APC 523
2D Particle-in-Cell (PiC)
Project Report

Nathaniel Chen, Yigit Gunsur Elmacioglu, Kian Orr, Dario Panici

May 2024

Contents

1 Introduction

For the final project, we have written a Particle in Cell (PIC) code which is a widely used simulation method in plasma physics and the electric space propulsion community. The flow of the charged particles inside a duct with a potential difference and a biased obtruding wall will be investigated.

Particle in Cell codes discretize the simulation domain into predefined grid points. These are then used to solve the Poisson's equation with finite-difference method. In 3D, the grid cell is centered at a given grid node and is defined as a cube with the surrounding grid nodes as its vertices. The particles are weighted to cell grids according to their distance to each grid point which will give the charge density term at each node for the Poisson's equation.

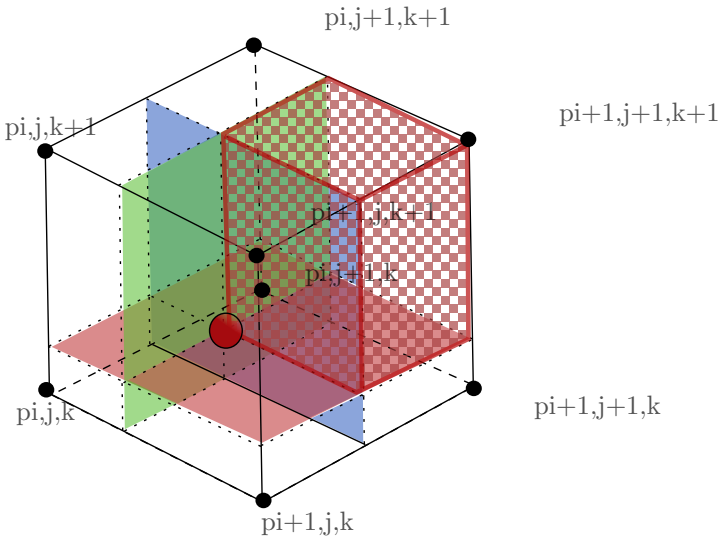

Figure 1: A cell in PIC method with 8 stencil

In Figure 1, 8 stencil of a cell is shown. However, for the actual problem in hand, we use 2D version of this method with 5 stencil points. The charged particle that is shown in red is weighted to grids with inverse proportion to the rectangular prism volume between the corresponding node. For example, the weight of the particle corresponding to node p_{ijk} is the volume of red rectangular prism over the total volume of the cell. This can be applied to each grid point and each particle inside the cell.

Particle in cell method is usually used for dilute gas or plasma states that is the density of the fluid is very low. However, even then, there are on the order of 10^{24} particles in a cubic centimeter. Therefore, it is almost impossible to simulate each particle 1 by 1. This problem is typically overcome using "macro-particles" that are 1 simulation particle that corresponds to $\sim 10^{6-7}$ real particles. This reduces the memory and computational load on the program without sacrificing too much actual physics thanks to the statistical distribution of the particles.

A general outline of the PIC method is below.

1. get initial velocity
2. get initial weights with random r_i
3. get initial electric potential ϕ
4. get initial electric field
5. get acceleration from electric field
6. In loop
   (a) update r_i
   (b) update weights
   (c) solve for electric potential ϕ array
   (d) update E_i
   (e) update velocity

For a time-independent system, the Maxwell equations are

$$\nabla \cdot E = \rho/\epsilon_0 \quad (1a) \qquad \nabla \times E = 0 \quad (1c)$$
$$\nabla \cdot B = 0 \quad (1b) \qquad \nabla \times B = 0 \quad (1d)$$

By combining the E equations,

$$E = -\nabla\phi \implies \nabla^2\phi = -\rho/\epsilon_0 \quad (2)$$

where ϕ is the potential and ρ is the electric charge density. This is known as the Poisson's equation. The special case of ρ=0 is known as the Laplace equation.

In electrostatic case, we can find the acceleration purely from electric force,

$$F = ma = -qE \implies a = -\frac{q}{m}E \tag{3}$$

where m is the mass and q is the charge of the particle. This is what pushes the particles. And from acceleration, we use various integration methods to find he velocity and then position of a particle.

