
    ifstream csv;
    string line;
    csv.open(csvFile,ios::in);
    if( csv.is_open() ){
        while(!csv.eof()){
            getline(csv,line);
            if(line != ""){


                std::istringstream ss(line);
                std::string token;
                std::vector<string> tokens;

                while(std::getline(ss, token, ',')) {
                    tokens.push_back(token);
                }

                Barrier b;
                b.identifier = atoi(tokens[0].c_str());
                float r_xA = stof(tokens[1]);
                float r_yA = stof(tokens[2]);
                float r_zA = stof(tokens[3]);

                float r_xB = stof(tokens[4]);
                float r_yB = stof(tokens[5]);
                float r_zB = stof(tokens[6]);


                float r_xC = stof(tokens[7]);
                float r_yC = stof(tokens[8]);
                float r_zC = stof(tokens[9]);
            b.points_coord = {r_xA, r_yA, r_zA, r_xB, r_yB, r_zB,r_xC, r_yC, r_zC};



            space.barriers.push_back(b);
          }
        }
        csv.close();
    }

}

// ------------------------------------------------------------------------
// ------------------------------------------------------------------------

void printParticle(vector<Particle>& particles, int t, int step, int l)
{
    ofstream outputCSV;
    //if(t>step-2){

outputCSV.open("log/particles_"+to_string(l)+"_iteration_"+to_string(t)+"_over_"+to_string(step)+".
csv", ios::out);
        if( outputCSV.is_open() ){

            for(int ci = 0; ci < particles.size(); ci++){
                particles[ci].csv(outputCSV);
            }

            outputCSV.close();
        }
    //}
}

/*
void printBarrier(Space space, int t, int step)
{
    ofstream outputCSV2;

    outputCSV2.open("log/barriers/barriers_iteration_"+to_string(t)+"_over_"+to_string(step)+".csv",
ios::out);
    if( outputCSV2.is_open() ){

        for(int ci = 0; ci < space.numbarr(); ci++){
            space.barriers[ci].csvBARR(outputCSV2);
        }

        outputCSV2.close();
    }
}
```

```
*/

//=============================================================================

void GiDheader(ofstream& output){

	output << "GiD Post Results File 1.0 \n \n \n" << endl;
}

void GiDheaderbarr(ofstream& output){

	output << "GiD Post Results File 1.0 \n \n \n" << endl;
	output << "GaussPoints \"Triangle\" Elemtype Triangle" <<endl;
	output << "  Number of Gauss Points: 1" << endl;
	output << "  Natural coordinates: internal" << endl;
	output << "end gausspoints" << endl;
}

void GiDcontentpre(ofstream& output, int step, int* counterprint, float timer){

	output <<
"#==============================================================================="<<endl;
	output << "#Step 1 Substep = "<< step <<"  Time = "<< ((*counterprint)*timer)
<<"\n\n\n"<<endl;
	output << "ResultGroup \" Particles simulation, Hertzian and brute force \" " <<
"\t"+to_string(step)+"   OnNodes \n" << endl;
	output << "\n ResultDescription \"positions\" Vector"<<endl;
	output << "\n   ComponentNames \"rx\" \"ry\" \"rz\" "<<endl;
	output << "\n ResultDescription \"velocities\" Vector"<<endl;
	output << "\n   ComponentNames \"vx\" \"vy\" \"vz\" "<<endl;
	output << "\nValues"<< endl;
}


void printGiD (ofstream& output, int nparticles, int step, vector<Particle> particles, int time, int*
counterprint,float timer)
{
	//cout << "stampo printGID << " << step << " test " << ((*counterprint)*time) << "==" <<
(step) << endl;
	//if sulla scelta dei timestep che voglio stampare
	if ((*counterprint)*time - 1 == step){
		//cout << "stampo printGID << " << step << endl;

		GiDcontentpre(output,step, counterprint , timer);
		*counterprint = *counterprint + 1;
		for(int i=0;i<nparticles;i++){
			particles[i].GiDcontent(output);
		}
		output << "End Values \n\n\n\n" <<endl;
	}
}
```

```
void mesher (ofstream& output, int i, int j, int k, int nummesh){
output << nummesh <<" "<< i << " "<< j << " "<< k << endl;

}
void printmeshGiD (ofstream& output, vector<Particle> particles, int nparticles, Space space, int*
numer){

output.open("log/GiD/resultsdata.post.msh",ios::out);

int nummesh = 1;
float xmax = 0.7;
float xmin = -1.5;
float ymax = 1.1;
float ymin = -0.6;
float zmax = 0.5;
float zmin = -0.5;

float discr = 0.01;

int value_x = (xmax - xmin)/discr;
int value_y = (ymax - ymin)/discr;
int value_z = (zmax - zmin)/discr;


output << "MESH \"particles\" dimension 3 Elemtype Sphere Nnode 1 \n"<<endl;
output <<"# color 242 132 111\n"<<endl;

output <<"coordinates\n"<<endl;

