ElectroMagnetic Object-Oriented Particle-in-Cell (EMOOPIC) Code Final Report

January 16, 2017

Alex Glasser, Jeff Lestz, Noah Mandell, Ian Ochs, Yuan Shi, Denis St. Onge

1. Introduction

Plasma dynamics, the dynamics of charged particles in self-consistent electromagnetic fields, is central to the study of many astronomical systems like stars and galaxies, as well as many laboratory devices like fusion reactors and semiconductor processors. The dynamics of plasmas can be very complex, due the the interactions of many degrees of freedoms over a large range of spatial and temporal scales. To study such dynamical systems, numerical simulations are usually necessary. The most comprehensive way of simulating plasma systems is to capture the dynamics on the fastest time scale, and resolve processes on the finest spatial scale. This type of simulation tracks the motion of individual charged particles under the influence of surrounding electromagnetic fields, and evolves the surrounding fields in response to current generated by charged particles. Upon discretizing space and time, point particles move in fields that reside on discrete cells, and therefore, such simulation is termed a “particle-in-cell” (PIC) simulation. Although computationally expensive, PIC simulation is currently the only type of plasma simulation that can capture the phase space dynamics of plasmas at the finest level of detail.

In this project, we have successfully built a functional PIC code that is capable of capturing the relativistic dynamics of charged particles in self-consistent electromagnetic fields (EM) using the object-oriented (OO) language C++. To simulate the dynamics of charged particles, we chose to use the well-established Boris algorithm[1], which conserves single particle phase space volume, and therefore ensures that the global energy error is bounded. To simulate the dynamics of the electromagnetic fields, we chose to use the well-established Yee’s algorithm[2], which respects the geometric structure of Maxwell’s equations, and therefore ensures that the local charge error is bounded. The Boris particle pusher and the Yee’s field evolver are interconnected in our program by our depositor, which deposits current carried by particles (in the continuum) to fields (on the cell), as well as our interpolator, which interpolates fields on the cells back to particles in the continuum. These four core objects of our program are further supported by: (i) a Poisson field initialization routine; (ii) a flexible implementation of boundary conditions; (iii) HDF5-modeled input/output capabilities; and (iv) an accelerating MPI domain-decomposition scheme. Users of our code interact with the program through a self-documenting input file, and are given a number of visualization tools to render the simulation results.

[1] J. Boris, in Proceedings of the Fourth Conference on Numerical Simulation of Plasmas (Naval Research Laboratory, Washington D. C., 1970), p. 3.

[2] Yee, Kane S. "Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media." *IEEE Trans. Antennas Propag* 14.3 (1966): 302-307.

In this report, we will first describe our project design (Sec. 2) and review its development history (Sec. 3). We will then explain details of the current version of our program (Sec. 4), demonstrate its results (Sec. 5), and analyze its performance (Sec. 6). Together with our accomplishments, we have also noticed a number of problems, which we will reflect upon in Sec. 7. Finally, a summary will be given and future work will be discussed.

