AA244B Project: Development of a 1-D Electrostatic Particle-in-Cell Plasma Simulation

Jeffrey B. Robinson
Master's Degree Candidate, Aeronautics & Astronautics, Stanford University

I. Introduction

The purpose of this project was to develop, and validate the performance of, a Particle-in-Cell (PIC) plasma simulation code. The initial goal of this project was to apply PIC simulation methodology to the analysis of plasma instabilities in magnetoplasmadynamic (MPD) thrusters, in an effort to replicate results gathered by other researchers using magnetohydrodynamic (MHD) simulations [1]. However, due to time constraints this work fell far short of this goal, and was ultimately restricted to the creation and observation of a 1-D PIC code. This paper will address the motivation for the original and revised goals of the project, provide some background on the applicability and theory of PIC simulation, summarize the structure of the PIC code developed here, and discuss the issues encountered in attempting to associate the output of the code with realistic plasma behaviors.

Nomenclature

e	=	elementary charge, $1.6022 \cdot 10^{-19}\ C$	q	=	charge, C
m_e	=	electron mass, $9.109 \cdot 10^{-31}\ kg$	ρ	=	charge density, Cm^{-3}
ϵ_0	=	vacuum permittivity, $8.8542 \cdot 10^{-12}\ C^2N^{-1}m^{-2}$	ϕ	=	electric potential, V
k_B	=	Boltzmann Constant, $1.3806 \cdot 10^{-23}\ JK^{-1}$	E	=	electric field, Vm^{-1}
λ_D	=	$\sqrt{\frac{\epsilon_0 k_B T_e}{ne^2}}$, Debye length, m	T_e, T_i	=	electron / ion temperature, eV
ω_{pe}	=	$\sqrt{\frac{ne^2}{\epsilon_0 m_e}}$, electron plasma frequency, $rad\ s^{-1}$	n	=	number density of real particles, m^{-3}
Δt	=	time step, s	N_{real}	=	number of real particles in simulation
Δx	=	grid node spacing (cell width), m	N_{macro}	=	number of macroparticles in simulation
x	=	particle position, m	L_{sys}	=	simulated system length, m
X	=	grid node position, m	i	=	subscript, particle index
v	=	particle velocity, ms^{-1}	j	=	subscript, grid node index

A. Background

Particle-in-Cell (PIC) simulation is a method of numerically simulating plasma physics in which the theory of single-particle motion is applied to a granular plasma consisting of charged particles, as in reality [2]. This is in contrast to fluid or magnetohydrodynamic (MHD) approaches, which rely on assumptions about collective behavior in the plasma to abstract away individual particles and treat the plasma as a continuous medium with bulk properties alone [3, 4]. Since PIC simulation treats each particle individually, and typically assumes a collisionless plasma for the sake of simplicity and computational efficiency, it is in some sense a literal implementation of the Vlasov equation of kinetic theory [5, 6]. This literal interpretation of plasma dynamics enables the examination of plasma behavior in more detail than is possible with fluid approaches due to the difficulty or impossibility of validating the fluid assumptions in certain cases, such as in plasma-surface interactions or rarefied environments, or the concurrence of these phenomena as in the Earth's ionosphere. In such cases, or really any for which the problem is computationally tractable, PIC simulation is an invaluable tool for the analysis of problems in plasma physics.

Although in theory PIC simulation is extensible to any number of particles present in a volume of ionized gas, the issue of tractability limits the extent and form of its application [2, 5]. The number density of charged particles in a plasma may range from $10^6\ m^{-3}$ in interplanetary space up to $10^{21}\ m^{-3}$ or higher in confined plasma experiments, so any attempt to simulate a significant volume of these plasmas 1-to-1 would quickly implicate the use of a supercomputer, if the problem is at all feasible [3]. Additionally, the assumption of non-collisionality in PIC simulations tends to break down when large numbers of particles are simulated, due to the cumulative effect of neglecting many close interactions between particles in a dense plasma [5]. For this reason, PIC simulations must make do with a reduced number of "macroparticles," with the charge and mass of each representing however many thousands of real particles of a given species. So long as the number of real particles represented by each macroparticle is kept to a reasonable order of magnitude, and thus the intended thermal velocity distributions and fields are sufficiently well represented, real plasma physics can still be well approximated by PIC simulation.

Another issue of approximation inherent to the computational load of simulating an arbitrarily dense plasma is that of solving the fields within the plasma [2, 5]. Since calculating pairwise coulomb forces between all simulated particles would quickly become an impossibly expensive task, the charge densities, potentials, and fields governing the motion of PIC-simulated plasmas are instead resolved on a spatial grid. This grid-based method requires the specification of an interpolation method, in essence the "shape" of the particles in terms of their spatial charge distribution, and thus how they apply to the grid. As with the abstraction of real particles to macroparticles, the level of granularity of the spatial grid must be a careful balance between realism and efficiency, and [2] and [5] provide approximate resolution recommendations to avoid the production of nonphysical behaviors in simulation.

Although PIC simulation is in concept a more realistic method for simulating plasma physics than fluid-based approaches, its inherent discretization of space and the associated approximations make it tricky to apply. Numerical instability and nonphysical behavior is easily generated by poor combinations of inputs, and care must be taken throughout to ensure that physical reality is distinguished from numerical artifacts.

B. Science Objectives

The original objective of this project was to expand on the work done by Hoyt [1], in which a MHD simulation was used to indirectly observe the conditions under which the "anode fall" instability occurs in an MPD thruster. As noted in [1], MPD simulation is unable to accurately represent the instability resulting from charge carrier starvation near the anode of a thruster, so only the conditions for the onset of the instability could be directly simulated using MHD. Since PIC simulation is capable of simulating both bulk plasma behavior as well as accurate surface interactions, a potential extention to this work could make use of PIC to represent the full process of instability in a thruster. A discussion of MPD thrusters and the potential importance of fully understanding their instabilities follows.

Magnetoplasmadynamic (MPD) thrusters, also known as Lorentz Force Accelerators (LFA), are a form of electric space propulsion capable of high thrust and specific impulse with respect to other types of electric propulsion, albeit with high power requirements [7, 8]. MPD/LFA thrusters at NASA Glenn Research Center and Princeton University using Hydrogen propellant have demonstrated thrust on the order of 100 newtons, exhaust velocities on the order of 100,000 meters per second, and efficiencies on the order of 50%, at power levels of 1-4 MW [9]. The combination of high specific impulse and thrust achievable relative to other forms of electric propulsion makes MPD/LFA thrusters a potentially promising option for primary propulsion of future manned interplanetary missions requiring high thrust transfer trajectories. However, MPD/LFA thrusters struggle with power handling, both in the sense that they require currently impractical power inputs to achieve high performance, and that they tend to develop instabilities at high power levels that drastically reduce thrust and cause accelerated component wear [10, 11]. Substantial research effort has been dedicated to the improvement of MPD/LFA efficiency and the suppression of instabilities in anticipation of the development of more robust spacecraft power systems such as nuclear reactors which will be able to support them [11–13].

Once the process of developing the PIC simulation for this paper began, it quickly became apparent that, despite the simplicity of the theory justifying PIC, the implementation is extremely challenging for various reasons including nondimensionalization, input parameter selection, and the choice of diagnostics, among others. Section III discusses some of the considerations made and difficulties encountered in the execution of this project in further detail. Ultimately, this project was limited to the development of a 1-D electrostatic PIC code, and the thus far unsuccessful attempts to remedy its issues with numerical instability and poor energy conservation.

II. Program Methodology

The majority of the methodology used in the development of this PIC code was borrowed from Birdsall and Langdon [2] to, metaphorically, avoid re-inventing the wheel. Birdsall and Langdon have created a comprehensive review of the methods used to develop a PIC code, and this project primarily seeks to replicate the results described by their work. The Julia programming language, a new entrant to the realm of scientific computing within the last decade, was used for this project due to its simple syntax and potential for high performance *. The primary deviation from the methodology used by Birdsall and Langdon is the extension of their concept of "computer variables" to a full suite of nondimensional parameters used throughout the calculations in this PIC code. This nondimensionalization is discussed in section II.B below as a method for simplifying calculations and controlling the relative magnitudes of parameters to minimize the potential for numerical errors.

A. Integration Method

The method of integration used for this code is the "leapfrog" method described by Birdsall and Langdon [2]. The leapfrog method is a simple and computationally efficient method for the integration of second-order linear systems of ordinary differential equations, particularly equations of motion for dynamical systems. The leapfrog method is analogous to the Euler Method for first-order ordinary differential equations, in which derivative information calculated for the current time step is used to calculate position information for the next time step according to Equation 1.

$$y_{\text{new}} = y_{\text{old}} + \dot{y}_{\text{old}}\Delta t \tag{1}$$

In the leapfrog method, the Euler method is applied sequentially to the first-order and second-order components of the system, with an offset of one-half of one time step between the calculations in order to minimize truncation error. This method is attractive when compared to more sophisticated methods such as the canonical RK4 (fourth-order Runge-Kutta) due to its computational expedience, reversibility, and symplectic nature, which promotes energy conservation when solving dynamical problems like those used in PIC simulation. Equation 2 shows the equations of motion used to simulate the motion of electrostatic plasma, as applied using the leapfrog method [2]. Figure 1 demonstrates the half-step offset employed by the leapfrog method to approximate concurrence. This half-step offset must be manually encoded when initializing the leapfrog method, since the equations do not inherently account for this.

$$\begin{aligned} x_{new} &= x_{\text{old}} + v_{\text{old}}\Delta t \\ v_{\text{new}} &= v_{\text{old}} + \frac{qE_{\text{old}}}{m}\Delta t \end{aligned} \tag{2}$$

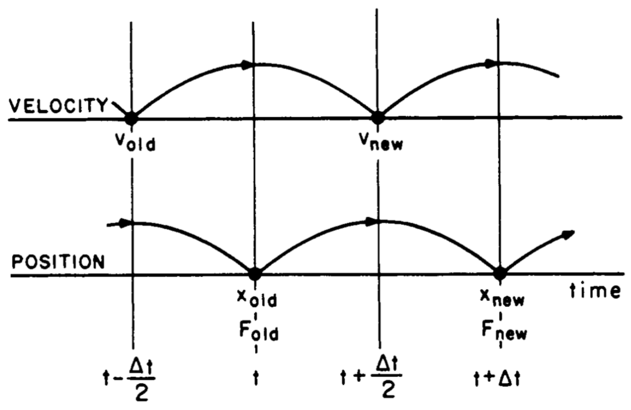


Fig. 1 Schematic of leapfrog method [2]

One key point about the use of the leapfrog method is the calculation of particle kinetic and potential energies to examine energy conservation within the simulated plasma. Since the electric field and potential is found from particle

*Julia is claimed to be comparable in performance to C under some conditions, which would be an invaluable advantage over slower languages like Matlab in intensive, high-dimensional tasks like PIC simulation https://julialang.org/benchmarks/

position information, any calculation of the potential energy naturally corresponds to the particle position steps, as shown in Figure 1. However, since the steps in velocity are offset, the kinetic energy of the particles must be calculated as a mean of the kinetic energy at the half-steps preceding and following the relevant position step. Birdsall and Langdon propose a few different formulas for this modified definition of the kinetic energy in section 4-10, but recommend using $KE = \frac{m}{2} v_{\text{old}} v_{\text{new}}$, as this formula will generate error on the same order as the quadratic ($(\Delta t)^2$) error inherent to the leapfrog method, obviating more accurate forms of the kinetic energy equation.