/*
for(int i=0;i<value_x;i++){
for(int j=0;j<value_y;j++){
for(int k=0;k<value_z;k++){
mesher (output,i,j,k,nummesh);
nummesh ++;
}
}
}
*/
for (int i=0;i<nparticles;i++){
particles[i].GiDcoord(output);
}
output <<"end coordinates\n\n"<<endl;

output <<"elements\n"<<endl;

for (int i=0;i<nparticles;i++){
particles[i].GiDmesh(output,value_y,value_z,discr,xmin,ymin,zmin);
*numer = *numer +1;
}
```

```
output <<"end elements\n\n"<<endl;
/*
output << "MESH \"particles\" dimension 3 Elemtype Triangle Nnode 3 \n"<<endl;
output <<"# color 100 100 33\n"<<endl;

output <<"coordinates\n"<<endl;
output <<"end coordinates\n\n"<<endl;
output <<"elements\n"<<endl;
for (int i=0;i<space.numbarr();i++){
space.barriers[i].GiDinit(output,value_y,value_z,discr,xmin,ymin,zmin,numer);
}
output <<"end elements\n\n"<<endl;
*/
output.close();
}


void printmshGiDbarr(ofstream& output, Space space, int numer){

output.open("log/GiD/resultsdatabarr.post.msh",ios::out);

output << "MESH \"particles\" dimension 3 Elemtype Triangle Nnode 3 \n"<<endl;
output <<"# color 130 251 252\n"<<endl;

output <<"coordinates\n"<<endl;

/*
for(int l=0;l<space.numbarr();l++){

space.barriers[l].GiDcoord(output, numer);
}
*/

int nummesh = 1;
float xmax = 1.6;
float xmin = 1.5;
float ymax = 1;
float ymin = -1;
float zmax = 1;
float zmin = -1;
float discr = 0.1;



float a=0;
float b=0;
float c=0;
float d=0;


int valuex = (xmax - xmin)/discr;
```

```
if (valuex==0){
valuex=1;}
int valuey = (ymax - ymin)/discr;
if (valuey==0){
valuey=1;}
int valuez = (zmax - zmin)/discr;
if (valuez==0){
valuez=1;}

for(int i=0;i<valuex;i++){
for(int j=0;j<valuey;j++){
for(int k=0;k<valuez;k++){

a = xmin + i*discr;
b = ymin + j*discr;
c = zmin + k*discr;
d = nummesh + numer;

output << d << " " << a  << " " << b << " " << c <<endl;


nummesh ++;
}
}
}

output <<"end coordinates\n\n"<<endl;

output <<"elements\n"<<endl;
for(int l=0;l<space.numbarr();l++){

space.barriers[l].GiDinit2(output, numer);
}
output <<"end elements\n\n"<<endl;
output.close();

}






void printGiDbarr (ofstream& output, Space space, int step, int time, int* counterprint,float timer,
int numer)
{
            //cout << "stampo printGID << " << step << " test " << ((*counterprint)*time) << "==" <<
(step) << endl;
            //if sulla scelta dei timestep che voglio stampare
```

```
		if ((*counterprint)*time - 1 == step){
			//cout << "stampo printGID << " << step << endl;

			GiDcontentprebarr(output,step, counterprint , timer);
			*counterprint = *counterprint + 1;
			for(int i=0;i<space.numbarr();i++){
				space.barriers[i].csvBARR(output, numer);
			}
			output << "End Values \n\n\n\n" <<endl;
		}
}

void GiDcontentprebarr(ofstream& output, int step, int* counterprint, float timer){

		output << 
"#=========================================================================="<<endl;
		output << "#Step 1 Substep = "<< step <<"  Time = "<< ((*counterprint)*timer) 
<<"\n\n\n"<<endl;
		output << "ResultGroup \" Particles simulation, Hertzian and brute force \" " << 
"\t"+to_string(step)+"   OnGaussPoints \"Triangle\" \n" << endl;
		output << "\n ResultDescription \"Forces\" Vector"<<endl;
		output << "\n   ComponentNames \"Nx\" \"Ny\" \"Nz\" "<<endl;
		output << "\n ResultDescription \"pressures\" Vector"<<endl;
		output << "\n   ComponentNames \"px\" \"py\" \"pz\" "<<endl;
		output << "\nValues"<< endl;
}
```

Particle.h

```
#ifndef __ParticleSimulator__Particle__
#define __ParticleSimulator__Particle__

#include <iostream>
#include <string>
#include <vector>
#include <fstream>

#include "Space.h"

// ----------------------------------------------------------------------

#define X 0
#define Y 1
#define Z 2

// ----------------------------------------------------------------------

using namespace std;

class Particle
```

```
{
public:

    Space *space;

    int identifier;
    int numcontact;

    //--------------------------------

    vector<double> r;
    vector<double> v;
    vector<double> a;
    vector<double> N;

    //--------------------------------

    vector<double> astar;
    vector<double> vstar;
    vector<double> rstar;
    vector<double> Nstar;

    vector<double> v2star;
    vector<double> r2star;