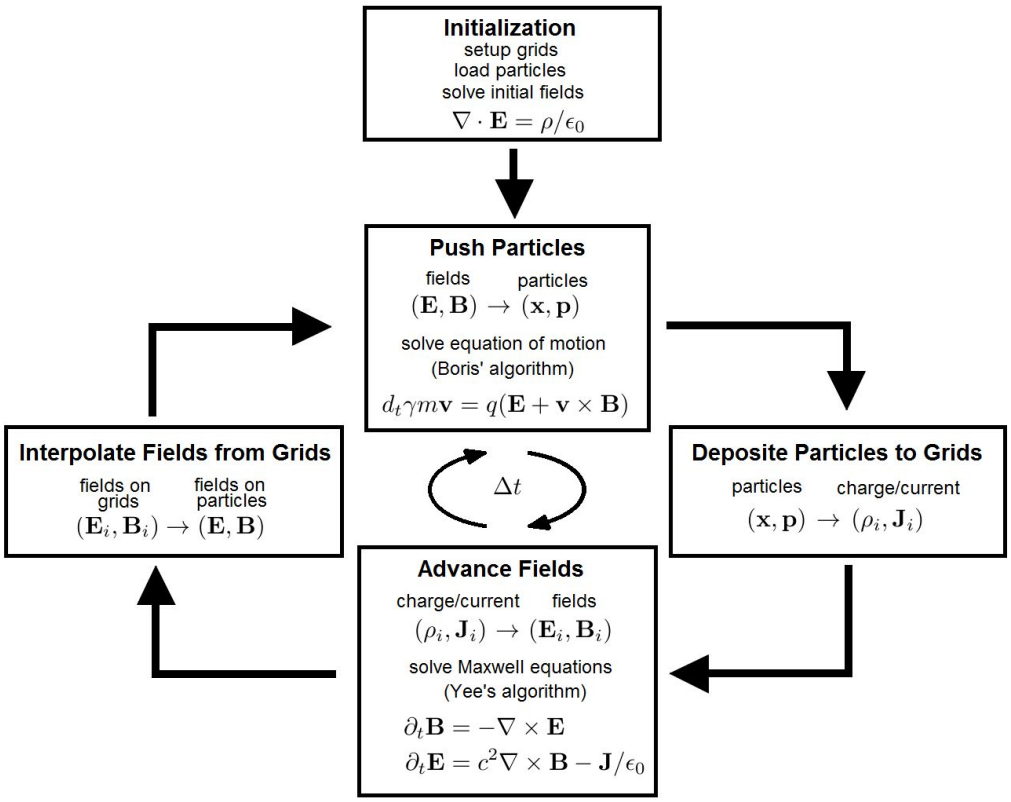


Figure 1: Electromagnetic PIC design. Adapted from Spitkovsky, “Kinetic Plasma Simulations”

2. Project Design

2.1. Architecture and Interface

At a high level, a particle-in-cell code is comprised of two components: (a) the **particles**, discrete chunks of charged matter which are accelerated by fields according to the Lorentz force, and (b) the **grid** (forming the “cells”), on which the electric and magnetic fields and currents are stored, and evolve in time.

The core physics functionality of our code thus lies in two broad suites of classes:

Particle suite:

- **Particle**: Basic info for each particle
- **Particle Handler**: wrapper for particle pushing and field interpolation
- **Pusher**: Lorentz force advancer
- **Interpolator**: Using grid data to derive field values at each particle location
- **Depositor**: Passing current information to grid

Grid suite:

- **Grid**: main class handling field evolution and setting/getting of fields
- **Poisson**: derived class of Grid which additionally implements the initial Poisson solve for fields

In order to make our code MPI-capable, we then added a **Domain** class which would wrap around grid, and enable communication between adjacent domains in a larger space.

As we got closer to testing our code, we realized that in order to explore any interesting physics, we would need to be able to drive the plasma in different ways, for instance by injecting electrostatic or light waves at the boundaries. Thus a suite of **Boundary Condition** code was necessary, and implemented consistently for both particles and fields. A small component of this codebase lies in the grid directory, in the GridBC class, due to its close relationship to the grid class.

Finally, the code has many different input options (particle attributes, number of particles, scale of simulation, number of processors, and boundary conditions, to name a few), and also produces vast amounts of data output. This input and output is handled by the **I/O suite**, which parses and checks configuration file inputs, and outputs results in a storage-efficient HDF5 format.

All of these different components are managed by a single driver program.

To interface with this code, one simply runs:

EMOOPIC <path to input configuration file>

at the command line.
This outputs an HDF5 file describing particle and field evolution, and also optionally outputs tracking information on tracer particles and their trajectories.

Initial design: Grid class, Particle Class…
Actual design: Poisson, Pusher, Boundary...
Possible better design in retrospect: …

2.2. Division of Work

Originally, we did not intend to parallelize the code, and the work was divided into six parts which were thought to be relatively independent:

1) **I/O:** Noah
2) **Driver / MPI:** Yuan
3) **Particles:** Denis

4) **Grid - Field Evolution:** Ian
5) **Grid - Particle Current Deposition:** Jeff
6) **Grid - Field Interpolation:** Alex

The grid class was originally envisioned to be responsible for most interpolation methods, hence the large number of team members assigned to grid.
However, due to the large numbers of particles within a single cell, we soon decided it would be more efficient to pass the information required for interpolation to a particle handler object, which would then use the same interpolation information for the broad set of particles, reducing redundant memory access.
We also realized that for 3D PIC, parallelization was really necessary to run a problem of reasonable size, which necessitated many new methods for ghost cell handling and information passing.
Finally, we decided to add a subclass of Grid which would perform a more complex initialization resulting in fully self-consistent initial electric and magnetic fields, via a solution of the Poisson equation.
Thus our division of labor became:

1) **I/O:** Noah
2) **Driver / Domain:** Yuan
3) **Particles:** Denis
4) **Grid Field Evolution:** Ian
5) **Current Deposition:** Alex (Particle side), Ian (Grid side)
6) **Field Interpolation:** Alex (Particle side), Ian (Grid side)
7) **Grid Initialization / Ghost Cells:** Jeff
8) **Poisson (Grid Subclass):** Jeff, Alex
9) **Boundary Conditions & MPI:** Denis (Particle side), Yuan (Grid side)

In reality the division of labor ended up being less clear cut (for a further discussion, see section on “Lessons Learnt”).

2.3. Software Development

Our distributed working model was facilitated by a heavy reliance on **GitHub**. After ~600 commits, the necessity of such a repository was amply demonstrated. We supplemented this distributed-programming effort with periodic in-person meetings, which were crucial for explaining our work to one another, jointly solving problems created by the subtler aspects of PIC codes, and reprioritizing our development needs.
The **gdb debugger** was used to help suss out bugs in our code, and **Intel VTune Amplifier** was used to profile the code and identify hotspots..
Documentation was generated by the **Doxygen** engine, operating on our annotated C++ code.