B. Nondimensionalization

Due to the frequency with which extreme exponents and widely varying orders of magnitude appear in plasma physics, the majority of operations within the PIC code are completed in terms of nondimensionalized values, i.e. as multiples of known reference values. Birdsall and Langdon discuss "computer variables" as useful both for the avoidance of numerical type overflow/underflow and for the circumvention of various mathematical operations otherwise required for the calculation of dimensional values throughout the code [2]. The most apparent case of nondimensionalization in Birdsall and Langdon is the translation described in Equation 3 below, in which particle position and velocity are converted to multiples of the simulation time step Δt and grid node spacing Δx.

$$\left\{\frac{x}{\Delta x}\right\}_{\text{new}} = \left\{\frac{x}{\Delta x}\right\}_{\text{old}} + \left\{\frac{v\Delta t}{\Delta x}\right\}_{\text{new}} \tag{3a}$$

$$\left\{\frac{v\Delta t}{\Delta x}\right\}_{\text{new}} = \left\{\frac{v\Delta t}{\Delta x}\right\}_{\text{old}} + \frac{q}{m}\frac{E_{\text{old}}\,(\Delta t)^2}{\Delta x} \tag{3b}$$

This nondimensionalization circumvents the need to explicitly multiply by Δt for every position step by redefining Δt as the unit of time over which changes in velocity and position occur, although this multiplication is still required for the velocity update calculation. Similar nondimensionalizations were applied to all variables used in the simulation for the aforementioned reasons of magnitude and minimization of mathematical operation count. For brevity, nondimensional variables used in this paper will henceforth be designated by a tilde. Below is a summary of the nondimensionalizations used for the variables in the PIC code, in which the subscript i refers to particle properties, and j refers to grid node properties. The same conversions are applied to all parameters of a given unit, but both particle and node applications are shown below for clarity.

$$\tilde{x}_i = \frac{x_i}{\Delta x} \qquad \tilde{X}_j = \frac{X_j}{\Delta x} \tag{4a}$$

$$\tilde{v}_i = v_i\frac{\Delta t}{\Delta x} \qquad \tilde{a} = \frac{F}{m}\frac{(\Delta t)^2}{\Delta x} \tag{4b}$$

$$\tilde{q}_i = \frac{q_i}{e} \qquad \tilde{q}_j = \frac{q_j}{e} \tag{4c}$$

$$\tilde{m}_i = \frac{m_i}{m_e} \tag{4d}$$

$$\tilde{\rho}_j = \rho_j\frac{(\Delta x)^3}{e} = \frac{q_j}{2e} = \frac{\tilde{q}_j}{2} \tag{4e}$$

$$\tilde{\phi}_i = \phi_i\frac{\epsilon_0\Delta x}{e} \qquad \tilde{\phi}_j = \phi_j\frac{\epsilon_0\Delta x}{e} \tag{4f}$$

$$\tilde{E}_i = E_i\frac{\epsilon_0(\Delta x)^2}{e} \qquad \tilde{E}_j = E_j\frac{\epsilon_0(\Delta x)^2}{e} \tag{4g}$$

For most of these nondimensional parameters, the conversion falls naturally out of the base units used, or the combination of multiplying factors used in the calculations, e.g. electric field E gains a factor of Δx versus the potential ϕ since $E = -\frac{\partial\phi}{\partial x}$. However, for the charge density, the conversion factor from node net charge $\tilde{q}_j$ is somewhat less clear for this 1-D simulation and is worth discussing. Due to the necessity of a volumetric rather than linear density in Poisson's equation, used to determine node potentials, the charge density must be nondimensionalized by $(\Delta x)^3/e$ for the resulting units to match those used to nondimensionalize ϕ. The nondimensional charge density $\tilde{\rho}_j$ must thus be understood as the nondimensional charge accrued over a volume of $(\Delta x)^3$ rather than simply a length Δx. Since the total charge q_j assigned to each node must come from within the grid cells to either side of it, in accordance with the linear interpolation method used (discussed below), the volume over which the net charge is found can be considered equivalent to sweeping a "unit" area, $(\Delta x)^2$, along a distance of $\pm\Delta x$ between the adjacent grid nodes, hence a total volume of $2(\Delta x)^3$. Thus, the nondimensional charge density can be found from the node charge with only a correction of $1/2$.

C. Interpolation

This PIC code uses linear interpolation for the assignment of charge, potential, and electric field. Linear interpolation, also described as "first-order" by Birdsall and Langdon [2], produces an effectively triangular particle "cloud" with an overall width of $2\Delta x$. When assigning charges to grid nodes, this method splits the charge of the particle into two components, one for each of the two nearest nodes along the 1-D grid, and sets their proportionality to the proximity of the particle to the respective node. Birdsall and Langdon note that linear interpolation will conserve momentum, but is not guaranteed to conserve energy as well as higher-order methods - a distinction that may have been implicated in difficulties encountered when attempting to characterize the energy conservation behavior of the code. The linear interpolation method was selected for use in this code over Nearest-Grid-Point weighting and higher-order weighting methods as a good balance between complexity and accuracy.

A preliminary implementation of this simulation used the Coulomb potential, $\phi = \frac{q}{4\pi\epsilon_0 r}$, to determine node potentials, by summing the coulomb potential of each particle in the cells adjacent to a node with respect to the node location. This method would circumvent the calculation of net charge at each node, and the computationally expensive solving of Poisson's equation at each time step, but suffers from singularities when particles are close to, or happen to become co-located with nodes. A modified form of the Coulomb potential, with some physically realistic approximation for small radii, would represent a higher-order interpolation method that could potentially improve the energy conservation behavior of the simulation since the Coulomb potential is a more fundamental and direct method for determining potential.

D. Program Structure

The particle species and spatial grid nodes for the simulation were each implemented as `structs`, with properties as listed below, to simplify and clarify the process of indexing their properties throughout the code. Each particle species is treated as a unit for the purpose of definining these data structures to minimize their memory footprint (vs. individual particle `structs`), ensure physical properties and spatial parameters are kept collated, and permit array-optimized operations when updating particle velocities and positions. The grid nodes were also treated as a single entity. All parameters stored in these data structures are stored in nondimensional form, and any dimensional versions are only stored locally, for example for calculation of energy conservation or plotting.

```
mutable struct PIC_particle_species
    name::String                    # e.g. "electrons"
    q::Float64                      # charge
    m::Float64                      # mass
    xs::Array{Float64, 1}           # positions
    vs::Array{Float64, 1}           # velocities
    vs_old::Array{Float64, 1}       # velocities at last time half-step
    phis::Array{Float64, 1}         # electric potential at particle locations
    Es::Array{Float64, 1}           # electric field at particle locations
end

mutable struct PIC_node_list
    Xs::Array{Float64, 1}           # positions
    charges::Array{Float64, 1}      # net charge of nearby particles
    phis::Array{Float64, 1}         # electric potential at node locations
    Es::Array{Float64, 1}           # electric field at node locations
end
```

The `vs_old` field in the `PIC_particle_species struct` stores the "old" particle velocities to enable calculation of particle kinetic energy concurrently with the particle position step [2]. The `phis` field similarly stores the potential at the particle locations, interpolated using the same equations as the electric field, for the purpose of calculating the particle potential energies.

The simulation code for this project was implemented as a series of functions, each designed to complete a single step of the procedure, such that the execution of the leapfrog cycle simply involves calling each function in sequence. This was done to clarify the structure of the code when read, and to simplify the process of debugging. The functions constituting the 1-D PIC code developed for this project are listed below, and described further later.

- `make_particles`
- `make_nodes`
- `update_node_charge!`
- `update_node_phi!`
- `update_node_E!`
- `update_particle_Es_phis!`
- `update_particle_vs!`
- `update_particle_xs!`
- `run_PIC`

The "!" at the end of most function names is a convention used by some languages to denote a function which modifies its arguments. In the case of this code, these functions do not explicitly output any results, but instead modify the properties of the grid nodes and particles "in-place." The `make_particles` function takes for each particle species: name, number of particles, charge in units of e, and mass in units of m_e; and returns an array of `PIC_particle_species` with the given inputs and zeroes for other parameters. The `make_nodes` function takes the desired number of nodes and returns a `PIC_node_list` with positions initialized to multiples of Δx and zeros for other parameters. The `run_PIC` function takes in the parameters listed below and runs the simulation loop for a specified number of cycles, outputting plots of kinetic, potential, and total energy once completed. The code also saves the input parameters and some key statistics on energy conservation to a .txt file, and saves all plots generated during the simuation.

- Particle species names
- Nondimensional particle charges
- Nondimensional particle masses
- Particle species temperatures in eV
- Particle species drift velocities in m/s
- Particle species number densities in m^{-3}
- Macroparticle counts by species
- Total system length in m
- Time step Δt in s
- Boundary condition identifier
- Number of time steps to simulate

The grid node spacing Δx is found from the total system length and number of nodes, where the number of nodes is set as 1/10 of the number of macroparticles. The actual charges and masses assigned to the particle species in the code are found by multplying the input values by the number of real particles each macroparticle is intended to represent. The latter is found from the number density input for each species and the system length. All calculations involving the charge or mass of the macroparticles thus use the "macro" properties rather than the actual species properties.

E. Particle Loading

Once the properties of the macroparticles have been determined from the input parameters, the positions and velocities of the macroparticles are initialized. For the particle positions, the procedure used depends on the method of initialization desired, and will be described in section III.C. The particle velocities are initialized by sampling randomly from a 1-D Maxwell-Boltzmann distribution, with the intent that a sufficient number of particles will tend to represent the distribution, and thus typical thermal motions, well. Packages for the Julia language provide the ability to sample from several common probability distribution functions, so this velocity initialization was accomplished by specifying a zero-mean Normal distribution with a variance of $\sqrt{k_B T/m}$. Given that the temperature provided to the program is in units of eV, the actual calculation of the variance converts the temperature to Joules rather than multiplying by k_B. The facility for inducing a drift velocity in either particle species was also included, but was not used.

F. Field Solver

The problem of solving for the electric fields at grid nodes, and subsequently particle locations, was broken down into four steps, each represented by a function in the PIC code. The first of these functions in each cycle of the simulation is `update_node_charge!`, which applies linear interpolation to determine the net charge of each grid node.