    //--------------------------------

    float rho;
    float q;
    float R;
    float E;
    float poisson;
    float err;
    float erv;

    //--------------------------------

    double m();
    int numcontacts(Particle& p2);

    void reset();
    void computeNf(Particle& p2);
    void interact(Particle& p2);
    void pre_iteration();
    void move(vector<Particle>& particles);

    void interact_barrier(int l);

    //--------------------------------

    void csv(ofstream&);
    void debugger(ofstream&);
    void description();
    void GiDcontent(ofstream&);
    void GiDmesh(ofstream&, int, int, float, float, float, float);
    void GiDcoord(ofstream&);


    //-------------------------------

    Particle(Space& s){ space = &s; }

};

// --------------------------------------------------------------------

#endif /* defined(__ParticleSimulator__Particle__) */
```

Particle.cpp

```
#include<iostream>

#include "Particle.h"
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#include "Fields.h"
#include <string>

using namespace std;

// --------------------------------------------------------------------
// --------------------------------------------------------------------

void Particle::computeNf(Particle& p2)
{
    float norma = sqrtf( pow(r[X]-p2.r[X], 2) + pow(r[Y]-p2.r[Y], 2) + pow(r[Z]-p2.r[Z], 2) );
if(identifier!=p2.identifier){
    for(int k=0; k<3; k++){

        if( q * p2.q < 0 )
            N[k] = N[k] + ( ( ALFA1 * q * p2.q ) * pow(norma, - BETA1 ) - ( ALFA2 * q * p2.q ) *  pow(norma,
- BETA2 ) ) * ( -(r[k] - p2.r[k]) / norma );
        else
            N[k] = N[k] - ( ( ALFA1 * q * p2.q ) * pow(norma, - BETA1 ) + ( ALFA2 * q * p2.q ) *  pow(norma,
- BETA2 ) ) * ( -(r[k] - p2.r[k]) / norma );

    }
    }

}
```

```
// ----------------------------------------------------------------------
// ----------------------------------------------------------------------


void Particle::interact(Particle& p2)
{
   float norma = sqrtf( pow(r[X]-p2.r[X], 2) + pow(r[Y]-p2.r[Y], 2) + pow(r[Z]-p2.r[Z], 2) );
   float delta = norma - (R + p2.R);

   if( delta <= 0 ){

      float R_star = (R * p2.R) / (R + p2.R);
      float E_star = (E * p2.E) / (p2.E * (1 - pow(poisson, 2) + E * (1 - pow(p2.poisson, 2))));

      for(int k=0; k<3; k++){
         N[k] = N[k] + ((4/3) * sqrtf(R_star) * E_star * pow(delta,3/2) + (R * delta) ) * ( -(r[k] - p2.r[k]) /
norma );
      }
   }
}


// ----------------------------------------------------------------------
// ----------------------------------------------------------------------


void Particle::move(vector<Particle>& particles)
{

   Nstar = {0,0,0};
   astar = {0,0,0};

   float DELTA_Tstar = DELTA_T;
   int uniformer=0;
   int counter=0;

   do{

      do{

         Nstar = {0,0,0};
         for(int i=0; i<particles.size(); i++){

            float normastar = sqrtf( pow(rstar[X]-particles[i].r[X], 2) + pow(rstar[Y]-particles[i].r[Y], 2) +
pow(rstar[Z]-particles[i].r[Z], 2) );

            if(identifier!=particles[i].identifier){

               float delta = normastar - (R + particles[i].R);
               float norma_star = sqrtf( pow(rstar[X]-particles[i].r[X], 2) + pow(rstar[Y]-particles[i].r[Y],
2) + pow(rstar[Z]-particles[i].r[Z], 2) );
```

```
          for(int h=0; h<3; h++){

            Nstar[h] = Nstar[h] - ( ( ALFA1 * q * particles[i].q ) * pow(norma_star, -BETA1) + ( ALFA2
* q * particles[i].q ) *  pow(norma_star, -BETA2) ) * ( -(r[h] - particles[i].r[h]) / norma_star );

            if( delta <= 0 ){

              numcontact ++;

              float R_star = (R * particles[i].R) / (R + particles[i].R);
              float E_star = (E * particles[i].E) / (particles[i].E * (1 - pow(poisson, 2)) + E * (1 -
pow(particles[i].poisson, 2)));

              Nstar[h] = Nstar[h] + ((4/3) * sqrtf(R_star) * E_star * pow(delta,3/2))
              * ( -(rstar[h] - particles[i].r[h]) / normastar );

            }

          }

        }

/////////////////////////////////////QQQQQQQQQQQQQQQQQQQQ   condizione speciale per il
contatto
/////////////////////////////////qqqq       elimina manualmente se non serve
float contact_opt = 2.3 - 2.4*R;

if (r[X]>=contact_opt){
        space->contacts = 0;

        for (int l=0; l<space->numbarr(); l++){

          interact_barrier(l);