2.4. Make vs. Buy

We were initially planning to “buy” a Poisson solver to initialize fields consistent with the particle distribution, but the simplicity of the **Jacobi iteration method** made it more efficient to implement our own.

We did, however, “buy” the C++ **HDF5** library, which allowed for efficient and compact output, along with ease of solving the parallel I/O problem.
We also “bought” the **libconfig** library, which allows for processing of structured text-based configuration files.
Additionally, we relied heavily on the “bought” **googleTest** framework for unit testing.
Finally, random number generation was built around the **Ran2** algorithm from Numerical Recipes. This generator has a large period (greater than 10^{15}) and is used to piggy-pack more complicated random number generation.

3. Version History

3.1. **Prototype (Dec 6)**: The grid class was Yee-evolving its electromagnetic fields and get/setting ghost cells. The particle class was generating particles, interpolating the grid’s field values at particle locations, and Boris-pushing the particles. Code had already begun to be doxygenated. Mere placeholders had been installed for unit-testing, boundary conditions, Poisson initialization and I/O. But the code was compiling successfully.

3.2. **Alpha version (Dec 13)**: An input file was now able to “seed” the code, and boundary conditions had been implemented (for particles). A more robust code for domain decomposition had enabled MPI-passing in one dimension, and particle current-deposition was implemented. Simple executions of the code were now possible, but Poisson initialization remained unimplemented, and HDF5 unintegrated.

3.3. **Current version**:
HDF5 I/O and visualization capability were added, as well as a full complement of boundary condition routines and functionality--for particles and fields, and for periodic and physical boundaries. MPI decomposition was made possible in three dimensions. Poisson initialization of fields was achieved. The code was fully annotated and extensively unit-tested with googleTests.

4. Program Structure

4.1. The current workflow
Currently, our program works in the following way. At the very beginning, MPI is initialized, and the master node reads the input file. After checking that the ***Input*** information is sufficient and self-consistent, the master node then broadcasts this information to all other nodes. Using the received information, each node begins the simulation initialization. The first step of the initialization determines which ***Domain***, partitioned evenly from the entire simulation box, is assigned to a given processor. On each MPI domain, the processor then constructs a ***Particle_Handler*** class, which comprises a collection of ***Particle*** objects, upon which the ***Pusher*** acts. The boundary conditions for particles are constructed by a ***Part_BC_Factory***, which instantiate, at each boundary, an instance of ***BC_Particle*** according to the user’s specification. Also

at this step, the processor constructs a ***Poisson_Solver*** class (subclass of ***Grid***), which handles fields that live on discrete cells. The fields boundary conditions are constructed by a ***Field_BC_Factory***, which instantiates ***BC_Fields*** on all boundaries. Having initialized classes that handle particles and fields, the initial particle distributions in the cell, as well as the initial fields values on the cell, are determined using numerical methods of the corresponding classes. Finally, the class that handles particle output ***OutputBoxQuantities***, and the class that handles the fields output ***Hdf5IO*** are initialized.

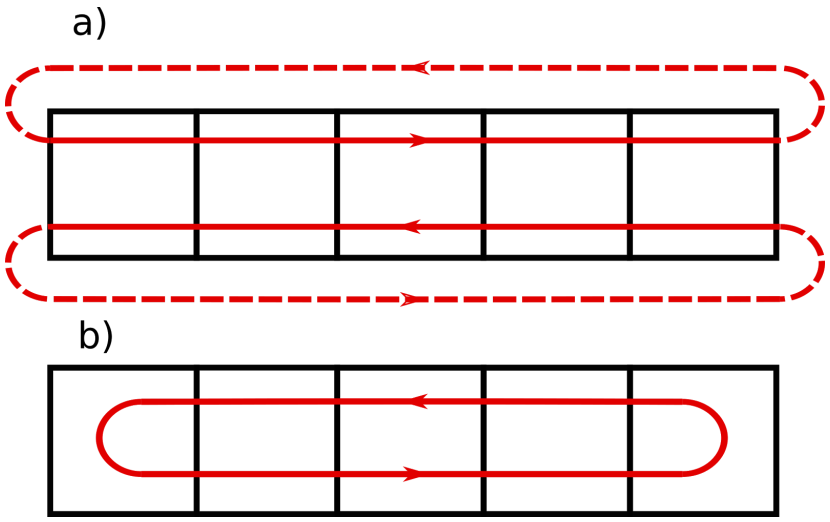


Fig 1 MPI communication topology in one dimension for a) periodic boundary conditions and b) all other boundary conditions. Squares denote processors.

Having finished the initial setup, the program then proceeds to its core time loop, in which the dynamical state of the plasma is advanced in time. This time loop proceeds as follows:

==========

Simulation timestep:

(i) At the beginning of each time step, particles are **pushed** to new positions and velocities.