This function takes in the list of `PIC_particle_species` and the `PIC_node_list`, as well as a boundary condition identifier. The function indexes through the list of particle positions for each species and finds the nodes bounding the grid cell in which each particle is located. The charge of the current particle is then allocated between these two nodes according to the linear interpolation method using Equation 5.

$$\tilde{q}_j \mathrel{+}= \tilde{q}_i \left(\tilde{X}_{j+1} - \tilde{x}_i\right) \qquad \tilde{q}_{j+1} \mathrel{+}= \tilde{q}_i \left(\tilde{x}_i - \tilde{X}_j\right) \tag{5}$$

Once the total charge $\tilde{q}_j$ at each node has been calculated, the next function, `update_node_phi!`, uses the finite-difference form of Poisson's equation to find the potential at each node location (Equation 2-5 (5) from [2]). In accordance with the nondimensionalization described earlier, the dimensional Equation 6a is converted to the nondimensional Equation 6b, which is then implemented in matrix form in the code as Equation 6c and solved using linear algebra facilities in the Julia language.

$$\phi_{j-1} - 2\phi_j + \phi_{j+1} = -\frac{(\Delta x)^2 \rho_j}{\epsilon_0} = -\frac{\tilde{\rho}_j e}{\Delta x \epsilon_0} = \frac{\tilde{q}_j e}{2\Delta x \epsilon_0} \tag{6a}$$

$$\tilde{\phi}_{j-1} - 2\tilde{\phi} + \tilde{\phi}_{j+1} = \frac{\tilde{q}_j}{2} \tag{6b}$$

$$\begin{bmatrix} -2 & 1 & 0 & \dots \\ 1 & -2 & 1 & \\ 0 & 1 & -2 & \\ \vdots & & & \ddots \end{bmatrix} \begin{bmatrix} \tilde{\phi}_2 \\ \tilde{\phi}_3 \\ \tilde{\phi}_4 \\ \vdots \\ \tilde{\phi}_{N_{\text{nodes}}-1} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} \tilde{q}_2 - \tilde{\phi}_1 \\ \tilde{q}_3 \\ \tilde{q}_4 \\ \vdots \\ \tilde{q}_{N_{\text{nodes}}-1} - \tilde{\phi}_{N_{\text{nodes}}} \end{bmatrix} \qquad \tilde{\phi}_1\,,\ \tilde{\phi}_{N_{\text{nodes}}} = \tilde{\phi}_{\text{bias}} \tag{6c}$$

Equation 6c represents the finite-difference form of Poisson's equation described by Birdsall and Langdon in Appendix D of [2]. In this form, the potentials of the first and last nodes are specified manually as either a single "bias" for the first in the case of the periodic boundary condition, or as two different voltage values determined by the known potentials of electrodes at either end of the simulation domain. The tridiagonal nature of the coefficient matrix makes this problem fairly efficient to solve. For the periodic boundary condition, the potential of the first node (bias potential) is specified as zero, the last node is not biased, and the node potentials $j = 2 : N_{\text{nodes}}$ are solved relative to the first node. The first node alone is biased since only the gradient of the potential is relevant in calculating the electric field, and the last node is not co-located with, but rather adjacent to the first node (further explained in section III.E). Birdsall and Langdon note that this method of solving for the potentials, by using a bias potential to define the system of equations, should not affect the calculation of electrostatic potential energy. This step of the simulation was likely the most suspect when verifying model results and debugging, so a further discussion of the modifications of this implementation attempted is provided in section III.F. Birdsall and Langdon discuss the use of a Fast Fourier Transform (FFT) based field solver in their ES1 PIC code, since the assumption of periodicity integral to the FFT method aligns well with the concept of periodic boundary conditions. This FFT method was not implemented in this code due to its complexity, but was considered briefly.

Following the calculation of node potentials, the `update_node_E!` function uses these potentials to calculate the electric field at the node locations using Equation 7c or 7d, depending on the boundary condition specified. This is the matrix form of Equation 7b, which is the nondimensional form of Equation 7a, and represents a central-difference approximation for the gradient of the potential.

$$E_j = -\frac{\partial \phi}{\partial x} = \frac{\phi_{j-1} - \phi_{j+1}}{2\Delta x} \tag{7a}$$

$$\tilde{E}_j = \frac{\tilde{\phi}_{j-1} - \tilde{\phi}_{j+1}}{2} \tag{7b}$$

$$\begin{bmatrix} 0 & -\frac{1}{2} & 0 & \dots & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & -\frac{1}{2} & & 0 & 0 \\ 0 & \frac{1}{2} & 0 & & 0 & 0 \\ \vdots & & & \ddots & & \\ 0 & 0 & 0 & & 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 & 0 & & \frac{1}{2} & 0 \end{bmatrix} \tilde{\boldsymbol{\phi}}_{\boldsymbol{j}} = \tilde{\boldsymbol{E}}_{\boldsymbol{j}} \tag{7c}$$

$$\begin{bmatrix} 1 & -1 & 0 & \dots & 0 & 0 \\ \frac{1}{2} & 0 & -\frac{1}{2} & & 0 & 0 \\ 0 & \frac{1}{2} & 0 & & 0 & 0 \\ \vdots & & & \ddots & & \\ 0 & 0 & 0 & & 0 & -\frac{1}{2} \\ 0 & 0 & 0 & & 1 & -1 \end{bmatrix} \tilde{\boldsymbol{\phi}}_{\boldsymbol{j}} = \tilde{\boldsymbol{E}}_{\boldsymbol{j}} \tag{7d}$$

Equation 7c is the form used for the periodic boundary condition, with the first and last rows "wrapping around" to connect the first and last nodes, as with Equation 6c above. In Equation 7d, which is used for non-periodic boundaries, the first and last rows of the matrix are converted to forward- and backward-difference form due to the lack of another neighboring node to use for central-difference. The utility of the matrix form of this equation is twofold: it eliminates the need for an iterative calculation at each node, and takes advantage of performance-optimized methods for solving linear algebraic equations in matrix/vector form, which are present in most programming languages intended for scientific computing such as Julia.

Once the electric field and potential are known at the locations of the grid nodes, the `update_particle_Es_phis!` function interpolates these values back to the positions of the particles using the same linear interpolation method described for `update_node_charge!`. Equation 8 is implemented in the code for this task, with the code indexing through the list of particle locations for each species to identify the nodes j and $j+1$.

$$\tilde{E}_i = \tilde{E}_j \left(\tilde{X}_{j+1} - \tilde{x}_i\right) + \tilde{E}_{j+1} \left(\tilde{x}_i - \tilde{X}_j\right) \tag{8a}$$

$$\tilde{\phi}_i = \tilde{\phi}_j \left(\tilde{X}_{j+1} - \tilde{x}_i\right) + \tilde{\phi}_{j+1} \left(\tilde{x}_i - \tilde{X}_j\right) \tag{8b}$$

This back-interpolation from nodes to particles has raised some suspicion while working on this code, as the charge of the particles to which these field values are being interpolated should have some effect on the fields at their location, but no satisfactory resolution to this discrepancy was found or implemented as of this writing.

G. Particle Mover

After the electric field and potential are calculated on the grid, and subsequently interpolated back to the locations of the particles, the simpler functions `update_particle_vs!` and `update_particle_xs!` are called to use this field information to advance the spatial particle parameters by one time step. The `update_particle_vs!` function first stores the current particle velocities in the `vs_old` parameter to enable kinetic energy calculation, then applies Equation 9a to update the particle velocities. The `update_particle_xs!` function then applies Equation 9b. The factor of Δt in Equation 9a is debatable, and several variations have been attempted: with this factor; with only the nondimensional variables; with dimensional values for q_i, m_i, and E_i and a factor of $(\Delta t)^2/\Delta x$; and others now forgotten. None of these methods seemed to produce significantly better output than the form shown here, and it is unclear at present which method is accurate to the nondimensionalizations used. It is apparent that multiplying by a dimensional value in determining the nondimensional velocity is contradictory, but the results obtained by modfying this equation to account for this inconsistency were, for lack of a better term, ugly - see Figure 3, as compared to Figure 2b.

$$\tilde{v}_i \mathrel{+}= \frac{\tilde{q}_i}{\tilde{m}_i} \tilde{E}_i \Delta t \tag{9a}$$

$$\tilde{x}_i \mathrel{+}= \tilde{v}_i \tag{9b}$$

Updating the particle velocities prior to the positions is key to adherence to the leapfrog integration method, as this is the key distinction that enables a "leapfrog" step rather than a "lagging" step, which would result from modifying particle positions prior to velocities (see Figure 1). Since the half-time-step offset implicit in leapfrog integration is not explicitly applied at each step by these equations, an initialization function is required to enforce this offset at the beginning of the simulation [2]. This reverse half-step is accomplished by solving for the fields generated by the initial particle placements and then applying the `update_particle_vs!` function, Equation 9a, with a $-\frac{1}{2}$ factor on Δt to send the particle velocities "back in time" half of one time step. Birdsall and Langdon make explicit note of a difference between applying this $\frac{1}{2}$ factor to the time step as opposed to the particle charge in section 3-10, but they leave the mathematical significance of this distinction unclear. Based on the Equation 9a, and the corresponding equations in [2], it seems that for an electrostatic PIC this is superficial.q

III. Diagnostics and Output Verification

The correspondence between physical reality and simulation behavior was the overriding concern when developing this simulation code, as it should be. Although some consideration was given to performance optimization and code organization in development, the majority of the time invested was spent on implementing, and subsequently attempting to interpret, diagnostic measures intended to characterize the physical realism of the simulated plasma behavior. The time spent on validation was divided mainly between: the design of realistic initial and boundary conditions to precipitate physically relevant behavior in the simulated plasma; and the implementation of diagnostic measures to observe that behavior in sufficient detail to characterize its behavior.

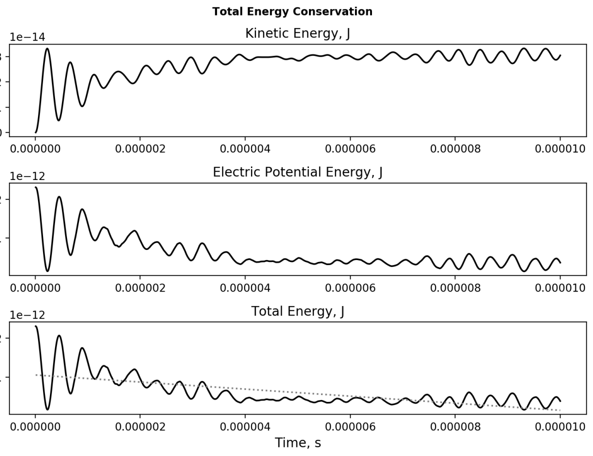
A. Numerical Instabilities

Numerical instabilities have been a feature of this PIC code throughout its development, and have as of yet not been fully characterized or accounted for despite prolonged effort to identify potential sources of instability. Figure 2 demonstrates some of the unpredictability of the code when running the same simulation with changes made only to the number of macroparticles. The input parameters used in these simulations were chosen to allow for approximately 10 Debye lengths within the simulation domain, with each cell representing a fraction of a Debye length in an attempt to avoid numerical instability. The time step and grid spacing were chosen as conservative values far below the maxima stated by [5] and [6] as leading to numerical instabilities, with the intent that these inputs would be removed as likely sources of error. Similarly, the number of macroparticles per physical particle was maintained below the order of 10^6.

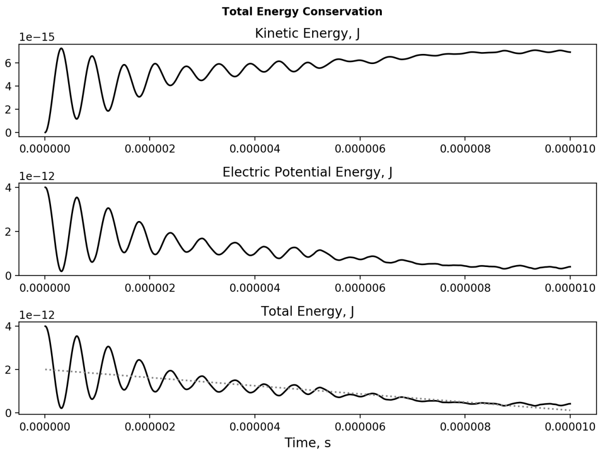
The simulations shown in Figure 2 are representative of the kinds of results generated by the code for a variety of temperature, time step, and density input conditions. Figure 3 represents results obtained when the velocity update equation was changed to be "actually" nondimensional. As demonstrated by the plots of kinetic energy, an electron plasma oscillation is initiated immediately in all cases, but in all cases this results in oscillation at a different frequency. As the number of particles grows, so does the period of the oscillation, with the case using 5000 macroparticles reaching approximately the correct frequency as described by the theory [3]. Additionally, in all cases a strong damping behavior is observed in the oscillations, and energy is not conserved. Observation of the energy conservation plots for the electrons alone shows that the vast majority of the kinetic energy is accounted for by their motion, so the kinetic energy plots here are certainly representative of electron motion.

An odd trend consistent throughout all simulations run with this code thus far is an apparent mismatch between the order of magnitude of the kinetic and potential energies, since their oscillations in all cases appear to be nearly perfectly out of phase and similar in profile. In all cases, however, the potential energy is several orders of magnitude larger than the kinetic energy, so the total energy tends to closely follow the behavior of the potential energy and show a net loss of energy from what should be a closed system.