        }
}
      }

      astar[X] = (q/m()) * ( ELE_X + vstar[Y]*MAG_Z - vstar[Z]*MAG_Y ) + Nstar[X]/m();
      astar[Y] = (q/m()) * ( ELE_Y - vstar[X]*MAG_Z - vstar[Z]*MAG_X ) + Nstar[Y]/m();
      astar[Z] = (q/m()) * ( ELE_Z + vstar[X]*MAG_Y - vstar[Y]*MAG_X ) + Nstar[Z]/m();

      v2star={0,0,0};
      r2star={0,0,0};

      if(counter>=5){
        DELTA_Tstar=DELTA_T/(pow(2,counter-4));

      }

      for(int k=0; k<3; k++){
            v2star[k]=vstar[k];
            vstar[k] = v[k] + DELTA_Tstar*(PHI*astar[k]+(1-PHI)*a[k]);

            r2star[k]=rstar[k];
            rstar[k] = r[k] + DELTA_Tstar*(PHI*vstar[k]+(1-PHI)*v[k]);

         }

         float den_err= sqrt(pow((rstar[X]-r[X]),2)+pow((rstar[Y]-r[Y]),2)+pow((rstar[Z]-r[Z]),2));
         float den_erv= sqrt(pow((vstar[X]-v[X]),2)+pow((vstar[Y]-v[Y]),2)+pow((vstar[Z]-v[Z]),2));
         float den_err0= sqrt(pow((rstar[X]),2)+pow((rstar[Y]),2)+pow((rstar[Z]),2));
         float den_erv0= sqrt(pow((vstar[X]),2)+pow((vstar[Y]),2)+pow((vstar[Z]),2));
         float nom_err= sqrt(pow((rstar[X]-r2star[X]),2)+pow((rstar[Y]-r2star[Y]),2)+pow((rstar[Z]-
r2star[Z]),2));
         float nom_erv= sqrt(pow((vstar[X]-v2star[X]),2)+pow((vstar[Y]-v2star[Y]),2)+pow((vstar[Z]-
v2star[Z]),2));

         if(den_err==0){
            err=nom_err/den_err0;
         }else{
            err= nom_err/den_err;
         }

         if(den_erv==0){
            erv=nom_erv/den_erv0;
         }else{
            erv= nom_erv/den_erv;
         }

         counter ++;

      }while(err> TOLR || erv>TOLV );

      v=vstar;
      r=rstar;
      a=astar;

      uniformer++;

   }while((uniformer*DELTA_Tstar)/DELTA_T==1);
}


// ------------------------------------------------------------------------
// ------------------------------------------------------------------------


void Particle::interact_barrier(int l)
{
   vector <float> Nstar_barr = {0,0,0};
   vector<float> vect_barr_cen={0,0,0};
   float delta_barr=0;
   float dist_tan_temp=0;
   float r_fict=0;
   float angle_tan=0;
   vector <float> proj_tan={0,0,0};
   float norma_dist_tan=0;
   vector <float> vect_bc_projn={0,0,0};
   float angle_norm=0;
   float dist_par_barr=0;
   float scal_prod_1=0;
   float scal_prod_2=0;
   float dt_eff=0;
   float tmp=0;
   float tmp1=0;
   float tmp2=0;
   vector<float> projection_n_t={0,0,0};
   float tmp3=0;
   float k_con=0;
   float dist_norm=0;
   double speed_norm=0;
   vector<double> speed_vers={0,0,0};
   float costeta=0;
   vector<float> multiplier={0,0,0};
   float mult_norm=0;

   for(int i=0;i<3;i++){

      vect_barr_cen[i]=r[i]-(space->barriers[l].center[i]);
      Nstar_barr[i]=0;


      vect_bc_projn[i]=(vect_barr_cen[i]*(space->barriers[l]).n_versor[i]);
      angle_norm=angle_norm+vect_bc_projn[i];

   }


dist_par_barr=sqrtf(pow((vect_barr_cen[0]),2)+pow((vect_barr_cen[1]),2)+pow((vect_barr_cen[2]),
2));
   angle_norm = acos(angle_norm/(dist_par_barr));
   dist_norm = sqrtf(pow(vect_bc_projn[0],2)+pow(vect_bc_projn[1],2)+pow(vect_bc_projn[2],2));
   delta_barr=dist_norm-R;

   if (delta_barr<=0){

      float E_star_p=0;
      E_star_p = (1-pow(poisson,2))/(2*E);

      float E_star_barr=0;
```

```
        E_star_barr=(1-pow((space->barriers[l].poisson),2))/(2*(space->barriers[l]).E);

        float E_star_con=0;
        float e_tmp = 0;
        e_tmp = E_star_barr + E_star_p;