(ii) The particles are sent and received via **MPI** across **domains**, or are otherwise processed according to the domain's **physical boundaries**. MPI is done via a send right/receive left, then a send left/receive right. Processors at the physical boundaries handle the communication in two different ways, depending on the type of boundary (Fig 1).

(iii) The **deposition** method is then invoked to deposit current carried by charged particles to the edges of cells.

(iv) It is then necessary to sum electric current on **MPI boundaries**, which receives deposition from two MPI domains.
(v) After summations, the fields are **evolved** according to Yee's algorithm.
(vi) Now the ***field*** **boundary conditions** are either passed via MPI implemented as a physical boundary.
(vii) Field values are then **interpolated** to the particles, allowing the timestep to be repeated.
==========
Auxiliary processes performed at the end of each simulation step:
(viii) Data and diagnostic files are written via HDF5 (written for MPI)
(ix) Memory is freed up.
(x) Timing of the timestep is recorded.
==========

4.2. The current classes and their interactions are summarized by the following table.

Class	Brief Description	Interact with
Input	Read and broadcast input file	Used in part by all classes
Domain	Set up MPI domain decomposition, register MPI neighbours	Used in part by all following classes
Particle	Single particle, storing position, velocity, charge, and mass. Also contains local fields on the particle.	Act upon by ***Pusher***, ***Particle_Handler***, and ***Particle_Compare***
Pusher ***Boris*** ***Relativistic_Boris***	Push single particle Non-relativistic pusher Relativistic pusher	Act on ***Particle***
Particle_Handler	Handle a collection of particles	Utilize ***Particle***, ***Pusher***, ***Depositor***, ***Interpolator***, ***Paricle_Compare***, and ***BC_Particle***
Particle_Compare	Sort particles, to improve memory access efficiency	Act on ***Particle***, utilized by ***Particle_Handler***
Part_BC_Factory	A singleton registering particle boundary conditions	Register ***BC_Particle***
BC_Particle ***BC_P_Periodic***	Particle boundary conditions, determine what happens to each particle when it reaches	Registered by ***Part_BC_Factory***.

BC_P_Reflecting ***BC_P_MPI***	physical or MPI domain boundaries.	Utilized by ***Particle_Handler***
Depositor	Deposit current/charge from particles in the continuum to discrete fields	Utilized by ***Particle_Handler***. Read ***Particle*** information and write fields in ***Grid***
Interpolator	Interpolate discrete fields to particles in the continuum	Utilized by ***Particle_Handler***. Read fields in ***Grid*** and write ***Particle***'s local fields.
Grid ***Poisson_Solver***	Grids on which E,B,rho, and J fields are defined. Contains methods to operate on these fields. Subclass allowing the use of Poisson solver to initialize fields. Contains additional phi and A fields.	Interact with particles through ***Depositor*** and ***Interpolator***. Utilize boundary conditions supplied by ***BC_Field***
GridBC ***ElectroStaticBC*** ***LightBC*** ***FieldBC*** ***PoissonBC***	Supply boundary conditions to fields Inject longitudinal Gaussian Pulse Inject transverse Gaussian Pulse Inject sinusoidal waves Provide boundary values	Utilized by ***BC_Field***, and modifies fields in ***Grid***
Field_BC_Factory	A singleton registering fields boundary conditions	Registering ***BC_Field***
BC_Field ***BC_F_Periodic*** ***BC_F_External*** ***BC_F_MPI***	Field boundary conditions. Determine what values should fields take on physical and MPI boundaries.	Registered by ***Field_BC_Factory***. Utilized by ***Grid***
Hdf5IO ***FieldTimeseriesIO***	Writes fields output in Hdf5 format	Write data in ***Grid***
OutputBoxQuantities	Writes particle output in ASCII format	Write data in ***Particle_Handler***
Random_Number_Generator	Provide random numbers for initialization, especially particles	Utilized primarily by ***Particle_Handler***

4.3. Possible better alternatives

In retrospect, a better organization of classes is summarized by Fig. 2. Our original design did not prioritize the use of MPI in the code, and the notion of a "domain" was therefore absent in our prototype. In that respect, the Grid class was overburdened early on.