2 Our Problem Setup

We will solve the dynamics of particles in a duct with a biased inlet and outlet, along with a biased barrier wall inside the duct, as shown in Figure 2. The top and bottom walls will have Neumann conditions on the electric potential such that the normal derivative is zero. We will thus solve the Laplace equation for the background electric potential with the following boundary conditions:

$$\frac{\partial \phi(x, y = 0)}{\partial y} = 0 \tag{4}$$

$$\frac{\partial \phi(x, y = height)}{\partial y} = 0 \tag{5}$$

$$\phi(x, y) = \begin{cases} V_{wall} & \text{if } x \in [x_{wall}, x_{wall} + w_{wall}] \text{ and } y \in [0, h_{wall}] \\ V_{in} & \text{if } x = 0 \\ V_{out} & \text{if } x = length \end{cases} \tag{6}$$

In the setup figure shown, $V_{in} = 1000$ Volts, $V_{out} = 0$ Volts and $V_{wall} = 3000$ Volts.

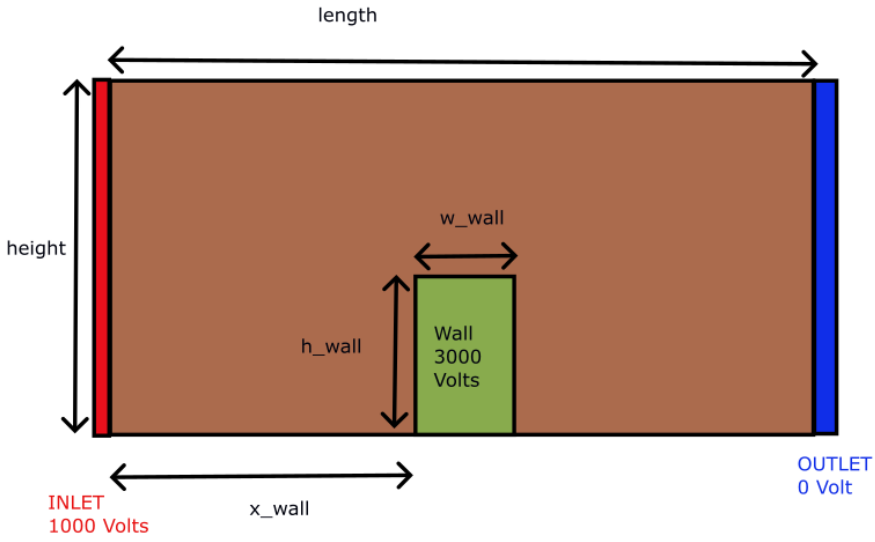

Figure 2: Problem setup for the 2D problem.

All surfaces except for the outlet will have reflecting boundary conditions, which will reverse the particle's velocity in the normal direction to the wall if it crosses the wall boundary. The outlet boundary condition will be an "open" boundary condition, meaning that any particle which exits the domain through the outlet is assumed to be lost and will no longer be tracked.

3 Finite Difference Method for Electric Potential and Field

We will use simple cartesian coordinates for our problem geometry that results in second order PDE in form,

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} \tag{7}$$

Using finite difference between grid points, we can express the PDE numerically as,

$$\frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{\Delta x^2} + \frac{\phi_{i,j-1} - 2\phi_{i,j} + \phi_{i,j+1}}{\Delta y^2} = -\frac{\rho_{i,j}}{\epsilon_0} \tag{8}$$

We can write these equations for each grid point and represent it in matrix form $\mathcal{A}\mathbf{x} = \mathbf{b}$. As we can see from the above equation, only 5 elements per row are none-zero. So, this problem is extremely appropriate for using sparse matrices, which are employed for efficiency. The `scipy.sparse` package is used for this purpose. To solve the resulting sparse linear system, `scipy.sparse.spsolve` is used. An example solution to the Laplace equation for our problem setup is shown in Figure 3. The electric field for the problem is given in Figure 4. We choose to explore this problem in the low-density limit, where the fields from the plasma are insignificant compared to the fields generated by the biasing voltages and therefore can be ignored. Therefore, instead of solving Poisson's equation, we will embrace Laplace's equation.