        E_star_con = 1/e_tmp;

        scal_prod_1 = 0;
        scal_prod_2 = 0;
        dist_tan_temp = 0;
        angle_tan = 0;
        tmp3 = sqrtf(pow(delta_barr,2));
        if (tmp3 <=R){
           r_fict = R * sin(acos((R-tmp3)/R));
        }else{
           r_fict = R * sin (acos((tmp3-R)/tmp3));
        }

        speed_norm=sqrtf(pow(vstar[0],2)+pow(vstar[1],2)+pow(vstar[2],2));
        for (int i=0;i<3;i++){

           projection_n_t[i]=vect_barr_cen[i]*sin(angle_norm);

           proj_tan[i]=(projection_n_t[i])*(space->barriers[l].t_versor[i]);
           speed_vers[i] = vstar[i]/speed_norm;

           costeta = costeta + speed_vers[i]*space->barriers[l].n_versor[i];
           angle_tan = angle_tan + (proj_tan[i]);
           dist_tan_temp = dist_tan_temp + pow((projection_n_t[i]),2);
           scal_prod_1 = scal_prod_1 + vect_barr_cen[i]*(space->barriers[l].t_versor_perpendicular[i]);
           scal_prod_2 =scal_prod_2 + proj_tan[i];

        }
        costeta = sqrtf(pow(costeta,2));

        mult_norm = sqrtf (mult_norm);

        norma_dist_tan =
    sqrtf(pow((projection_n_t[0]),2)+pow((projection_n_t[1]),2)+pow((projection_n_t[2]),2));

        tmp1=angle_tan/norma_dist_tan;

        tmp2=acos(tmp1);

        angle_tan = tmp2;

        dt_eff = norma_dist_tan - pow(r_fict,2);
        if (dt_eff<0){
           dt_eff = 0;
        }
    tmp = scal_prod_1 * scal_prod_2;

    if (tmp < 0){
        if ( scal_prod_1 >= 0){
            angle_tan = 2*M_PI - angle_tan;
        }

    } else {
        if ( scal_prod_1 < 0){
            angle_tan = 2*M_PI - angle_tan;

        }
        angle_tan = angle_tan;

    }

    float checker=0;
    if(angle_tan <= ((space->barriers[l]).teta[1])){

        checker = (space->barriers[l]).norm_c1*cos(angle_tan);
        checker = sqrtf(pow(checker,2));

        if(dt_eff<=checker){

            space->contacts = space->contacts + 1;
            for (int n=0;n<3;n++){
                k_con = 4/3*(sqrtf(R))*(E_star_con);

                Nstar_barr[n] = Nstar_barr[n] - k_con * pow(delta_barr,3/2)*(-(space-
>barriers)[l].n_versor[n]);

                space->barriers[l].Nstar[n] = - Nstar_barr[n];
                space->barriers[l].contacts ++;

            }
        }
    }else{
        if(angle_tan <= ((space->barriers[l]).teta[2])){

            checker = (space->barriers[l].norm_c2)*cos(angle_tan);
            checker = sqrtf(pow(checker,2));
            if(dt_eff<=checker){
                space->contacts = space->contacts + 1;
                for (int n=0;n<3;n++){
                    k_con = 4/3*(sqrtf(R))*(E_star_con);

                    Nstar_barr[n] = Nstar_barr[n] - k_con * pow(delta_barr,3/2)*(-(space-
>barriers)[l].n_versor[n]);

                    space->barriers[l].Nstar[n] = - Nstar_barr[n];
                    space->barriers[l].contacts ++;
                }
            }
        }else{
            checker = ( space->barriers[l].norm_c3 )*cos(angle_tan);
            checker = sqrtf(pow(checker,2));

            if(dt_eff<= checker){

                space->contacts = space->contacts + 1;
                for (int n=0;n<3;n++){
                    k_con = 4/3*(sqrtf(R))*(E_star_con);

                    Nstar_barr[n] = Nstar_barr[n] - k_con * pow(delta_barr,3/2)*(-(space-
>barriers)[l].n_versor[n]);

                    space->barriers[l].Nstar[n] = - Nstar_barr[n];
                    space->barriers[l].contacts ++;
                }
            }
        }
    }

  }
  if (space->contacts != 0){
      for (int n=0;n<3;n++){
          Nstar[n] = Nstar[n] + Nstar_barr[n] / space->contacts;
      }


          if(space->barriers[l].contacts!=0){
              for (int n=0;n<3;n++){
                  space->barriers[l].N[n] = space->barriers[l].N[n] + space->barriers[l].Nstar[n] / space-
>barriers[l].contacts;
                  space->barriers[l].pressure[n] = space->barriers[l].N[n]/space->barriers[l].A;
              }
          }
      space->barriers[l].contacts = 0;
  }
}


// -----------------------------------------------------------------------
// -----------------------------------------------------------------------


double Particle::m(){
  return rho * M_PI * R*R*R * 1.33333333;
}
```

```
void Particle::reset(){

   N={0,0,0};

   for (int l=0; l<space->numbarr(); l++){
      space->barriers[l].contacts = 0;
      space->barriers[l].Nstar = {0,0,0};
      space->barriers[l].N = {0,0,0};
      space->barriers[l].pressure = {0,0,0};
   }