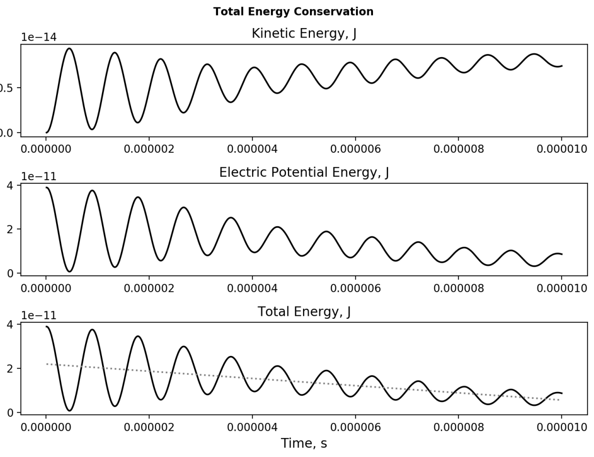
The nondimensionalizations used for this code are highly suspect in the investigation of sumerical instabilities due to the simplistic conversions used compared to [5] or [14]. The units used here were derived mainly following from the "computer variables" used by [2], and it seems that an insufficient level of consideration was given to the implications of this nondimensionalization for the calculations. While the exact sources of the instabilities are not clear, one plausible explanation is the incorrect application of nondimensional units resulting in absent or excess factors in the calculations. A potential improvement to this code would be to make use of the more physically grounded nondimensionalizations used by [5, 14] to ensure that this is not the source of instability.



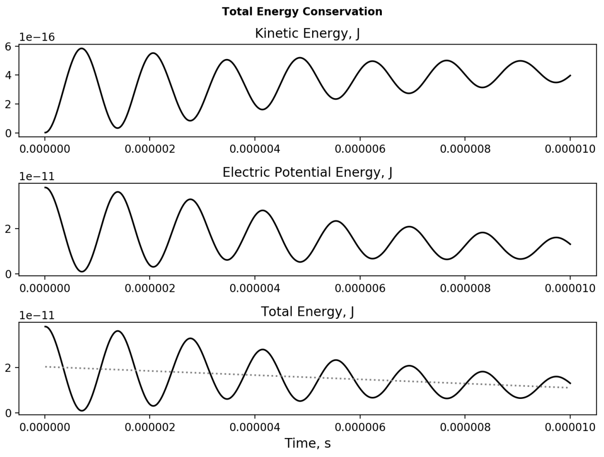
(a) $N_{\text{macro}} = 500,\ N_{\text{nodes}} = 50\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 10^5,\ \frac{\lambda_D}{\Delta x} = 5.26\right)$



(b) $N_{\text{macro}} = 1000,\ N_{\text{nodes}} = 100\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 5 \cdot 10^4,\ \frac{\lambda_D}{\Delta x} = 10.5\right)$



(c) $N_{\text{macro}} = 2000,\ N_{\text{nodes}} = 200\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 2.5 \cdot 10^4,\ \frac{\lambda_D}{\Delta x} = 21\right)$



(d) $N_{\text{macro}} = 5000,\ N_{\text{nodes}} = 500\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 10^4,\ \frac{\lambda_D}{\Delta x} = 52.6\right)$

Fig. 2 Net energy conservation and oscillation frequency variation with macroparticle count (N_{macro}, applied per species). Input Conditions: realistic electron and proton masses and charges, $T_e = 0.005\ eV, T_i = 0.001\ eV$, $n = 10^8\ m^{-3}$ (both species), $L_{\text{sys}} = 0.5\ m$ ($\lambda_D = 0.0526\ m$), $\Delta t = 10^{-8}\ s$ ($\omega_{pe} = 5.64 \cdot 10^5\ s^{-1}$, $(\omega_{pe}\Delta t)^{-1} = 177.3$), 1000 time steps shown, periodic boundary condition

B. Diagnostics

In order to characterize the behavior of the plasma being simulated, and to assist the determination of the sources of numerical instability in the code, several diagnostic tools were implemented to track particle motion. These include the logging of kinetic, potential, and total energy, as shown in Figures 2 and 6, as well as plots of particle phase-space, grid node potential, and grid node electric field which were generated at specified time step intervals throughout the simulation. Figure 4 demonstrates these plots for the same simulations as Figure 2, immediately following particle loading and initialization via the reverse half-step in velocity. These plots demonstrate the effect of increased particle density on filling out the phase-space, and conversely the strong correlations between particle position and velocity at lower macroparticle counts. Also of note in these diagnostic plots is the jagged electric field in all cases, which is somewhat questionable, albeit difficult to diagnose as accurate or errant.

In Figure 2, the dotted lines on the total energy plots represent linear regression of the total energy data. In all four cases the slope of this linear fit was between -0.02 and -0.04 percent per time step relative to the initial value. In addition to the linear fit, the deviation of the total energy is tracked by the code in terms of its maximum deviation, RMS deviation, and deviation at last time step. These values are typically on the order of 50 to 100 percent, reflecting the

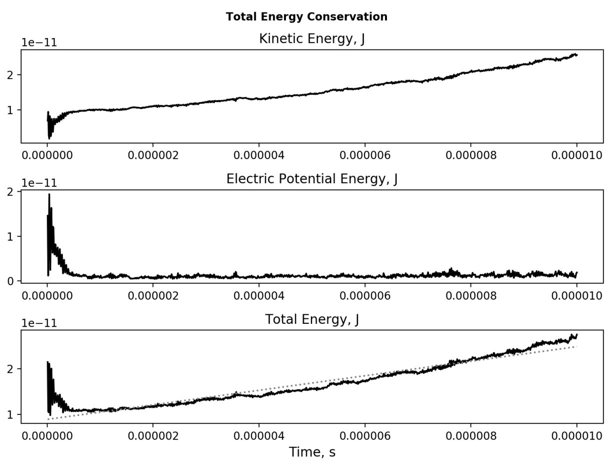


Fig. 3 Net energy conservation plots for same conditions as Figure 2b, with "correct" dimensional ⟶ nondimensional velocity update calculation $\tilde{v}_i$ += $\frac{\tilde{q}_i}{\tilde{m}_i}\tilde{E}_i\frac{e^2}{m_e\epsilon_0(\Delta x)^2}\frac{(\Delta t)^2}{\Delta x}$

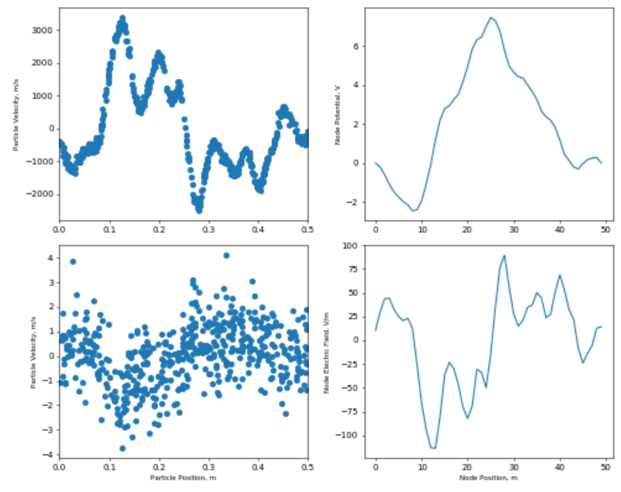
massive stability issues with this code in its present state.

In addition to the above diagnostics, the code also tracks the velocity distribution of each species at the same time step intervals as it outputs plots. At each of these steps, a Normal distribution is fitted to the list of velocities for each particle species, and the mean and variance of these distributions is stored for comparison with the distribution used to initialize particle motion. The code outputs several statistics about the mean and variance of the velocity distributions after completion, including RMS error, mean error, and error at the first step after initialization and the last step of the simulation. These "error" calculations are all done with respect to the initial variance of the velocity distribution since it is a zero-mean distribution unless a drift velocity is specified. For the simulations shown in Figures 2 and 4, the accuracy with which the intended thermal velocity distribution is represented depends heavily on the number of macroparticles used, as the distribution in the first step after velocity initialization was displaced by more than 100 percent for the case with 500 macroparticles per species. In the case with 5000 macroparticles per species, the velocity distributions for both species only deviated by one to two percent. The statistics tracking deviation of the distribution over time typically show changes in the distribution of thousands of percent, and it is not clear whether this is due to physical or non-physical behaviors.

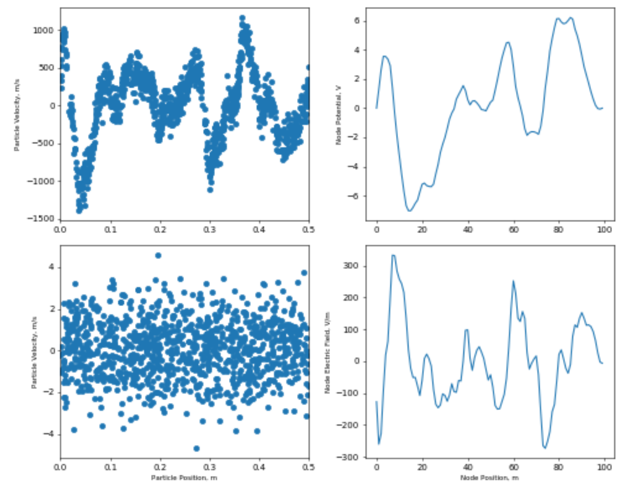
An attempt was made to incorporate Fast Fourier Transforms (FFTs) of particle position histories to empirically determine the frequency of any plasma oscillations with greater accuracy than simply looking at the kinetic energy plots in Figures 2 or 6. The `FFTW` package for Julia was used to generate FFTs of the position histories of each particle in the simulation, but none of the FFT results showed peaking beyond zero frequency, despite ample time-domain resolution and duration to capture any oscillations. This issue is especially odd considering that observations of the phase-space and grid potential plots throughout simulation, e.g. Figure 4, seemed to show oscillations, or some other type of "sloshing" behavior, at what appeared to be the plasma frequency based on time step resolution. This discrepancy has not been solved as of yet.