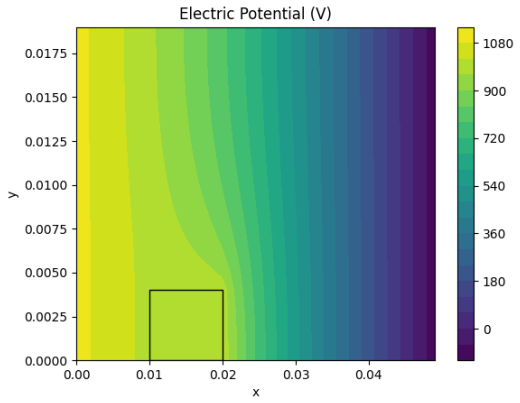

Figure 3: Laplace solution for the electric potential for a domain with both length $L = 0.05m$ and height of size $H = 0.02m$ and a grid spacing in both the horizontal and vertical directions of $h = 10^{-3}m$. The wall is located at $x_{wall} = 0.01m$ and has a width $L/5$ and a height of $H/5$. The biasing voltages are $V_{in} = 1100$ Volts, $V_{out} = -100$ Volts, and $V_{wall} = 1000$ Volts.

Although our particles are assigned to corresponding grid points for electric potential calculation using finite difference method, the motion of the charged particles occur in the 2D continuous domain. Since particles are usually not on the grid points, we need to calculate the force on the particle at some arbitrary point. First, we will use *numpy.gradient* function to get the electric field at the grid points by using the solution of electric potential. Then, we will use *scipy.interpolate.interp2d* to get the interpolated function such that particle pusher can execute the acceleration. There are many available options for interpolation function between 2 grid points, such as 'linear', 'cubic', 'quintic' etc. We use linear as it doesn't make any substantial change to use higher order polynomial for our problem. Since the electric potential is solved using sparse matrix, we already have enough resolution.

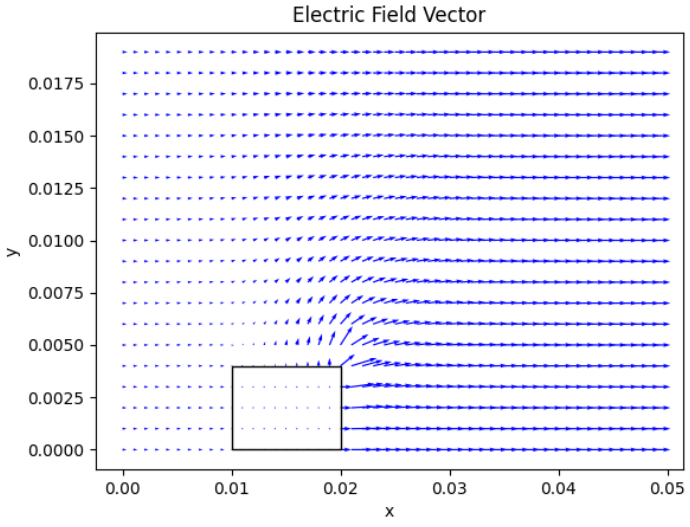

Figure 4: The electric field vector resulting from the gradient of ϕ shown in Figure 3.

4 Algorithm in the Negligible Plasma Field Limit

In the case of the plasma fields being negligible, the only electric field calculation needed is to initially solve the Laplace equation with the boundary conditions given by the biasing voltages outlined in the prior section. The simplified algorithm then becomes

1. get initial velocity
2. get initial positions $\mathbf{r}_i = (x_i, y_i)$
3. Solve for electric potential ϕ
4. Get electric field $\mathbf{E} = -\nabla\phi$
5. Particle Pushing Loop:
   (a) Interpolate electric field from grid it was solved on to the position $\mathbf{r}_i$
   (b) get acceleration $\mathbf{a}(x_i, y_i)$
   (c) update velocity $\mathbf{v}_{i+1}$
   (d) update position $\mathbf{r}_{i+1}$
   (e) Check if a domain boundary was crossed, and apply boundary conditions if so

Steps (b),(c), and (d) are what the different choices of integration algorithm will affect, as different algorithms will have different update schemes.

The reflecting boundary condition is implemented by checking if any of the domain boundaries have been crossed (except the outlet), and if so, the transverse velocity of the particle is reversed at the point of the crossing so that it reflects (conserving energy in the process, like an elastic collision). An example of the reflecting boundary conditions is seen in Figure 4.