}


void Particle::pre_iteration()
{
   vector<float> astar={0,0,0};
   vector<float> v2star={0,0,0};
   vector<float> r2star={0,0,0};
   vector<float> Nstar={0,0,0};

   vstar=v;
   rstar=r;
   err=0;
   erv=0;
}


int Particle::numcontacts(Particle& p2){
   return numcontact+p2.numcontact;
}


// ----------------------------------------------------------------------
// ----------------------------------------------------------------------


void Particle::description()
{
   cout << "Particle " << identifier << " {" << endl;
   cout << "\tr[x]=" << r[X] << "\tr[y]=" << r[Y] << "\tr[z]=" << r[Z] << endl;
   cout << "\tv[x]=" << v[X] << "\tv[y]=" << v[Y] << "\tv[z]=" << v[Z] << endl;
   cout << "\tN[x]=" << N[X] << "\tN[y]=" << N[Y] << "\tN[z]=" << N[Z] << endl;
   cout << "\trho=" << rho << endl;
   cout << "\tq=" << q << endl;
   cout << "\tR=" << R << endl;
   cout << "\tE=" << E << endl;
   cout << "\tpoisson=" << poisson << endl;
}

void Particle::csv(ofstream& output){
   output << identifier << "," << r[X] << "," << r[Y] << "," << r[Z] << "," << v[X] << "," << v[Y] << "," <<
v[Z] << "," << numcontact << "," << q << "," << R <<endl;
}


void Particle::debugger(ofstream& output){
   output << identifier << "," << r[X] << "," << r[Y] << "," << r[Z] << "," << v[X] << "," << v[Y] << "," <<
v[Z] << "," << astar[X] << "," << astar[Y] << "," << astar[Z] <<","<< Nstar[X] << "," << Nstar[Y] << "," <<
Nstar[Z] << "," << numcontact << endl;
}


// ---------------------------------------------------------------------
// ---------------------------------------------------------------------

void Particle::GiDcontent(ofstream& output)
{
	//cout << "\t\tstampo particle " << identifier << endl;
	output << identifier+1 << " " << r[X] << " " << r[Y] << " " << r[Z] << " " << v[X] << " " << v[Y]<< "
" <<v[Z] << endl;
}

void Particle::GiDmesh(ofstream& output, int value_y, int value_z, float discr, float xmin, float ymin,
float zmin){

/*
int xpar = (r[X]-xmin)/discr;
int ypar = (r[Y]-ymin)/discr;
int zpar = (r[Z]-zmin)/discr;

int idmesh = xpar*(value_y+value_z) + ypar*(value_z) + zpar;

output << identifier+1 << "  " << idmesh << "  " << R << endl;

*/

output << identifier+1 << " " << identifier + 1 << " " << R << endl;

}

void Particle::GiDcoord(ofstream& output){
output << identifier + 1 << " " << r[0] << " " << r[1] << " " << r[2] << endl;
}
```

Barrier.h

```
#ifndef __ParticleSimulator__Barrier__
#define __ParticleSimulator__Barrier__

#include <iostream>
```

```
#include <string>
#include <vector>
#include <fstream>

using namespace std;

class Barrier
{
public:

    int identifier;
    int contacts;

    vector<float> points_coord;
    vector<float> center;
    vector<float> n_versor;
    vector <float> t_versor;
    vector <double> N;
    vector <double> Nstar;
    vector<float> t_versor_perpendicular;
    vector<float> c1;
    vector<float> c2;
    vector<float> c3;

    double A;
    vector<double>pressure;

    float  norm_c1;
    float   norm_c2;
    float norm_c3;


    vector<float> teta;
    vector<float> delta_teta;

    float E;
    float poisson;

    vector<float> param;
    vector<float> hess_norm;

    void normal_def();
    void t_perpendicular();
    void csvBARR(ofstream& ,int);

    double Area();

    void GiDinit(ofstream&, int, int, float, float, float, float, int);
    void GiDcoord(ofstream&, int);
    void GiDinit2(ofstream&, int);
    Barrier();
    string toString();


};

#endif /* defined(__ParticleSimulator__Barrier__) */
```

Barrier.cpp

```
#include <math.h>
#include <vector>
#include "Barrier.h"
#include <iostream>
#include <sstream>
#include<iostream>

using namespace std;

void Barrier::normal_def(){

    vector<float> c(3);

    vector<float> teta_temp(3);

    float norm_c1_2 = 0;
    float norm_c2_2 = 0;
    float norm_c3_2 = 0;

    vector<float> normal_vect(3);
    float normal_norm = 0;
    vector<float> n_v(3);

    int l=0;
    int m=0;

    for (int i=0;i<3;i++){
        l=i+3;
        m=i+6;

        c[i] = (points_coord[i]+points_coord[l]+points_coord[m])/3;

        c1[i] = c[i]-points_coord[i];
        c2[i] = c[i]-points_coord[l];
        c3[i] = c[i]-points_coord[m];