C. Initialization / Particle Loading

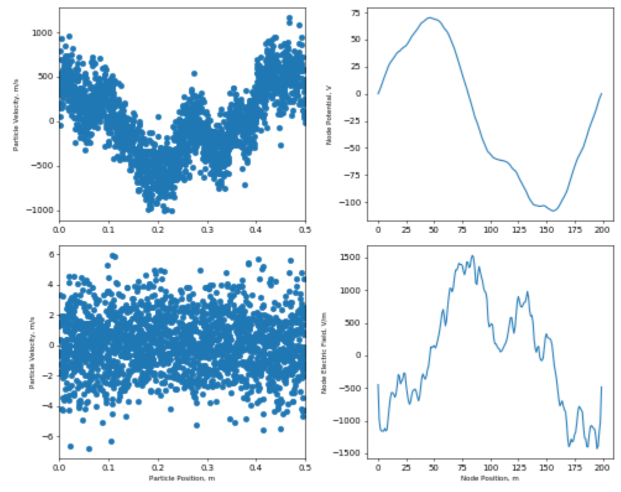
A realistic particle loading and initialization strategy is essential for the validity of a PIC simulation, as "where the plasma came from" necessarily defines its future behavior. For this PIC code, the plasma was initialized as distributed evenly over the simulation domain according to a given distribution function or spacing method rather than injection from a source. The particle loading strategy used for the majority of the testing with this code involved sampling particle positions from a uniform distribution between zero and the system length, with the intent that a sufficient number of particles should fill the domain evenly. In terms of the simple goal of achieving an apparently uniform distribution of particles this method works well, as demonstrated by the horizontal particle spacing shown in Figure 4. However, Figure 4 also demonstrates the spurious electric fields and odd potential distributions generated by this method, since the locations of electrons and ions are sampled from an identical distribution with no regard for the



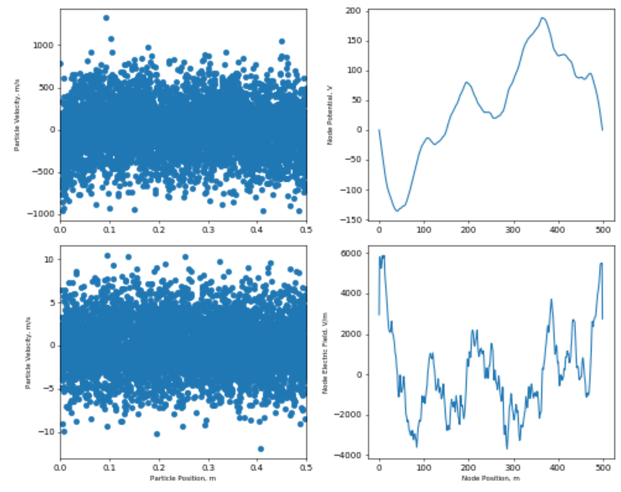
(a) $N_{\text{macro}} = 500,\ N_{\text{nodes}} = 50\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 10^5,\ \frac{\lambda_D}{\Delta x} = 5.26\right)$



(b) $N_{\text{macro}} = 1000,\ N_{\text{nodes}} = 100\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 5 \cdot 10^4,\ \frac{\lambda_D}{\Delta x} = 10.5\right)$



(c) $N_{\text{macro}} = 2000,\ N_{\text{nodes}} = 200\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 2.5 \cdot 10^4,\ \frac{\lambda_D}{\Delta x} = 21\right)$



(d) $N_{\text{macro}} = 5000,\ N_{\text{nodes}} = 500\ \left(\frac{N_{\text{real}}}{N_{\text{macro}}} = 10^4,\ \frac{\lambda_D}{\Delta x} = 52.6\right)$

Fig. 4 Phase space, grid potential, and grid electric field variation with macroparticle count, shown after initialization and reverse half-step of velocity. Upper left of each plot is phase-space for electrons (m/s vs m), lower left is phase-space for protons (ions) (m/s vs m), upper right is grid potential (V vs node index), lower right is grid electric field (V/m vs node index). Input Conditions same as in Figure 2.

dynamics of charged particle interaction that would lead to such configurations. Birdsall and Langdon discuss the effect of such inconsistencies in spatial distribution as effective in initiating plasma oscillations, and otherwise not significantly detrimental to the simulation of realistic plasma behavior [2]. Indeed, this method seemed to be effective in initiating oscillations, but due to difficulties elsewhere in the code these oscillations were not stable.

Other spatial particle loading methods tested included: mathematically uniform particle spacing, in which case the particles were spaced in intervals determined from the system length and number of macroparticles; and the same, but with a "shuffle" of 1 percent of the spacing interval to add some randomness to an otherwise perfectly uniform particle spread. Neither of these methods produced notably different results from random sampling when supplied with the same velocity distribution.

The method used for particle velocity loading in this code has already been discussed in section II.E. To reiterate, the particle velocities in all test cases were sampled randomly from a Normal distribution with mean equal to the drift velocity specified by the user and variance $\sqrt{k_B T/m}$ - making this a 1-D Maxwell-Boltzmann distribution. For most test cases the temperature of both species was kept at or below 0.005 eV in an attempt to limit the Debye length of the plasma, and thus allow many Debye lengths within the simulation domain. This was also done to keep the plasma firmly in the "cold plasma" regime, where the frequency of any electron plasma oscillations observed could be expected to match the theory (which assumes a background of stationary ions) to a reasonable margin of error [3]. Birdsall and Langdon dedicate a chapter in [2] to the discussion of particle loading, and particularly the use of non-random scrambling of velocity and position pairings to ensure good representation of the intended velocity distribution down to small scales within the simulation. This method was not implemented in this code primarily due to a lack of awareness of this possibility until fairly late in this project. Considering the potentially significant impact initial conditions can have on the outcome of a simulation, a more thorough investigation into the source of instabilities in this code would likely make use of such non-random initialization procedures to eliminate additional process variables.

D. Boundary Conditions

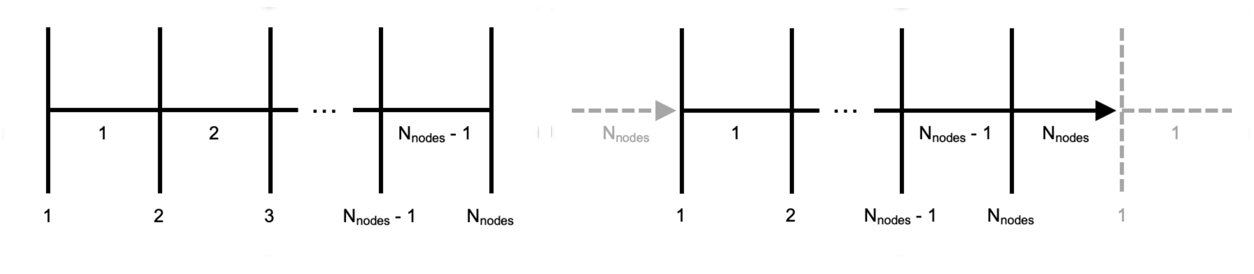
The boundary conditions applied to this simulation have already been briefly addressed with regard to the equations used for electric field calculation, but a more thorough discussion follows. This PIC code was designed with facilities for two types of boundary conditions: periodic and "sheath." The periodic boundary, as the name suggests, treats the plasma as if it were composed of infinitely many adjacent copies of the simulated domain. In this paradigm, all fields are calculated as if the first and last nodes were adjacent, as shown in Equations 7c and 6c. Similarly, when particle positions are updated at each time step, their position is actually taken as the modulo of their updated position and the length of the system, ensuring that particles near the system bounds are retained and enabled to traverse the simulation domain boundary without discontinuity. During the development of this code, however, several issues were encountered with the periodic boundary condition, largely stemming from the interpretation of distance used for the calculations involving linear interpolation. For Equations 8 and 5, in particular, the issue of calculating particle and node weighting by measuring along the entire length of the simulation domain lead to anomalous accelerations among particles in both the first and last cells, causing an extreme "stirring" motion of the particles resulting in behavior reminiscent of a two-stream instability. Once this was corrected, the behavior of the system appeared to be corrected, but there is still some uncertainty about energy and momentum conservation behavior at the edges of the simulation domain.

The "sheath" boundary condition is in some sense the natural counterpart to the periodic boundary condition, and simulates the behavior of the plasma when bounded on either side by ideally absorbing dielectric walls, hence its designation as the case in which sheath buildup should be observed. The "walls" in this case are simply the nodes at either end of the simulation domain, and whenever the electrical force on a particle causes it to travel beyond either of the walls, it is "removed" from the simulation. Since actually removing the particle from the simulation would involve altering the lengths of the arrays containing particle information and permanently offsetting the charge of the end grid nodes, the particles beyond the simulation bounds are not explicitly deleted from the simulation. Instead, once a particle crosses beyond the simulation domain, the electric force and potential are no longer interpolated to its location from the grid, and it is allowed to drift under the influence of whatever velocity it possessed at the moment it passed beyond the last grid node. In order to account for the fact that this particle has been "absorbed" by whichever wall it crossed, its charge is contributed entirely to the grid node representing that wall in the `update_node_charge!` function. The sheath boundary condition was examined far less thoroughly than the periodic boundary condition during development and debugging of this PIC code, so its behavior is less well understood than the periodic boundary condition.

E. Node Location Relative to System Bounds

In both boundary conditions, an important issue of interpretation when initializing the simulation is the issue of where to place the nodes. While this may not seem to be an issue, as the grid spacing is well defined by the system length and node count, the interpretation of the first and last node locations becomes important when considering the issue of discontinuity at the boundary in the periodic case. As shown in Figure 5, the edge of the simulation domain can be considered to be at the location of the last grid node, or at the end of the next Δx unit of distance beyond the last grid node depending on convention. In the present PIC code, the latter interpretation was applied for the periodic boundary condition, such that a particle traversing the edge of the grid would be able to exist in the interstitial space between the last and first grid nodes without inducing any discontinuity in the fields or "disappearing" temporarily. Correspondingly, the former interpretation involving limiting the simulation domain by the node locations was applied for the "sheath" boundary condition.

This distinction manifested itself in several locations in the code, beginning with the calculation of Δx from a given



(a) Non-periodic ($N_{\text{cells}} = N_{\text{nodes}} - 1$) **(b) Periodic ($N_{\text{cells}} = N_{\text{nodes}}$)**

Fig. 5 Node and cell count conventions for 1-D PIC boundary conditions

system length and node count. The periodic boundary condition needs to include an "extra" Δx beyond the last node in order for the first and last nodes not to be co-located, so the number of nodes and cells must be equal, such that $\Delta x = L_{\text{sys}}/N_{\text{nodes}}$. For the "sheath" boundary condition, the system instead must end at the last node, such that there is one fewer cells than nodes: $\Delta x = L_{\text{sys}}/(N_{\text{nodes}} - 1)$. In the periodic boundary condition, this distinction also appeared in the particle position update calculation, as the positions of the particles following this calculation could lie beyond either end of the simulation domain. To account for this possibility, an additional step was added to the particle mover function to set the new particle particle position as the modulo of the calculated particle position and the length of the system. This allows particles to seamlessly traverse the bounds of the system without any discontinuity in the field solver. Similarly, as discussed in section II.F, the matrix forms of the field solvers were modified slightly to allow finite difference formulae to include the first and last nodes as adjacent.

F. Poisson Solver

The solution to Poisson's equation, used to determine the node potentials, is one of the prime suspects in searching for sources of numerical instability and energy conservation issues. In its first incarnation, this portion of the code was implemented using a similar formula to Equation 6c in section II.F, but with the voltage biases omitted and all nodes thus handled simultaneously. The resulting coefficient matrix was nearly tridiagonal, but with the top right and lower left corners equal to 1. This was a preliminary and rather literal interpretation of the finite-difference form of Poisson's equation, wherein the odd corners equal to one resulted from allowing the [1, -2, 1] chain of coefficients to wrap around the end of the matrix row for the first and last nodes (rows), and thus connect them to solve for the periodic boundary condition. While this implementation appeared to work well, a closer inspection of the behavior of this coefficient matrix yielded the worrying discovery that its inverse contained exclusively values of order 10^{15}. Although the exact interpretation of this information was clouded by a lack of understanding of the linear algebra concepts governing this phenomenon, this seemed like a situation ripe for numerical instability. With the coefficient matrix in tridiagonal form, as shown in Equation 6c, the inverse matrix appears far more reasonable, with all values of more reasonable magnitude.

In addition to issues with linear algebra, the Poisson solver step of the program also contained the most questionable application of nondimensionalization, that being the conversion between node charge and charge density. The full rationale for this conversion and its mathematical statement are shown in section II.B. In determining a satisfactory rationale for the nondimensionalization used, several different configurations were tried in the solver code, including among others: using a factor of $\frac{3}{4\pi}$ instead of $\frac{1}{2}$ with the justification that the node charge represented the charge within a spherical volume of Δx radius; using a factor of $\frac{\Delta x}{2}$ instead of $\frac{1}{2}$ with the justification that the charge density was a linear density, and thus would only cancel a single factor of Δx. The former attempt produced fairly little change in the output of the code, as shown in Figure 6, with the only notable change in output for the same initial conditions being a change in the frequency of oscillation. In fact, due to the obscuring effects of numerical instability, evident in the increasing kinetic energies and oscillating total energies in Figure 6, the apparent plasma frequency of the simulation using the factor of $\frac{1}{2}$ actually appears to be farther from the actual plasma frequency than that of the simulation using the other factor. The factor of $\frac{1}{2}$, reported in earlier sections, was eventually determined to be the correct factor, given the explanation in section II.B.