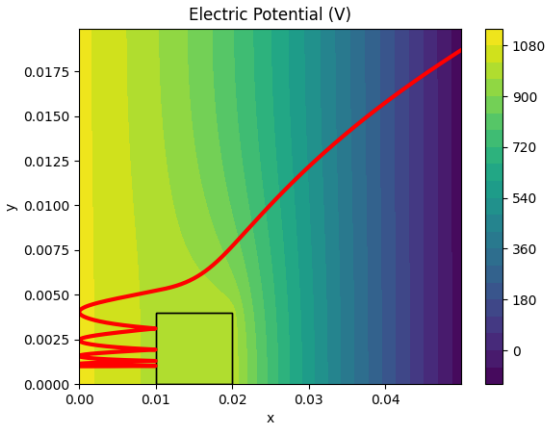

Figure 5: Trajectory of particle that reflects at boundaries.

5 Comparison of Different Integration Methods for Particle Position

For a select number of these integration schemes, they are used on a toy problem of a charged particle undergoing cyclotron motion in a uniform magnetic field in the z direction. The expected behavior is periodic circular motion at a constant radius from the axis.

5.1 General Forward Integrating Schemes

5.1.1 Euler Forward Integration

This is the simplest scheme to carry out. It approximates

$$\frac{q}{m}E(x) = \frac{\mathrm{d}v}{\mathrm{d}t} \approx \frac{v^{n+1} - v^n}{\Delta t} \tag{9}$$

$$v^n = \frac{\mathrm{d}x}{\mathrm{d}t} \approx \frac{x^{n+1} - x^n}{\Delta t} \tag{10}$$

So, the integration formula is,

$$v^{n+1} = v^n + \frac{q}{m} E(x^n)\Delta t \tag{11}$$

$$x^{n+1} = x^n + v^{n+1}\Delta t \tag{12}$$

5.1.2 4th Order Runge-Kutta (RK4)

Given h step size, Euler schemes have errors that grow at h^2 so that the global error is h. RK4 decreases this error to h^5 per step, which is h^4 globally. For general state vector x^n, we can write the fourth order Runge-Kutta algorithm as follows,

$$\begin{aligned} k_1 &= f(\mathbf{x}^n) \\ k_2 &= f(\mathbf{x}^n + k_1\Delta t/2) \\ k_3 &= f(\mathbf{x}^n + k_2\Delta t/2) \\ k_4 &= f(\mathbf{x}^n + k_3\Delta t) \\ \mathbf{x}^{n+1} &= \mathbf{x}^n + (k_1 + k_2 + k_3 + k_4)\Delta t/6 \end{aligned}$$

Here, f is the derivative function. For our problem, $\mathbf{x}^n = [x^n, v^n]^T$ and $f(\mathbf{x}^n) = [v^n, qE(x^n)/m]^T$.

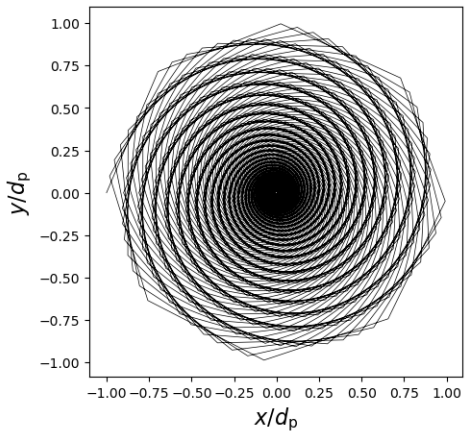

Figure 6: RK4 Method, showing a lack of energy conservation for long enough integration time.

5.2 Conservative Symplectic Integrators for No Magnetic Field

When looking at force-free integration methods, the Leap Frog method is stable for oscillatory motion since it conserves energy. Verlet integration is very similar to leapfrog, only different in that it solves the position and velocity at the same time step. Both methods are less accurate than RK4 though. The algorithm for leapfrog is as follows,

$$\begin{aligned} v^{n+1/2} &= v^n + \frac{q}{m} E(x^n)\Delta t/2 \\ x^{n+1} &= x^n + v^{n+1/2}\Delta t \\ v^{n+1} &= v^{n+1/2} + \frac{q}{m} E(x^{n+1})\Delta t/2 \end{aligned}$$

6 Symplectic Integrators with the Lorentz Force

Charged particles interact with electromagnetic fields. We discussed the electric field effect in the above section. The magnetic field also exerts a force on moving charged particles that is $F_B = q(\mathbf{v} \times \mathbf{B})$. Therefore we'd want to find integrators that can include the magnetic field in a way that's fast but also symplectic. This section will present a few ubiquitous methods and also include comparisons of a single particle circular trajectory in an electromagnetic field to show how energy is or isn't conserved.