    }

    normal_vect[0] = c1[1]*c2[2]-c1[2]*c2[1];
    normal_vect[1] = -c1[0]*c2[2]+c1[2]*c2[0];
    normal_vect[2] = c1[0]*c2[1]-c1[1]*c2[0];
    normal_norm=sqrtf(pow(normal_vect[0],2)+pow(normal_vect[1],2)+pow(normal_vect[2],2));

    for(int i=0;i<3;i++){

        n_v[i]=normal_vect[i]/normal_norm;
        norm_c1_2 = norm_c1_2 + pow(c1[i],2);
        norm_c2_2 = norm_c2_2 + pow(c2[i],2);
        norm_c3_2 = norm_c3_2 + pow(c3[i],2);

    }

    norm_c1 = sqrtf(norm_c1_2);
    norm_c2 = sqrtf(norm_c2_2);
    norm_c3 = sqrtf(norm_c3_2);

    for(int i=0;i<3;i++){
        center[i] = c[i];
        n_versor[i] = n_v[i];
        t_versor[i] = c1[i]/norm_c1;

    }
    for (int i=0;i<3;i++){
        teta_temp[0] = teta_temp[0] + ((c1[i]*c2[i]));
        teta_temp[1] = teta_temp[1] + ((c2[i]*c3[i]));
        teta_temp[2] = teta_temp[2] + ((c1[i]*c3[i]));
    }

    delta_teta[0]=acos(teta_temp[0]/(norm_c1*norm_c2));
    delta_teta[1]=acos(teta_temp[1]/(norm_c3*norm_c2));
    delta_teta[2]=acos(teta_temp[2]/(norm_c1*norm_c3));

    teta[0]=0;
    teta[1]=delta_teta[0];
    teta[2]=teta[1]+delta_teta[1];

    t_perpendicular();
    A = Area();
}


void Barrier:: t_perpendicular(){

    float TETA = 3*M_PI/2;
    vector<vector<float>> rotate;

    for (int i=0;i<3;i++){
        vector<float> tmp(3);
        for(int k=0;k<3;k++){
            tmp[k]=0;
        }
        rotate.push_back(tmp);
    }

    for (int i=0;i<3;i++){

        for (int j=0;j<3;j++){
            if(j==i){
                rotate[i][j] = cos(TETA) + pow(t_versor[j],2)*(1-cos(TETA));

            }else{

                if(i<j){
                    int l=j+1;
                    int m=i+j;
                    if (l>2){
                        if( m % 2){
                            l=0;
                        }else{
                            l=j-1;
                        }
                    }
                    if(m % 2){
                        rotate[i][j]=t_versor[i]*t_versor[j]*(1-cos(TETA))-t_versor[l]*sin(TETA);
                    }else{
                        rotate[i][j]=t_versor[i]*t_versor[j]*(1-cos(TETA))+t_versor[l]*sin(TETA);
                    }
                }else{


                    int l=i+1;
                    int m=i+j;
                    if (l>2){
                        if( m % 2){
                            l=i-1;
                        }else{
                            l=0;
                        }
                    }
                    if(m % 2){
                        rotate[i][j]=t_versor[i]*t_versor[j]*(1-cos(TETA))+t_versor[l]*sin(TETA);
                    }else{
                        rotate[i][j]=t_versor[i]*t_versor[j]*(1-cos(TETA))-t_versor[l]*sin(TETA);

                    }
                }
            }
        }
    }
    for(int j=0;j<3;j++){
        for(int i=0;i<3;i++){
            t_versor_perpendicular[j]=t_versor_perpendicular[j]+rotate[i][j]*t_versor[i];
        }
    }
}


double Barrier::Area(){

double Area=0;
double p=0;
double l1=0;
double l2=0;
double l3=0;
int l=0;
int m=0;

for (int i=0;i<3;i++){
l=i+3;
m=i+6;
l1 = l1+pow(points_coord[i]-points_coord[l],2);
l2 = l2+pow(points_coord[i]-points_coord[m],2);
l3 = l3+pow(points_coord[l]-points_coord[m],2);
}

l1=sqrtf(l1);
l2=sqrtf(l2);
l3=sqrtf(l3);

p=(l1+l2+l3)/2;

return sqrtf(p*(p-l1)*(p-l2)*(p-l3));
}


// ------------------------------------------------------------------------
// ------------------------------------------------------------------------


Barrier::Barrier()
{
    E=100000000;
    poisson=0.15;
    center = {0,0,0};
    n_versor = {0,0,0};
    t_versor = {0,0,0};

    N = {0,0,0};
    Nstar = {0,0,0};

    contacts = 0;

    teta={0,0,0};
```

```
   delta_teta={0,0,0};

   c1={0,0,0};
   c2={0,0,0};
   c3={0,0,0};
   t_versor_perpendicular={0,0,0};

   norm_c1=0;
   norm_c2=0;
   norm_c3=0;