An alternative method for calculating electric fields is discussed in Appendix D of [2], using Gauss' Law rather than Poisson's equation to translate directly from node charges to electric fields, and in so doing circumvent the need to specify a bias voltage to define the potential of the system. However, this method instead requires either the specification

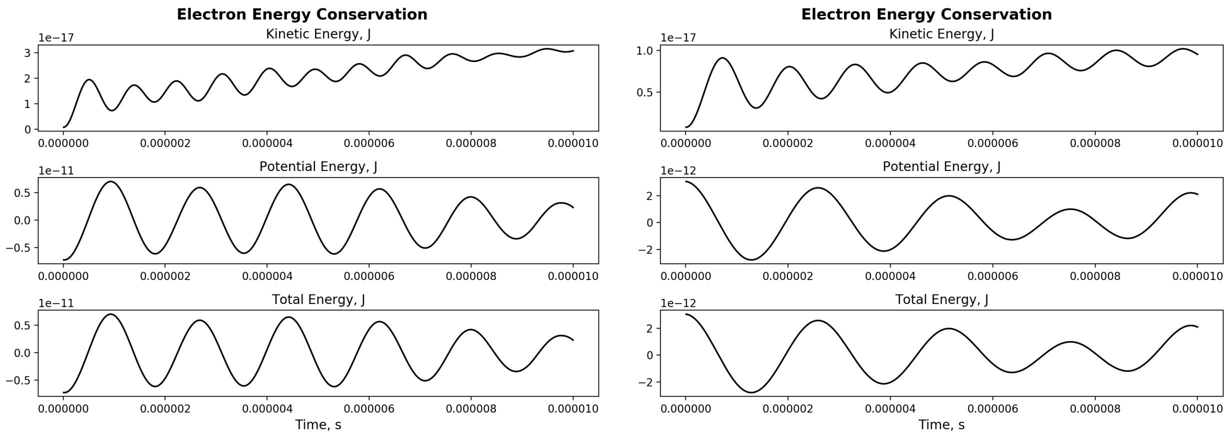


Fig. 6 Energy conservation for electrons in representative simulation output, varying nondimensionalization factor in Poisson's equation solver (left: $1/2$, right: $3/(4\pi)$. Input Conditions: physically realistic electron and proton masses and charges, $T = 0.005\ eV$ (both species), $N_{\text{macro}} = 2000$ (both species), $n = 10^8\ m^{-3}$ (both species), $N_{\text{nodes}} = 200$, $L_{\text{sys}} = 0.5\ m$, $\Delta t = 10^{-8}\ s$, 1000 time steps, periodic boundary

of a node electric field, or the assumption that the net electric field is zero, making it subject to similar errors of system specification as the presently applied method using Poisson's equation. As a result, this method was not implemented for testing, despite the difficulties encountered with the implementation of Poisson's Equation.

References

[1] Hoyt, R. P., "Magnetic nozzle design for high-power MPD thrusters," *Proc. of the 29th Int. Electric Propulsion Conf.(Princeton, NJ)*, 2005, pp. 2005–230.

[2] Birdsall, C. K., and Langdon, A. B., *Plasma Physics via Computer Simulation*, IOP Publishing Ltd., 1991.

[3] Chen, F. F., *Introduction to Plasma Physics and Controlled Fusion*, 2nd ed., Plenum Press, 1984.

[4] Martin, D., "Electrostatic PIC simulation of plasmas in one dimension," 2007.

[5] Forslund, D. W., "Fundamentals of plasma simulation," *Space Science Reviews*, Vol. 42, No. 1, 1985, pp. 3–16. doi: 10.1007/BF00218219, URL `https://doi.org/10.1007/BF00218219`.

[6] Tskhakaya, D., Matyash, K., Schneider, R., and Taccogna, F., "The Particle-In-Cell Method," *Contributions to Plasma Physics*, Vol. 47, No. 8-9, 2007, pp. 563–594. doi:10.1002/ctpp.200710072, URL `https://doi.org/10.1002/ctpp.200710072`.

[7] "Fact Sheet: Magnetoplasmadynamic Thrusters," , 2004. URL `https://www.nasa.gov/centers/glenn/about/fs22grc.html`.

[8] "Lorentz Force Accelerators (LFAs)," , ???? URL `https://alfven.princeton.edu/research/lfa`.

[9] LaPointe, M. R., Strzempkowski, E., and Pencil, E., "High Power MPD Thruster Performance Measurements," *NASA/TM—2004-213226*, 2004. URL `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20040139544.pdf`.

[10] Giannelli, S., and Andrenucci, M., "Azimuthal instability of MPD thruster plasmas and inception of critical regimes," *47th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, 2011. doi:10.2514/6.2011-5887, URL `https://arc.aiaa.org/doi/abs/10.2514/6.2011-5887`.

[11] Paganucci, F., Zuin, M., Agostini, M., Andrenucci, M., Antoni, V., Bagatin, M., Bonomo, F., Cavazzana, R., Franz, P., Marrelli, L., Martin, P., Martines, E., Rossetti, P., Serianni, G., Scarin, P., Signori, M., and Spizzo, G., "MHD instabilities in magneto-plasma-dynamic thrusters," *Plasma Physics and Controlled Fusion*, Vol. 50, No. 12, 2008, p. 124010. doi:10.1088/0741-3335/50/12/124010, URL `https://iopscience.iop.org/article/10.1088/0741-3335/50/12/124010/pdf`.

[12] Martines, E., Paganucci, F., Zuin, M., Bagatin, M., Cavazzana, R., Rossetti, P., Serianni, G., Antoni, V., and Andrenucci, M., "Performance Improvement due to Kink Instability Suppression in MPD Thrusters," *Presented at the 29th International Electric Propulsion Conference, Princeton University*, 2005.

[13] Myers, R. M., "Applied-field MPD thruster performance with hydrogen and argon propellants," *Journal of Propulsion and Power*, Vol. 9, No. 5, 1993, pp. 781–784. doi:10.2514/3.23691, URL `https://doi.org/10.2514/3.23691`.

[14] Blandòn, J. S., Grisales, J. P., and Riascos, H., "Electrostatic plasma simulation by Particle-In-Cell method using ANACONDA package," *Journal of Physics: Conference Series*, Vol. 850, 2017, p. 012007. doi:10.1088/1742-6596/850/1/012007.

Appendix: PIC Code (`AA244B_Project.jl`)

```
#=
AA244B Project - 1D PIC Code
Jeff Robinson - jbrobin@stanford.edu
=#

using PyPlot
using Distributions
using LinearAlgebra
using Random
using DataFrames
using GLM
using FFTW

## CONSTANTS ##
kb = 1.3806485279*10^-23 #J/K
eps0 = 8.854187812813*10^-12 # F/m
# kc = 8987551787.3681764 # 1/(4*pi*eps0) # N-m^2/C^2
e = 1.602176634*10^-19 # C
# kce = 1.4399645165009467812752224*10^-9 # N-m^2/C
me = 9.109383701528*10^-31 # kg
mp = 1.6726219236951*10^-27 # kg
mpme = 1836.152673439956 # mp/me

mutable struct PIC_particle_species
    name::String
    q::Float64 # -> charge in units of e
    m::Float64 # -> mass in units of me
    xs::Array{Float64, 1} # -> positions in units of dx
    vs::Array{Float64, 1} # -> velocities in units of dx/dt
    vs_old::Array{Float64, 1} # -> old velocities in units of dx/dt
    Es::Array{Float64, 1} # -> electric field magnitude in units of e/eps0*dx^2
    phis::Array{Float64, 1} # -> electric potential in units of e/eps0*dx
end

mutable struct PIC_node_list
    Xs::Array{Float64, 1} # -> locations in units of dx
    charges::Array{Float64, 1} # -> net charges in units of e
    phis::Array{Float64, 1} # -> electric potential in units of e/eps0*dx
    Es::Array{Float64, 1} # -> electric field magnitude in units of e/eps0*dx^2
end

function make_particles(names, NPs, qs, ms)
    particle_list = Array{PIC_particle_species, 1}()
    N_species = length(names)
    for i in 1:N_species
        particle = PIC_particle_species(
            names[i],
            qs[i],
            ms[i],
            zeros(NPs[i]), # positions
            zeros(NPs[i]), # velocities
            zeros(NPs[i]), # old velocities
            zeros(NPs[i]), # Fields
            zeros(NPs[i]) # potentials
            )
        push!(particle_list, particle)
    end
    return particle_list, N_species
end

function make_nodes(N_nodes)
    node_list = PIC_node_list(
        [x for x in 0:N_nodes-1], # Xs
        zeros(N_nodes), # charges
        zeros(N_nodes), # phis
        zeros(N_nodes) # Es
        )
    return node_list
end

#= BIRDSALL & LANGDON EQ 2-6 (1-2) =#
function update_node_charge!(particle_list, node_list, N_nodes, BC)
    node_list.charges = zeros(N_nodes)

    if BC == "zero"
        for particle_spec in particle_list
            for i = 1:length(particle_spec.xs)
                if particle_spec.xs[i] < 0 || particle_spec.xs[i] > node_list.Xs[end]
                    continue
                end
                node_idx_lo = floor(Int64, particle_spec.xs[i]) + 1

                node_list.charges[node_idx_lo] += particle_spec.q * (node_idx_lo - particle_spec.xs[i])
                node_list.charges[node_idx_lo + 1] += particle_spec.q * (1 - (node_idx_lo - particle_spec.xs[i]))
            end
        end

    elseif BC == "sheath"
        for particle_spec in particle_list
            for i = 1:length(particle_spec.xs)
                if particle_spec.xs[i] <= 0
                    node_list.charges[1] += particle_spec.q
                    continue
                elseif particle_spec.xs[i] >= node_list.Xs[end]
                    node_list.charges[end] += particle_spec.q
                    continue
                end
                node_idx_lo = floor(Int64, particle_spec.xs[i]) + 1

                node_list.charges[node_idx_lo] += particle_spec.q * (node_idx_lo - particle_spec.xs[i])
                node_list.charges[node_idx_lo + 1] += particle_spec.q * (1 - (node_idx_lo - particle_spec.xs[i]))
            end
        end

    elseif BC == "periodic"
```

```julia
        for particle_spec in particle_list
            for i = 1:length(particle_spec.xs)
                node_idx_lo = mod(floor(Int64, particle_spec.xs[i]), N_nodes)+1
                node_idx_hi = mod(node_idx_lo, N_nodes)+1

                node_list.charges[node_idx_lo] += particle_spec.q * (node_idx_lo - particle_spec.xs[i])
                node_list.charges[node_idx_hi] += particle_spec.q * (1 - (node_idx_lo - particle_spec.xs[i]))

            end
        end

    end
end

#= BIRDSALL & LANGDON EQ 2-5 (5-6) =#
function update_node_phi!(particle_list, node_list, N_nodes, dx, BC)
    # A = zeros(N_nodes, N_nodes)
    # if BC == "periodic"
        # for i = 1:N_nodes
        #     A[i, i] = -2
        #     A[mod(i, N_nodes)+1, i] = 1
        #     A[i, mod(i, N_nodes)+1] = 1
        # end
        # A = Symmetric(A) # LinearAlgebra.jl optimization for Symmetric matrices
        # Symmetric A in this form may cause numerically unstable solution

    # end

    #= Birdsall & Langdon Appendix D, EQ. (3) & (5) =#
    if BC == "periodic"
        max_idx = N_nodes - 1
    else
        max_idx = N_nodes - 2
    end
    A = zeros(max_idx, max_idx)
    for i = 1:max_idx
        A[i, i] = -2
        A[mod(i, max_idx)+1, i] = 1
        A[i, mod(i, max_idx)+1] = 1
    end
    A = Tridiagonal(A)
    if BC == "periodic"
        node_list.phis[1] = 0.0
        node_list.phis[2:end] = A \ (-node_list.charges[2:end]/2)
    elseif BC == "zero"
        node_list.phis[1], node_list.phis[end] = 0.0, 0.0
        node_list.phis[2:end-1] = A \ (-node_list.charges[2:end-1]/2)
    elseif BC == "sheath"
        node_list.phis[1], node_list.phis[end] = -1.0 * (eps0*dx)/e, -1.0 * (eps0*dx)/e
        node_list.charges[2] -= node_list.phis[1]
        node_list.charges[end-1] -= node_list.phis[end]
        node_list.phis[2:end-1] = A \ (-node_list.charges[2:end-1]/2)
    end
end

#= BIRDSALL & LANGDON EQ 2-5 (4) =#
function update_node_E!(node_list, N_nodes, BC)
    M = zeros(N_nodes, N_nodes)
    for i = 1:N_nodes
        M[mod(i, N_nodes)+1, i] = 0.5
        M[i, mod(i, N_nodes)+1] = -0.5
    end
    if BC == "zero" || BC == "sheath"
        M = Tridiagonal(M)
        M[1, 1] = 1
        M[1, 2] = -1
        M[N_nodes, N_nodes-1] = 1
        M[N_nodes, N_nodes] = -1
    end
    node_list.Es = M * node_list.phis
end

#= BIRDSALL & LANGDON EQ 2-6 (3) =#
function update_particle_Es_phis!(particle_list, node_list, N_nodes, BC)
    if BC == "zero" || BC == "sheath"
        for particle_spec in particle_list
            for i = 1:length(particle_spec.xs)
                if particle_spec.xs[i] < 0 || particle_spec.xs[i] > node_list.Xs[end]
                    particle_spec.Es[i] = 0
                    particle_spec.phis[i] = 0
                    continue
                end
                node_idx_lo = floor(Int64, particle_spec.xs[i]) + 1

                particle_spec.Es[i] = (node_idx_lo - particle_spec.xs[i]) * node_list.Es[node_idx_lo] + (1 - (node_idx_lo - particle_spec.xs[i])) * node_list.Es[node_idx_lo + 1]

                particle_spec.phis[i] = (node_idx_lo - particle_spec.xs[i]) * node_list.phis[node_idx_lo] + (1 - (node_idx_lo - particle_spec.xs[i])) * node_list.phis[node_idx_lo
        + 1]
            end
        end

    elseif BC == "periodic"
        for particle_spec in particle_list
            for i = 1:length(particle_spec.xs)
                node_idx_lo = mod(floor(Int64, particle_spec.xs[i]), N_nodes)+1
                node_idx_hi = mod(node_idx_lo, N_nodes)+1

                particle_spec.Es[i] = (node_idx_lo - particle_spec.xs[i]) * node_list.Es[node_idx_lo] + (1 - (node_idx_lo - particle_spec.xs[i])) * node_list.Es[node_idx_hi]

                particle_spec.phis[i] = (node_idx_lo - particle_spec.xs[i]) * node_list.phis[node_idx_lo] + (1 - (node_idx_lo - particle_spec.xs[i])) * node_list.phis[node_idx_hi
        ]
            end
        end

    end
end