6.1 Tajima Implicit Method

Tajima's method is basically the leapfrog method but includes a Lorentz force ($F_{EM} = q\mathbf{E} + q\mathbf{v} \times \mathbf{B}$) integrator such that the cross-product term can be included in a way that works nicely with the integration steps.

$$v^{n-1/2} = v - \frac{q}{m} E \frac{\Delta t}{2} \tag{13}$$

$$\epsilon = \frac{qB}{m} \frac{\Delta t}{2} \tag{14a}$$

$$R = \frac{1}{B} \begin{pmatrix} 0 & B_z & -B_y \\ -B_z & 0 & B_x \\ B_y & -B_x & 0 \end{pmatrix} \tag{14b}$$

$$v^{n+1/2} = [I - R\epsilon]^{-1}[I + R\epsilon]v^{n-1/2} + [I - R\epsilon]^{-1} E \frac{q}{m} \Delta t \tag{15}$$

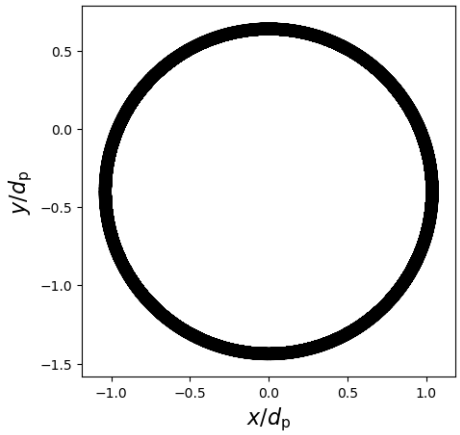

Figure 7: Implicit Tajima Method, showing excellent energy conservation as shown by the orbit maintaining its circular shape.

6.2 Tajima Explicit Method

Notice for the implicit method you need to invert a matrix. This makes the operation computationally expensive. This can be linearized if you approximate a small time step such that you can Taylor expand the inverse matrix. So then you end up with the explicit method with a different v term.

The issue with this though is that it's not conservative. So, you can see that the energy changes hence the thick spiral.

$$v^{n+1/2} = [I + R\epsilon][I + R\epsilon]v + [I + R\epsilon]E \frac{q}{m} \Delta t \tag{16}$$

$$= \frac{q}{m} E \frac{\Delta t}{2} + (I + R\epsilon) \left[v^{n-1/2} + \frac{q}{m} \frac{\Delta t}{2} E \right] \tag{17}$$

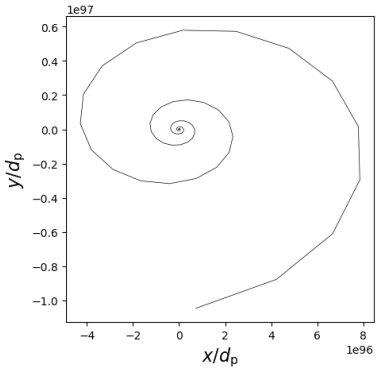

Figure 8: Explicit Tajima Method, showing instability with too large of a time step.

6.3 Boris Method

Boris's method, renowned in computational physics, is particularly valued for its efficiency and accuracy in solving differential equations associated with charged particle motion in electromagnetic fields. This method achieves second-order accuracy, which significantly enhances its precision in numerical simulations. Its computational efficiency is further underscored by the fact that it avoids the need for large-scale matrix multiplications, typically required in other algorithms, thus speeding up the processing time.

Technically, Boris's method employs a leapfrog scheme that alternates between updating velocities and positions, which effectively reduces computational overhead. This leapfrogging contributes to the method's conservation properties, as it approximates a symplectic integrator by conserving phase space volume—a critical feature for long-term numerical stability in dynamical systems.

Despite its symplectic-like behavior, it's crucial to understand that Boris's method does not fully qualify as a symplectic integrator. Symplectic integrators preserve the symplectic two-form exactly, which is a mathematical way of stating they conserve the geometric properties of phase space that are invariant under time evolution. Boris's method only approximates this conservation, manifesting through its ability to skip half velocity steps, which aids in conserving the qualitative nature of the trajectory over time but with some bounds on strict energy conservation. This nuanced distinction is important for applications where long-term energy conservation is critical, such as in celestial mechanics and molecular dynamics simulations.