   A=0;
   pressure={0,0,0};
}

// ----------------------------------------------------------------------
// ----------------------------------------------------------------------

string Barrier::toString()
{
   stringstream ss;

   ss << "Barrier{\n";
   ss << "E = " << E << "\n";
   ss << "id = " << identifier << "\n";
   ss << "points_coord = ";

   for(int i=0; i<points_coord.size(); i++){
      ss << points_coord[i] << ",";
   }

   ss << "\n";
   ss << "norm_c1 = " << norm_c1 << "\n";
   ss << "norm_c2 = " << norm_c2 << "\n";
   ss << "norm_c3 = " << norm_c3 << "\n";
   ss << "}\n";

   return ss.str();
}

void Barrier::csvBARR(ofstream& output, int numer){

int id = identifier + numer + 1;
output << id << " " << N[0] << " " << N[1] << " " << N[2] << " " << pressure[0] << " " << pressure[1] <<
" " << pressure[2] << endl;
}

void Barrier::GiDinit(ofstream& output, int value_y, int value_z, float discr, float xmin, float ymin,
float zmin, int numer){
```

```
int id = identifier + numer + 1;

vector<int> idmesh ={0,0,0};
/*
int l=0;
int m=0;

for (int i=0;i<3;i++){

m=i+1;
l=i+2;

int xpar = (points_coord[i]-xmin)/discr;
int ypar = (points_coord[m]-ymin)/discr;
int zpar = (points_coord[l]-zmin)/discr;

idmesh[i] = xpar*(value_y+value_z) + ypar*(value_z) + zpar;
}

output << id << "  " << idmesh[0] << "  " << idmesh[1] << "  "<< idmesh[2] << endl;
*/

for(int i=0;i<3;i++){

idmesh[i] = id * 3 - (3 - i) + 1;
}



output << id << "  " << idmesh[0] << "  " << idmesh[1] << "  "<< idmesh[2] << endl;
}

void Barrier::GiDcoord(ofstream& output, int numer){
/*
int id = identifier + numer + 1;
vector<int> idmesh ={0,0,0};

for (int i=0;i<3;i++){
int l=i+1;
int m=i+2;

idmesh[i] = id * 3 - (3 - i) + 1;

output << idmesh[i] << " " << points_coord [i] << " " << points_coord[l] << " " << points_coord[m] <<
endl;
}
*/
int nummesh = 1;
double xmax = 15;
double xmin = -15;
double ymax = 15;
double ymin = -15;
double zmax = 15;
double zmin = -15;

double discr = 1;

float a=0;
float b=0;
float c=0;
float d=0;


int value_x = (xmax - xmin)/discr;
if (value_x=0){
value_x=1;}
int value_y = (ymax - ymin)/discr;
if (value_y=0){
value_y=1;}
int value_z = (zmax - zmin)/discr;
if (value_z=0){
value_z=1;}

for(int i=0;i<value_x;i++){
for(int j=0;j<value_y;j++){
for(int k=0;k<value_z;k++){

a = xmin + i*discr;
b = ymin + j*discr;
c = zmin + k*discr;
d = nummesh + numer;

output << d << " " << a  << " " << b << " " << c <<endl;


nummesh ++;
}
}
}





}




void Barrier::GiDinit2(ofstream& output, int numer){


int idbarr = identifier + numer + 1;
/*
```

```
vector<int> idmesh ={0,0,0};

for(int i=0;i<3;i++){

idmesh[i] = id * 3 - (3 - i) + 1;
}



output << id << "  " << idmesh[0] << "  " << idmesh[1] << "  "<< idmesh[2] << endl;
*/

int nummesh = 1;
double xmax = 15;
double xmin = -15;
double ymax = 15;
double ymin = -15;
double zmax = 15;
double zmin = -15;

double discr = 1;

int valuex = (xmax - xmin)/discr;
if (valuex==0){
valuex=1;}
int valuey = (ymax - ymin)/discr;
if (valuey==0){
valuey=1;}
int valuez = (zmax - zmin)/discr;
if (valuez==0){
valuez=1;}

vector<int> posx ={0,0,0};
vector<int> posy ={0,0,0};
vector<int> posz ={0,0,0};
vector<int> id ={0,0,0};

for (int i=0;i<3;i++){
int l=i+1;
int m=i+2;

 id[i] = 0;
 posx[i]=0;
 posy[i]=0;
 posz[i]=0;
 posx[i] = (points_coord[i] - xmin)/discr;
 posy[i] = (points_coord[l] - ymin)/discr;
 posz[i] = (points_coord[m] - zmin)/discr;

 id[i] = posx[i]  + posy[i]  + posz[i];
}

output << idbarr << " " << id[0] << " " << id[1] << " " << id[2] << endl;

}
```

Space.h

```
#ifndef __ParticleSimulator__Space__
#define __ParticleSimulator__Space__

#include <iostream>
#include <string>
#include <vector>
#include <fstream>

#include "Barrier.h"

using namespace std;

class Space
{
public:

   int contacts;
           vector<Barrier> barriers;

   int numbarr();

};

#endif /* defined(__ParticleSimulator__Space__) */
```

Space.cpp

```
#include <iostream>
#include <math.h>
#include <vector>

#include "Barrier.h"
#include "Space.h"

using namespace std;

int Space::numbarr()
{
   return (int)barriers.size();
}
```