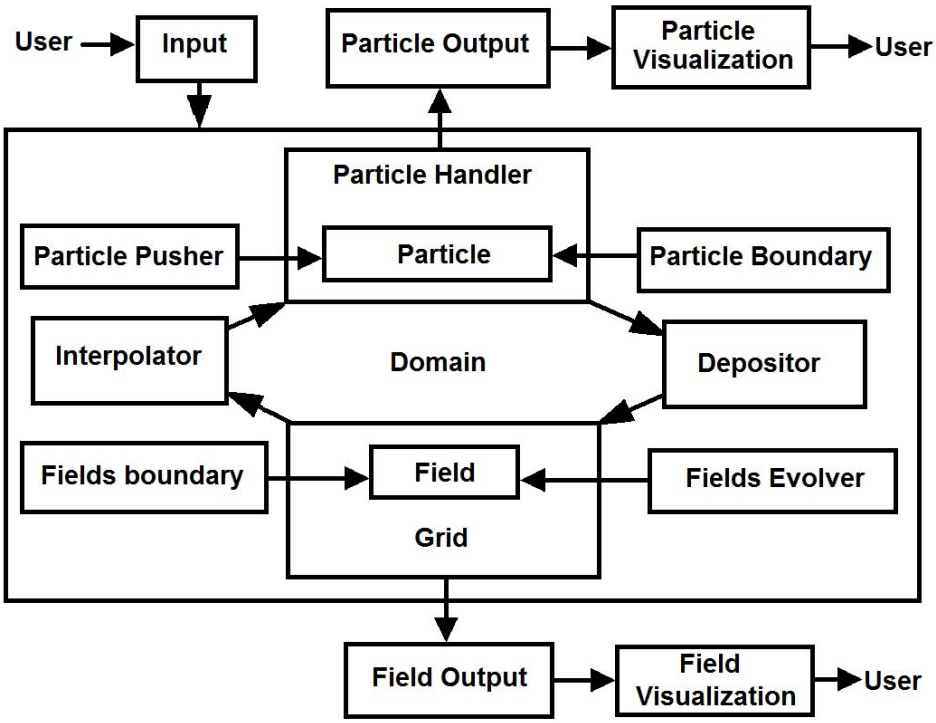


Fig. 2 A possible better organization of classes and their interactions.

5. Simulation Results

5.1. **Single Particle:** data/testcase/SinglePart/ConstEBTest/input.txt To test single particle motion, a single domain was initialized with a constant B field in the z direction, a constant E field in the y direction, and a single electron. It is well known that a charged particle subject to these fields will exhibit a drift in the E x B direction (x direction in this example). Confirmation of this result is shown in Fig. 3. The electron's motion is a combination of gyration around the magnetic field in a right-handed sense, steadily drifting in the x direction. While not shown here, it has also been verified that positive charges correctly gyrate in the opposite direction. This motion is robust for more than 3,000 gyro-orbits (300,000 simulation steps in this case), even when acted on by its own self field. However, the long term behavior diverges from this motion, likely due to outstanding bugs in the code.

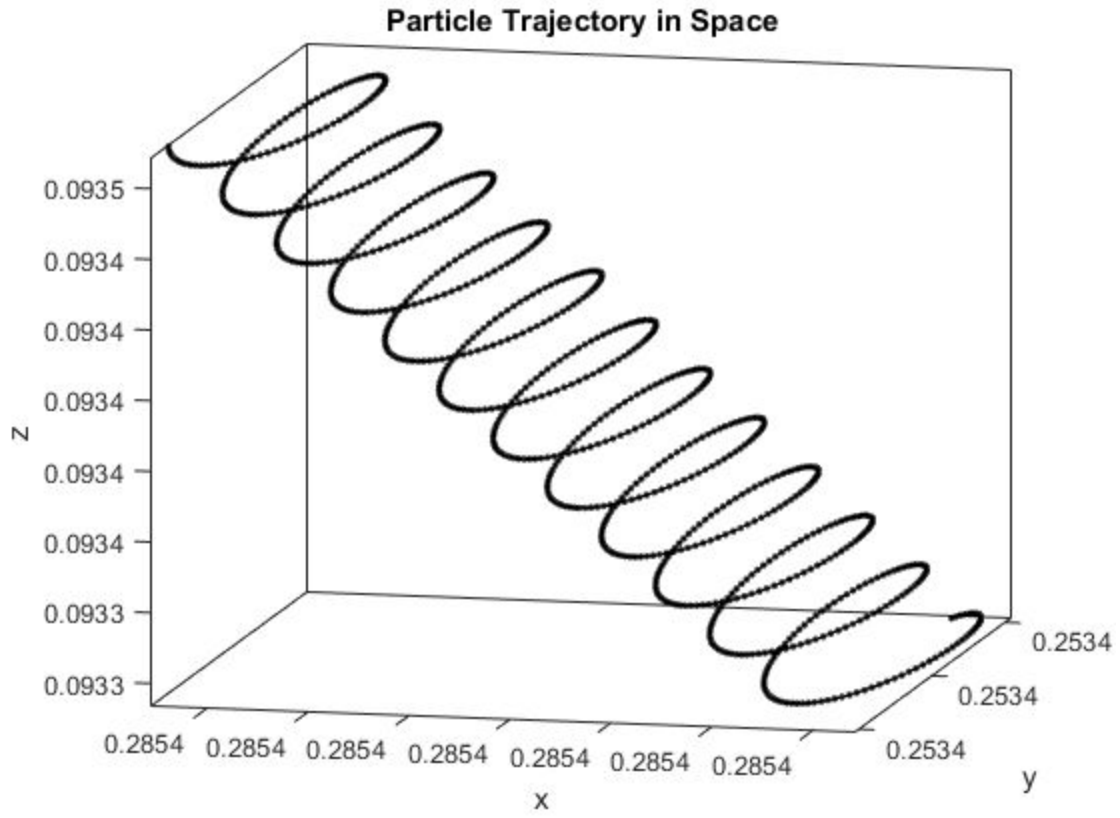


Fig. 3 Single electron's E x B drift in 1cm cube due to constant B field in z direction and constant E field in y direction.