So here are the steps. First, get velocity rotation,

$$v^- = v + \frac{q}{m} E \frac{\Delta t}{2} \tag{18a}$$

$$t = \frac{qB}{m} \frac{\Delta t}{2} \tag{18b}$$

$$s = \frac{2t}{1 + t^2} \tag{18c}$$

$$v' = v^- + v^- \times t \tag{18d}$$

$$v^+ = v^- + v' \times s \tag{18e}$$

Update velocity and then position,

$$v = v^+ + \frac{qE}{m} \frac{\Delta t}{2} \tag{19a}$$

$$x = x + v\Delta t \tag{19b}$$

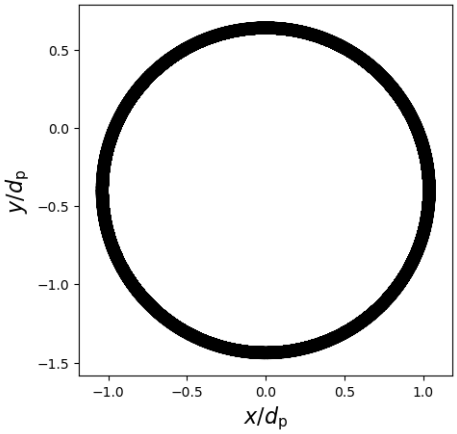

Figure 9: Boris Method solution.

7 Comparison of Result with Different Integration Schemes

For the Euler, RK4 and leapfrog integration scheme outlined above, a single charged particle was initialized at the inlet with positive non-zero horizontal velocity and zero vertical velocity for the problem setup shown earlier in Figure 2. No magnetic fields were included. The particle is then pushed using time steps of size dt using the specified integration scheme, until they exited the domain out the outlet. We will compare,

- phase space trajectories of the particles
- the total energy, $E = \frac{1}{2}mv^2 + qV$

7.1 Comparison of Path Trajectories

With a big enough time step ($1e-7$), the particle trajectories of a particle differ for each integrator, as seen in 10. These trajectories converge when the time step is increased to about $1e-9$ in Figure 11.

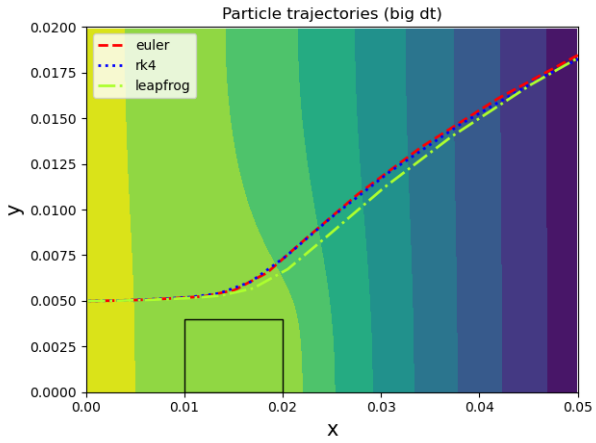

Figure 10: Comparison between forward-Euler, RK4 and leapfrog integrators with $dt = 1 \times 10^{-7}$. Clear differences between each method are seen.

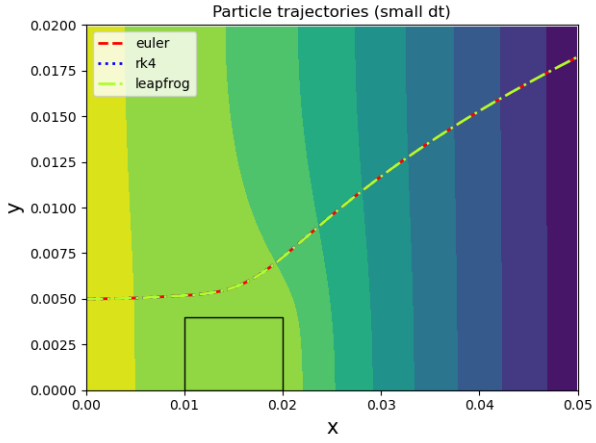

Figure 11: Comparison between forward-Euler, RK4 and leapfrog integrators with $dt = 1 \times 10^{-9}$. The methods converge with a small enough time step

7.2 Comparison of Total Energy

The leapfrog method is a symplectic integrator so energy is expected to be conserved. RK4 is an extremely accurate method, so it is also expected to conserve energy because the system itself conserves energy and it will commit very small errors in energy conservation. Forward-Euler, on the other hand, is only first-order accurate and is not symplectic so it does not conserve energy. Figure 14 displays this behavior for each integrator without a wall. We show the results without the wall, because for this case the electric field, hence the force on the particles, is constant. With a wall, the leapfrog integrator deviates slightly from its original energy.