```

```
#= BIRDSALL & LANGDON EQ 3-5 (3) =#
function update_particle_vs!(particle_list, dx, dt)
    for particle_spec in particle_list
        particle_spec.vs_old = particle_spec.vs
        particle_spec.vs .+= particle_spec.q/particle_spec.m*particle_spec.Es*dt
        # particle_spec.vs .+= particle_spec.q/particle_spec.m*particle_spec.Es * e^2/(me*eps0) * dt^2/dx^3
    end
end

#= BIRDSALL & LANGDON EQ 3-5 (4) =#
function update_particle_xs!(particle_list, node_list, N_nodes, BC)
    for particle_spec in particle_list
        particle_spec.xs .+= particle_spec.vs
        if BC == "periodic"
            particle_spec.xs = mod.(particle_spec.xs, N_nodes)
        end
    end
end

#= BIRDSALL & LANGDON 3-10 =#
# function init_particle_vs!(particle_list, node_list, N_nodes, BC, dx, dt)
#     update_node_charge!(particle_list, node_list, N_nodes, BC)
#     update_node_phi!(particle_list, node_list, N_nodes, dx, BC)
#     update_node_E!(node_list, N_nodes, BC)
#     update_particle_Es_phis!(particle_list, node_list, N_nodes, BC)
#     for particle_spec in particle_list
#         particle_spec.vs .+= (-particle_spec.q/2)/particle_spec.m*particle_spec.Es*dt
#         # particle_spec.vs .+= (particle_spec.q * e)/(2*particle_spec.m * me)*(particle_spec.Es * e/(eps0*dx^2)) * dt*dt/dx
#     end
# end

function run_PIC(;
    names = ["electrons", "protons"],
    qs = [-1, 1],
    ms = [1, mpme],
    # ms = [1, 10],
    temps = [0.005, 0.005], # eV
    drifts = [0.0, 0.0], # m/s
    n = [10^8, 10^8], #10^8
    NPs = [1000, 1000],
    L_sys = 0.5, # m
    dt = 1e-8, # s
    BC = "periodic",
    # BC = "zero",
    # BC = "sheath",
    # pos_IC = "uniform_exact",
    # pos_IC = "uniform_exact_1percentshuffle",
    pos_IC = "uniform_rand",
    N_steps_max = 1000,
    N_steps_save = 10,
    plotting = false
    )

    N_nodes = round(Int64, maximum(NPs)/10) # 10 particles per cell
    node_list = make_nodes(N_nodes)

    N_real_per_macro = n ./ NPs * L_sys
    qs .*= N_real_per_macro
    ms .*= N_real_per_macro
    particle_list, N_species = make_particles(names, NPs, qs, ms)

    if BC == "zero" || BC == "sheath"
        dx = L_sys/(N_nodes-1)
    elseif BC == "periodic"
        dx = L_sys/N_nodes
    end

    vel_dist_variance = zeros(N_species)
    for k = 1:N_species
        if pos_IC == "uniform_rand"
            particle_list[k].xs = rand(Uniform(0, L_sys/dx), NPs[k])
        elseif pos_IC=="uniform_exact" || pos_IC=="uniform_exact_1percentshuffle"
            particle_list[k].xs = [L_sys/(NPs[k]*dx)*(i + k/N_species) for i = 0:NPs[k]-1]
            if pos_IC == "uniform_exact_1percentshuffle"
                particle_list[k].xs .+= .01*L_sys/(NPs[k]*dx)*(rand(RandomDevice(), NPs[k]).-0.5)
            end
        end
        particle_list[k].xs = mod.(particle_list[k].xs, N_nodes) # start in domain regardless of boundary

        vel_dist_variance[k] = sqrt(temps[k]*e / (me*particle_list[k].m))
        particle_list[k].vs = rand(Normal(drifts[k], vel_dist_variance[k]), NPs[k]) * dt/dx
        shuffle!(RandomDevice(), particle_list[k].vs) # eliminate correlations
    end
    # init_particle_vs!(particle_list, node_list, N_nodes, BC, dx, dt)

    #= SIMULATION STATISTICS =#
    debye_real = sqrt(eps0*temps[1]/(n[1]*e))
    debye_macro = sqrt(eps0*temps[1]/(length(particle_list[1].xs)/L_sys*e*abs(particle_list[1].q)))
    wp_real = sqrt(n[1]*e^2/(eps0*me))
    wp_macro = sqrt(length(particle_list[1].xs)/L_sys*(e*particle_list[1].q)^2/(eps0*me*particle_list[1].m))

    inputs_string =
    """
    INPUT CONDITIONS
                     Particle types: $(names)
                   Particle charges: $(qs./N_real_per_macro) e
                    Particle masses: $(ms./N_real_per_macro) me
                       Temperatures: $(temps) eV
                   Drift Velocities: $(drifts) m/s
           Number of macroparticles: $(NPs)
                   Number Density n: $(n)
    Real particles per macroparticle: $(N_real_per_macro)
                         Node Count: $(N_nodes)
                        System size: $(L_sys) m
               Grid node spacing dx: $(dx) m
                  Real Debye Length: $(debye_real) m
                 Macro Debye Length: $(debye_macro) m
           dx per Real Debye Length: $(debye_real/dx)
              Real Plasma Frequency: $(wp_real)
             Macro Plasma Frequency: $(wp_macro)
                       Time step dt: $(dt)
    Time steps / plasma oscillation: $(1/(dt*wp_real))
     Number of time steps simulated: $(N_steps_max)
                 Boundary Condition: $(BC)
         Initial Position Condition: $(pos_IC)
   """
   println(inputs_string)
   savefile = open("PIC Output/_Simulation_Info.txt", "w")
   write(savefile, inputs_string)

   #= ENERGY CONSERVATION TRACKING =#
   KE_spec = [Array{Float64, 1}(undef, N_steps_max) for i=1:N_species]
   PE_spec = [Array{Float64, 1}(undef, N_steps_max) for i=1:N_species]
   TE_spec = [Array{Float64, 1}(undef, N_steps_max) for i=1:N_species]
   KEs = Array{Float64, 1}(undef, N_steps_max)
   PEs = Array{Float64, 1}(undef, N_steps_max)
   TEs = Array{Float64, 1}(undef, N_steps_max)
   # PE_alt = Array{Float64, 1}(undef, N_steps_max) # from grid nodes

   #= MOVING AVERAGES =#
   moving_avg_x_log = [zeros(length(particle_list[i].xs)) for i=1:N_species]
   moving_avg_v_log = [zeros(length(particle_list[i].xs)) for i=1:N_species]
   moving_avg_phi_log = zeros(N_nodes)
   moving_avg_E_log = zeros(N_nodes)
   N_steps_avg = max(round(Int64, N_steps_save/10), 1)
   # N_steps_avg = 10

   #= VELOCITY DISTRIBUTION TRACKING =#
   # mean, variance, mean error, variance error
   vel_dist_params = [[[] for j = 1:4] for i=1:N_species]

   #= PARTICLE POSITION TRACKING FOR FFT =#
   # particle_pos = [Array{Float64, 1}(undef, N_steps_max) for i=1:NPs[1]]
   # fft_freqs = [i*1/dt/N_steps_max for i = 0:N_steps_max-1]

   step_idx = 0
   fig_idx = 3
   println("Progress:")
   #= MAIN SIMULATION LOOP =#
   while step_idx < N_steps_max
       update_node_charge!(particle_list, node_list, N_nodes, BC)
       update_node_phi!(particle_list, node_list, N_nodes, dx, BC)
       update_node_E!(node_list, N_nodes, BC)
       update_particle_Es_phis!(particle_list, node_list, N_nodes, BC)
       if step_idx == 0
           update_particle_vs!(particle_list, dx, -dt/2)
       end
       update_particle_vs!(particle_list, dx, dt)
       update_particle_xs!(particle_list, node_list, N_nodes, BC)
       step_idx += 1
       if mod(step_idx, round(min(N_steps_max/10, 100))) == 0 # TRACK PROGRESS
           print("\e[2K") # clear line
           print("\e[1G") # move cursor to first column
           print(step_idx," / ",N_steps_max)
           if step_idx == N_steps_max
               print("\n\n")
           end
       end

       #= PARTICLE POSITION TRACKING FOR FFT =#
       # for i = 1:NPs[1]
       #     particle_pos[i][step_idx] = particle_list[1].xs[i] * dx
       # end

       #= ENERGY CONSERVATION TRACKING =#
       for k in 1:N_species
           #= Kinetic Energy m*Vold*Vnew/2 =#
           KE_spec[k][step_idx] = sum(0.5 * dx/dt * dx/dt * me * particle_list[k].vs .* particle_list[k].vs_old * particle_list[k].m)
           #= Potential Energy q =#
           PE_spec[k][step_idx] = sum(particle_list[k].phis * e/(eps0*dx) * particle_list[k].q * e)
           #= Total Energy =#
           TE_spec[k][step_idx] = KE_spec[k][step_idx] + PE_spec[k][step_idx]

           #= MOVING AVERAGES FOR PLOTTING =#
           if mod(step_idx, N_steps_save) >= N_steps_save-N_steps_avg
               moving_avg_x_log[k] .+= particle_list[k].xs * dx
               moving_avg_v_log[k] .+= particle_list[k].vs * dx/dt
           end

           #= VELOCITY DISTRIBUTION =#
           if mod(step_idx, N_steps_save) == 0
               vel_dist = params(fit(Normal, moving_avg_v_log[k]/N_steps_avg))
               push!(vel_dist_params[k][1], vel_dist[1])
               push!(vel_dist_params[k][2], vel_dist[2])
               push!(vel_dist_params[k][3], (vel_dist[1])/vel_dist_variance[k])
               push!(vel_dist_params[k][4], (vel_dist[2] - vel_dist_variance[k])/vel_dist_variance[k])
           end
       end
       KEs[step_idx] = sum([KE_spec[i][step_idx] for i=1:N_species])