5.2. **Vacuum Wave:** data/testcase/VacuumTest/EMWaveTest/input.txt
To test vacuum electromagnetic wave propagation, a single domain was initialized without particles, and gaussian pulses of light were inserted at the boundary. Results are shown in Fig. 4. The light propagates in the proper direction at the proper speed; however, there are slight numerical anomalies along axes which should be uniform.

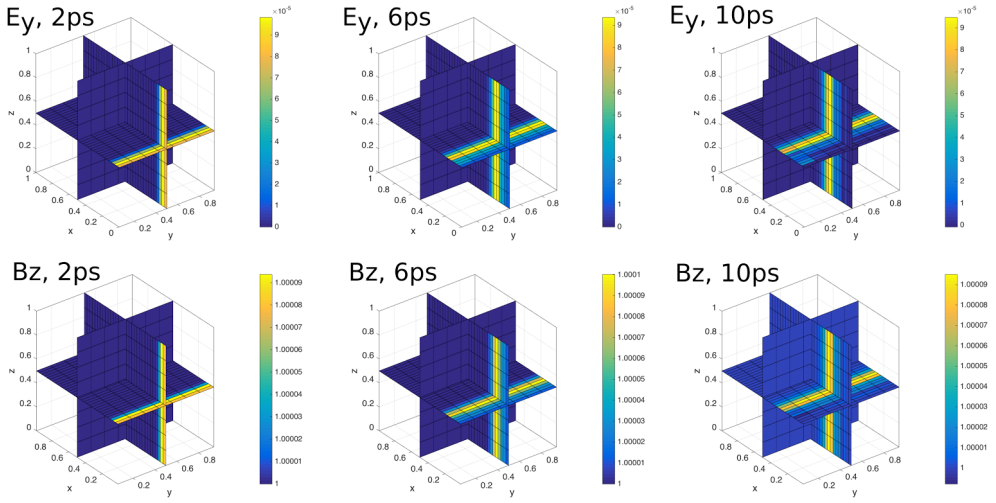


Fig. 4 Driven gaussian EM wave pulse propagating in 1 cm cube over 10 ps. Note that the wave has the proper polarization, and propagates ~0.3 cm in 1e-11 seconds, consistent with the speed of light in vacuum. However, what should be a plane wave has slight non-uniformities in the *y* and *z* directions, which could be due to slight errors in the boundary conditions or machine error (given the large initial values of *Bz*).

5.3. Known problems in simulation

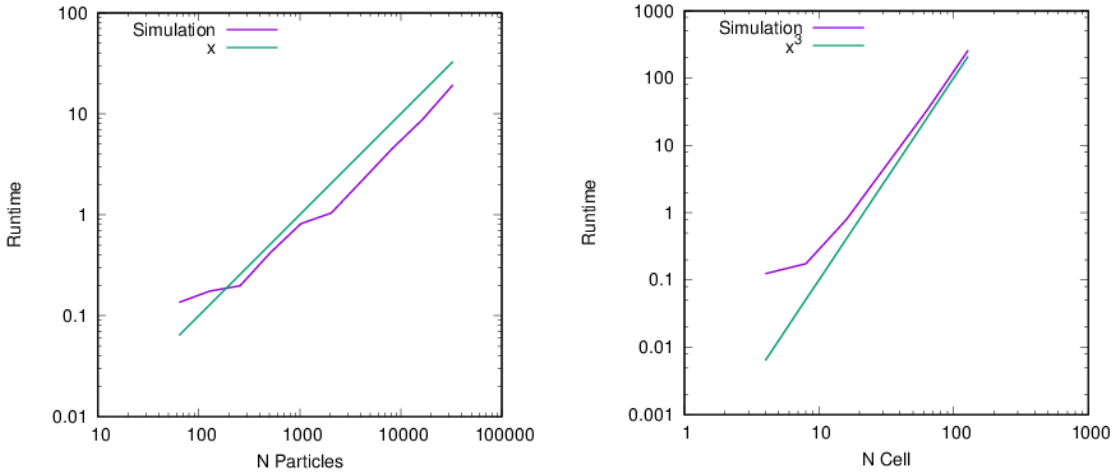
(1) **Poisson initialization + periodic boundaries + MPI:** When the user specifies Poisson initialization and periodic boundary conditions, the simulation requires total charge neutrality to run. This is because the Laplace operator, with periodic (more generally, symmetric) boundary conditions, is a self-adjoint operator. Since the kernel of the Laplace operator contains all constant field solutions (including the vector of all ones), any source field in its image must be orthogonal to these constant fields. Thus, $1^T \rho = 0$. However, the check for total charge neutrality is currently performed on each individual MPI domain. Since the Poisson solution is global, this check must be relaxed to allow Poisson initialization for parallel simulation.