We check mathematically the energy change in each algorithm step for the Forward Euler and leapfrog in the case of constant electric field only in the x direction (i.e. the no wall case). In this case, the electric potential $V = -x$, the electric field $\mathbf{E} = 1\hat{\mathbf{x}}$, and the potential energy $U = -qx$ where q is the charge of the particle. $\mathbf{F} = q\mathbf{E} = q\hat{\mathbf{x}}$ is the force on the particle, where m is the particle mass, and therefore since $\mathbf{F} = m\mathbf{a}$, $\mathbf{a} = q/m\hat{\mathbf{x}}$. We take $K = \frac{1}{2}mv^2$ to be the energy. Let $E = K + U$ be the total energy and $E_i = K_i + U_i$ and $E_{i+1} = K_{i+1} + U_{i+1}$ be the energy at the time step i and $i+1$ respectively. Since the problem is essentially 1D in this limit, we drop the vector notation for the derivations.

7.2.1 Forward Euler Energy Derivation

$$v_{i+1} = v_i + a\Delta t \tag{20}$$
$$x_{i+1} = x_i + v_{i+1}\Delta t \tag{21}$$
$$E_{i+1} - E_i = \frac{1}{2}mv_{i+1}^2 - qx_{i+1} - \frac{1}{2}mv_i^2 + qx_i \tag{22}$$
$$\text{After plugging in and simplifying, we find that} \tag{23}$$
$$= -\frac{q^2}{2m}\Delta t^2 \tag{24}$$

This shows that the Forward Euler scheme will not conserve energy, and in fact the energy will decrease with each timestep as Δt^2. The difference in energy between the initial and final energy versus the timestep size is shown in Figure 13, along with a quadratic fit of the data, showing that we recover the expected scaling numerically.

7.2.2 Leapfrog Energy Derivation

$$v_{i+1/2} = v_i + a\Delta t/2 \tag{25}$$
$$x_{i+1} = x_i + v_{i+1/2}\Delta t \tag{26}$$
$$v_{i+1} = v_{i+1/2} + a\Delta t/2 \tag{27}$$
$$E_{i+1} - E_i = \frac{1}{2}mv_{i+1}^2 - qx_{i+1} - \frac{1}{2}mv_i^2 + qx_i \tag{28}$$
$$\text{After plugging in and simplifying, we find that} \tag{29}$$
$$= 0 \tag{30}$$

This shows that the leapfrog scheme conserves energy even in the discrete scheme, this is reflected in the figures showing the energy.

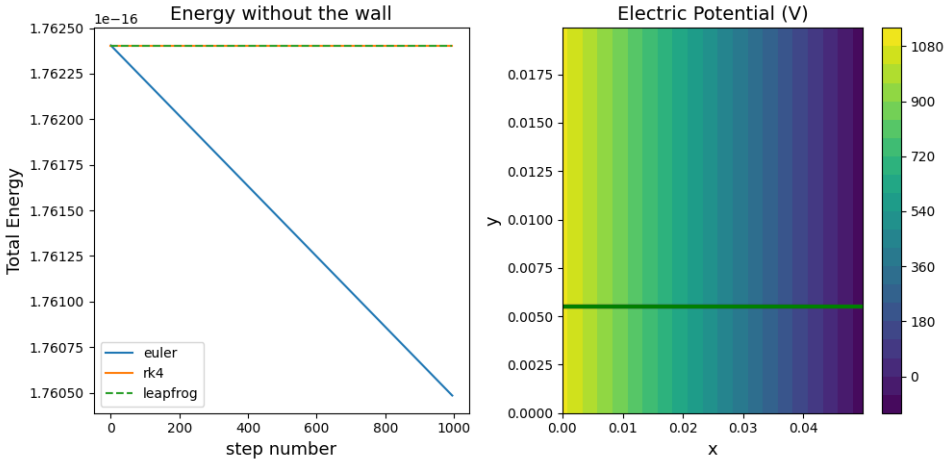

Figure 12: Comparison of the total energy between forward-Euler, RK4 and leapfrog integrators without a wall. RK4 and leapfrog conserved energy, as expected, and forward-Euler does not conserve.