       PEs[step_idx] = sum([PE_spec[i][step_idx] for i=1:N_species])
       TEs[step_idx] = sum([TE_spec[i][step_idx] for i=1:N_species])
       # PE_alt[step_idx] = sum(node_list.phis * e/(eps0*dx) * node.charge * e)

       #= MOVING AVERAGES FOR PLOTTING =#
       if mod(step_idx, N_steps_save) >= N_steps_save-N_steps_avg
           moving_avg_phi_log .+= node_list.phis * e/(eps0*dx)
           moving_avg_E_log .+= node_list.Es * e/(eps0*dx^2)
       end

       #= PHASE SPACE & NODE PLOTS =#
       if plotting == true || step_idx == 1
           if step_idx == 1 || mod(step_idx, N_steps_save) == 0
               fig_idx += 1
               # fig1 = figure(fig_idx)
               fig1 = figure(0)
               fig1.clf()
               (ax1, ax2, ax3, ax4) = fig1.subplots(nrows = 2, ncols = 2)
               # ax1.set_xlabel("Particle Position, m")
               ax1.set_ylabel("Particle Velocity, m/s", fontsize=8)
               ax2.set_xlabel("Particle Position, m", fontsize=8)
               ax1.set_xlim((node_list.Xs[1]*dx, (node_list.Xs[end]+1)*dx))
               ax2.set_xlim((node_list.Xs[1]*dx, (node_list.Xs[end]+1)*dx))
               ax2.set_ylabel("Particle Velocity, m/s", fontsize=8)
               # ax3.set_xlabel("Node Position, m")
               ax3.set_ylabel("Node Potential, V", fontsize=8)
               ax4.set_xlabel("Node Position, m", fontsize=8)
               ax4.set_ylabel("Node Electric Field, V/m", fontsize=8)
               # Plot phase space representation of particles
               if step_idx == 1
                   ax1.scatter(particle_list[1].xs * dx,
                               particle_list[1].vs * dx/dt)
                   ax2.scatter(particle_list[2].xs * dx,
                               particle_list[2].vs * dx/dt)
               else
                   moving_avg_x_log /= N_steps_avg
                   moving_avg_v_log /= N_steps_avg
                   ax1.scatter(moving_avg_x_log[1], moving_avg_v_log[1])
                   ax2.scatter(moving_avg_x_log[2], moving_avg_v_log[2])
               end
               # Plot grid potential & field
               if step_idx == 1
                   node_phis = node_list.phis * e/(eps0*dx)
                   node_Es = node_list.Es * e/(eps0*dx^2)
                   ax3.plot(node_list.Xs, node_phis)
                   ax4.plot(node_list.Xs, node_Es)
               else
                   moving_avg_phi_log /= N_steps_avg
                   moving_avg_E_log /= N_steps_avg
                   ax3.plot(node_list.Xs, moving_avg_phi_log)
                   ax4.plot(node_list.Xs, moving_avg_E_log)
               end
               fig1.set_size_inches(10, 8)
               fig1.tight_layout()
               fig1.savefig("PIC Output/fig_$(fig_idx-3).png", dpi=50)

               moving_avg_x_log = [zeros(length(particle_list[i].xs)) for i=1:N_species]
               moving_avg_v_log = [zeros(length(particle_list[i].xs)) for i=1:N_species]
               moving_avg_phi_log = zeros(N_nodes)
               moving_avg_E_log = zeros(N_nodes)
           end
       end
   end
   close(fig1)

   times = [i*dt for i = 1:N_steps_max]

   #= TOTAL ENERGY LINEAR FIT =#
   TE_DF = DataFrame(times = times, TEs = TEs)
   TE_lm = lm(@formula(TEs ~ times), TE_DF)
   TE_coeffs = coef(TE_lm)
   TE_fit = TE_coeffs[1] .+ TE_coeffs[2]*times

   #= ENERGY CONSERVATION PLOTS =#
   fig1 = figure(0)
   fig1.clf()
   fig1.set_size_inches(8, 6)
   fig2 = figure(1)
   fig2.clf()
   fig2.set_size_inches(8, 6)
   fig3 = figure(2)
   fig3.clf()
   fig3.set_size_inches(8, 6)
   (ax1, ax2, ax3) = fig1.subplots(nrows = 3, ncols = 1)
   (ax4, ax5, ax6) = fig2.subplots(nrows = 3, ncols = 1)
   (ax7, ax8, ax9) = fig3.subplots(nrows = 3, ncols = 1)
   fig1.suptitle("Electron Energy Conservation",fontsize=14,fontweight="bold")
   ax1.set_title("Kinetic Energy, J", fontsize=12)
   ax2.set_title("Potential Energy, J", fontsize=12)
   ax3.set_title("Total Energy, J", fontsize=12)
   ax3.set_xlabel("Time, s", fontsize=12)
   fig2.suptitle("Proton Energy Conservation",fontsize=10,fontweight="bold")
   ax4.set_title("Kinetic Energy, J", fontsize=12)
   ax5.set_title("Potential Energy, J", fontsize=12)
   ax6.set_title("Total Energy, J", fontsize=12)
   ax6.set_xlabel("Time, s", fontsize=12)
   fig3.suptitle("Total Energy Conservation",fontsize=10,fontweight="bold")
   ax7.set_title("Kinetic Energy, J", fontsize=12)
   ax8.set_title("Electric Potential Energy, J", fontsize=12)
   ax9.set_title("Total Energy, J", fontsize=12)
   ax9.set_xlabel("Time, s", fontsize=12)
   ax1.plot(times, KE_spec[1],
       color = (0,0,0),
       linestyle = "-",
       label = particle_list[1].name)
   ax2.plot(times, PE_spec[1],
       color = (0,0,0),
       linestyle = "-",
       label = particle_list[1].name)
   ax3.plot(times, TE_spec[1],
       color = (0,0,0),
       linestyle = "-",
       label = particle_list[1].name)
   ax4.plot(times, KE_spec[2],
       color = (0,0,0),
       linestyle = "-",
       label = particle_list[2].name)
   ax5.plot(times, PE_spec[2],
       color = (0,0,0),
       linestyle = "-",
       label = particle_list[2].name)
   ax6.plot(times, TE_spec[2],
       color = (0,0,0),
       linestyle = "-",
       label = particle_list[2].name)
   ax7.plot(times, KEs,
       color = (0,0,0),
       linestyle = "-")
   ax8.plot(times, PEs,
       color = (0,0,0),
       linestyle = "-")
   # ax8.plot(times, PE_alt,
   #     color = (0.5,0.5,0.5),
   #     linestyle = "-")
   ax9.plot(times, TEs,
       color = (0,0,0),
       linestyle = "-")
   ax9.plot(times, TE_fit, # TE LINEAR FIT
       color = (0.5,0.5,0.5),
       linestyle = ":")
   fig1.tight_layout(rect=[0, 0, 1, 0.96])
   fig2.tight_layout(rect=[0, 0, 1, 0.96])
   fig3.tight_layout(rect=[0, 0, 1, 0.96])
   fig1.savefig("PIC Output/Energy Conservation Electrons.png", dpi=200)
   fig2.savefig("PIC Output/Energy Conservation Protons.png", dpi=200)
   fig3.savefig("PIC Output/Energy Conservation Total.png", dpi=200)
   close(fig1)
   close(fig2)
   close(fig3)

   max_TE = max(maximum(TEs)-TEs[1], TEs[1]-minimum(TEs))/TEs[1]*100
   RMS_TE = sum(((TEs.-TEs[1])/TEs[1]).^2/N_steps_max)*100
   RMS_vel_mu_err = zeros(N_species)
   RMS_vel_var_err = zeros(N_species)
   mean_vel_mu_err = zeros(N_species)
   mean_vel_var_err = zeros(N_species)
   for k in 1:N_species
       RMS_vel_mu_err[k] = sqrt(sum(vel_dist_params[k][3].^2)/length(vel_dist_params[k][3]))
       RMS_vel_var_err[k] = sqrt(sum(vel_dist_params[k][4].^2)/length(vel_dist_params[k][4]))
       mean_vel_mu_err[k] = sum(vel_dist_params[k][3])/length(vel_dist_params[k][3])
       mean_vel_var_err[k] = sum(vel_dist_params[k][4])/length(vel_dist_params[k][4])
   end
   diagnostics_string =
   """
   ENERGY CONSERVATION
          Max Total Energy Error: $(max_TE) %
          RMS Total Energy Error: $(RMS_TE) %
    Last Step Total Energy Error: $((TEs[end]-TEs[1])/TEs[1]*100) %
   Total Energy Linear Fit Slope: $(TE_coeffs[2]/TEs[1]*100*dt) %/dt

   VELOCITY DISTRIBUTION DEVIATION FROM INITIAL
   MEAN (Initial: $(drifts) m/s) - Statistics relative to initial variance
          RMS: $(RMS_vel_mu_err*100) %
         Mean: $(mean_vel_mu_err*100) %
   First Step: $([vel_dist_params[1][3][1]*100, vel_dist_params[2][3][1]*100]) %
    Last Step: $([vel_dist_params[1][3][end]*100, vel_dist_params[2][3][end]*100]) %

   VARIANCE (Initial: $(vel_dist_variance) m/s)
          RMS: $(RMS_vel_var_err*100) %
         Mean: $(mean_vel_var_err*100) %
   First Step: $([vel_dist_params[1][4][1]*100, vel_dist_params[2][4][1]*100]) %
    Last Step: $([vel_dist_params[1][4][end]*100, vel_dist_params[2][4][end]*100]) %
   """
   println(diagnostics_string)
   write(savefile, diagnostics_string)
   close(savefile)

   # particle_fft = [Array{Float64, 1}(undef, N_steps_max) for i=1:NPs[1]]
   # for i = 1:NPs[1]
   #     particle_fft[i] = abs.(fft(particle_pos[i]))
   # end

   # return TEs
end


#= TO DO
   ultra low Density (just reduce particle count)
   ultra high Density
   induce electric field
   add charge clumping
   plot distribution fxn after generation
   make linear fit of energy and plot for each sim
   SPURIOUS ELECTRIC FIELDS
   =#
```