(2) **Energy conservation:** Grid and particle heating appear to be persistent features of EMOOPIC's simulations. Given enough timesteps, a particle will eventually demonstrate non-analytic behavior, at tremendous velocities. Even a vacuum simulation, for which no particle self-field is achievable, demonstrates an eventual blow-up in field energy. These numerical inaccuracies arise for reasons which are well-documented in the PIC code literature, and require careful fixes. The next steps in addressing such inaccuracies would be to implement higher order deposition and interpolation methods, n-point field smoothing, and to add dissipation and collisions in our fields and particles. Beyond such measures, one might entertain the implementation of fully geometric particle pusher and field evolver, or a predictor-corrector scheme to increase stability.

6. Performance Profiling

6.1. Serial version

For the serial scaling studies, we set up a base case with nCell = [8, 8, 8], and nParticles_tot = 64. We then separately increase the number of particles and number of cells and measure the runtime. We see that the scaling is as expected, with the runtime scaling linearly with the number of particles, and linearly with the total number of cells, nCell_x*nCell_y*nCell_z = nCell^3.

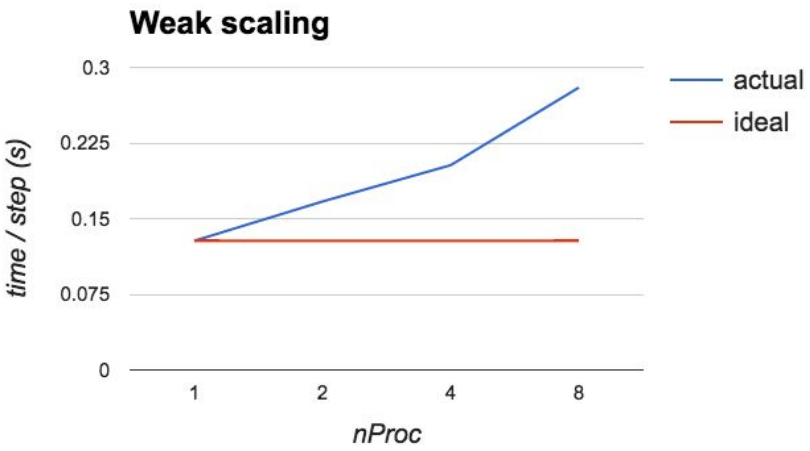


6.2. Parallel version

For parallel scaling studies, we set up a base case with nCell = [128, 128, 128] and nParticles_tot = 50000. We then obtain the time per timestep (in seconds) by dividing the total runtime by the number of timesteps.

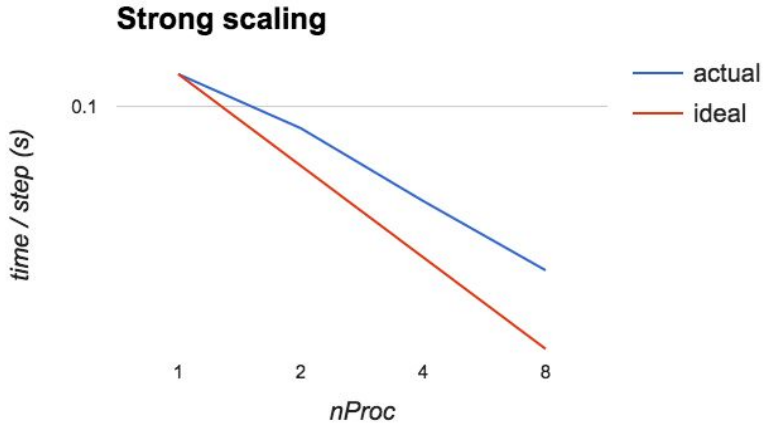
6.2.1. Weak scaling

To measure the weak scaling, we run several cases in which we simultaneously double the number of MPI processes, the number of cells, and the number of total particles. This means that the workload assigned to each process stays constant for all cases. Ideally, this will produce the same runtime across all cases. For EMOOPIC, we see that the weak scaling is not quite ideal.



6.2.2. Strong scaling

To measure strong scaling, we run several cases keeping the base case parameters (and therefore the global problem size) fixed while increasing the number of MPI processes. Ideally, doubling the number of processes will halve the runtime. For EMOOPIC, we see that the actual strong scaling, while not quite ideal, is still rather good. Using 8 MPI processes reduces the runtime by about a factor of 4. This will certainly be useful for speeding up production-level runs.