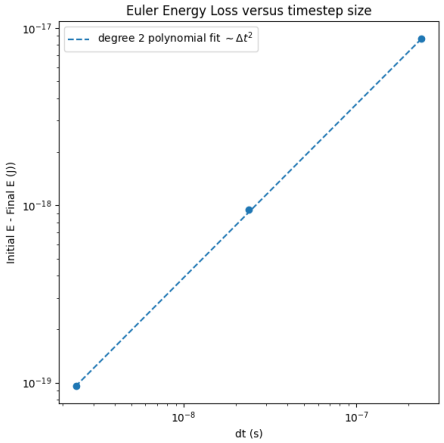

Figure 13: $E_{initial} - E_{final}$ versus Δt for the Euler method, along with a quadratic fit of the data. It shows nearly perfectly the expected scaling of the energy loss with Δt derived earlier in this report..

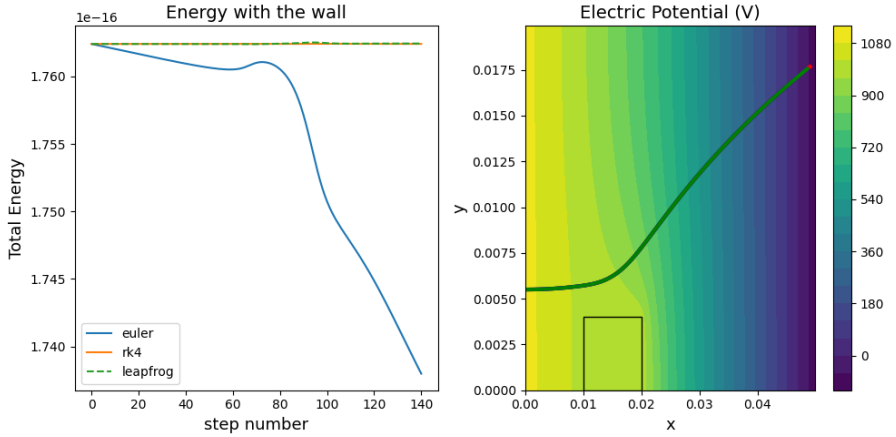

Figure 14: Comparison of the total energy between forward-Euler, RK4 and leapfrog integrators with a wall. The wall causes a slight bump in all three total energies.

7.3 Comparison with Analytical Expression (No Wall)

For a constant electric field, the solution is

$$x = \frac{1}{2}at^2 + v_0 t = \frac{1}{2}\frac{qE}{m}t^2 + v_0 t. \tag{31}$$

This exact solution is compared with the Euler solution in Figure 15

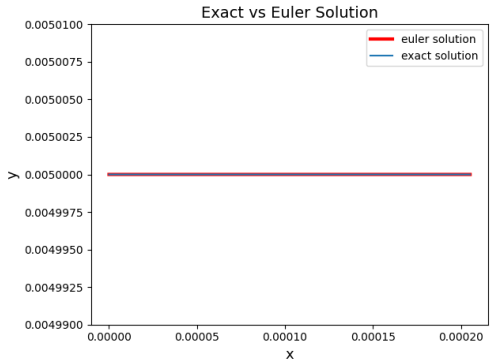

Figure 15: The numerical Euler solution compared against the analytical solution. The solutions agree very closely, with a $dt = 2e - 7$.

8 How to run the code?

We organised our code modularly such that the user only need to call the *__main__.py* to get the trajectory and field plots presented in this report. Additional scenarios can be created using $grid, particle, field$ and *integrator* modules, the names are self explanatory. We didn't use any non-standard Python packages. So, as long as the device have $numpy, matplotlib$ and *scipy*, there shouldn't be any errors. For basic results, the user can type "python -m pic".

Although the structure and using sparse matrices are good practices for the simulation we are running, we could improve our code further. First of all, we heavily use matrix operations and this could be accelerated using a GPU compatible packages like Numba or JAX. Moreover, the way we push the particles is currently by creating a for loop over each particle and updating the position and velocity of them one by one. However, Python for loops are very slow, and the operation that we are doing can be easily converted into matrix multiplication if we use *particles.get_positions* and *particles.get_velocities* functions. This would get rid of the for loop and by combining it with fast matrix operations of GPU, we could upscale our code for orders of magnitude higher number of particles. We didn't implement these steps, because we don't require an extremely high performance for the problem in hand. And those would require redundant complexity to the code and the user who reads the code.