7. Lessons Learnt

There were many lessons learned during the course of this project, mostly from the challenges that were encountered. The most prominent ones are enumerated below:

1. *Testing must take precedence over features*

   While our code has a large array of features that improve its flexibility and allow it to be applied to many different plasma physics scenarios, it has not been very thoroughly validated, or even passed some key benchmarks at this time. This was due to our

ambition to make this code as robust as possible -- a goal which may not have aligned very well with the scope and timeline of this assignment. While we did write unit tests for the code each of us contributed, they were not exhaustive enough to catch all problems. If we were to restart this project, we would each write stronger unit tests, and as a group we would aim to pass basic validations (energy conservation, wave propagation, single particle motion) after only essential functions had been written. For instance, no MPI, only periodic boundaries for both particles and fields, a single initial condition. Then after passing validation, we could move on to implement the additional features we want, and verify that it can still pass these physics tests. While this design paradigm was covered in the course, our hubris prevented us from making it a top priority, so this lesson had to be learned again from our mistakes.

2. *Early design decisions have lasting consequences*
   This is lesson was addressed in the course, and became more and more apparent as our project progressed. Although the choice of this project over our other ideas was largely due to being comprised of what we thought were six fairly independent modules, these lines blurred as we better understood the low level problems that had to be solved. After selection of the project, we intentionally devoted significant time and effort to writing strong interfaces and forecasting the difficult problems that each of the modules would need to solve. Despite this framework and careful planning, there were some consequences of our design choices that we did not catch at the outset. Most importantly, we have an unsolved problem where the current density and consequent electromagnetic fields diverge as the cell size becomes small. This is a natural consequence of our implementation, but not one that we realized until validation began (far too late, as addressed above). The Grid class is also an area that would benefit from even more thoughtful design. Since the standard field evolution algorithm cleverly has currents and electric fields on cell edges while magnetic fields live on cell faces, this leads to a litany of special edge cases to handle.
3. *Communication is not always effective communication*
   We often had meetings to resolve issues in the code where email communication was insufficient. However, it would have been helpful to follow up each meeting with a detailed document describing the meeting's conclusion, to make sure everyone was actually on the same page.
4. *Barriers between code components can be hard to maintain*
   While our initial approach was to divide the code into several self-contained sections, which would only interact via predetermined interfaces, we found that our forecasts as to which information had to be shared between classes were inaccurate. This failure in forecasting lead to frequent bottlenecks, where team members had to request new interface features on the fly, which often could not be implemented immediately due to the misalignment of different team members' schedules.
5. *When the barriers break, so does the code*
   Often, in an effort to avoid these bottlenecks and hasten development, developers would add implementations to classes which were not under their ownership, or modify existing implementations to return slightly different output types. However, this meant that often a

team member would return to their code to find that it had changed, that their unit tests no longer passed, and that they no longer knew how the code they were responsible for worked. This added confusion made bugs extremely difficult to track down.

8. Summary and Future Work

In November we set out to write an object-oriented particle in cell code to simulate plasmas with the standard Boris and Yee algorithms. Two months later, we have a complete -- albeit not fully validated -- code that consistently incorporates this functionality and more. The code works as both a serial and parallel version, which were written simultaneously and are built from the same source files. In parallel the domain can be decomposed in 3D, which surpasses our initial design of 1D decomposition in a fixed direction. There are multiple options for the boundaries for both fields and particles, also surpassing the original design for simple periodic boundaries.

Overall the code is much more sophisticated than we originally planned, yet this came at a cost. Strong unit testing did not keep pace with the development of the code, especially as specifications morphed on the fly due to unanticipated challenges. Whole simulation validation was an afterthought instead of a high priority alpha version milestone. Prioritizing features over testing was the most costly mistake of the project. Along the way we each learned many intangible lessons from our experiences working on a large coding project with several group members, as well as a deeper appreciation for the low-level challenges of creating a robust PIC code from scratch.

If work were to continue on this code, the top priorities would be writing strong unit tests and then focusing on simple, but demanding, full simulation tests. Once these are complete and the code is satisfactorily validated against known physics, there are a number of new features that could extend the current code. Behind the scenes, the scope of the Grid class could be further reigned in by writing a Field class to encapsulate data and methods for individual fields (especially important for handling edge cases). New dynamic load balancing would improve performance in simulations where particles bunch on certain domains. Inclusion of more ghost cells for higher order finite differencing and a predictor/corrector integration scheme would improve the code's fidelity.

The physics features of the code could also be extended. Introducing a δf particle scheme would increase the signal to noise ratio when simulating problems where the perturbation to the particle distribution function is expected to be small. It could also be beneficial to include an option to subtract off the self field from each particle. While this should not be a large correction, it could still hamper very detailed quantitative testing. A somewhat unorthodox extension to the code would be to include the option to transform it into an n-body gravitational simulation. Due to the similarities between gravitational and electrostatic potentials, this would not require many additions to the code.