LCODE 3D Documentation

I. Kargapolov, K. Lotov, I. Shalimova, A. Sosedkin

May 07, 2019

Contents

Useful links:

- LCODE 3D source: https://github.com/lotov/lcode3d
- LCODE 3D documentation: https://lcode3d.readthedocs.org
- LCODE team website: https://lcode.info
- LCODE team email: team@lcode.info

Note: Please consider using the online version of this document at https://lcode3d.readthedocs.org instead if you are reading this documentation for the first time and you are interested in the implementation details.

Documentation in other formats is not officially supported, is not guaranteed to be complete and is generally provided only for convenience.

If you are interested in not just *what* LCODE does, but also *how* does it do it, and you are reading this documentation for the first time, the recommended procedure is to obtain it in HTML format, glance over the first chapter and then dive straight into the source code, switching back and forth using [source] and [docs] cross-links (available only in HTML version). Trust us, at 500 lines of code it's even shorter than the docs, and if you get stuck, you'll always have a link back to the explanations.

CHAPTER 1

Overview

1.1 Problem

Objective

LCODE 3D calculates the plasma response to an ultrarelativistic charged beam.

Simulating particle beams is definitely planned for the future.

Geometry

Quasistatic approximation is employed, with time-space coordinate $\xi = z - ct$.

From the perspective of the beam, ξ is a space coordinate. The head of the beam corresponds to $\xi = 0$, with its tail extending into *lower, negative* values of ξ.

From the perspective of a plasma layer, penetrated by the beam, ξ is a time coordinate. At $\xi = 0$ the plasma layer is unperturbed; as the beam passes through it, ξ values decrease.

The remaining two coordinates x, y are way more boring [*Simulation window and grids* (page 6)].

The problem geometry is thus ξ, x, y.

Beam

The beam is currently simulated as a charge density function $\rho_b(\xi, x, y)$, and not with particles [*Beam* (page 30)].

Plasma

Only the electron motion is simulated, the ions are represented with a static backround charge density [*Background ions* (page 27)].

$$\begin{aligned}
\frac{d\vec{p}}{d\xi} &= -\frac{q}{1 - v_z}\left(\vec{E} + \left[\vec{v} \times \vec{B}\right]\right) \\
\frac{dx}{d\xi} &= -\frac{v_x}{1 - v_z} \\
\frac{dy}{d\xi} &= -\frac{v_y}{1 - v_z} \\
\vec{v} &= \frac{\vec{p}}{\sqrt{M^2 + p^2}}
\end{aligned}$$

The plasma is simulated using a PIC method with an optional twist: only a 'coarse' grid of plasma (think 1 particle per 9 cells) is stored and evolved, while 'fine' particles (think 4 per cell) are bilinearly interpolated from it during the deposition [*Coarse and fine plasma* (page 22)]. The plasma is effectively made not from independent particles, but from a fabric of 'fine' TSC-2D shaped particles.

Fields

Both the plasma movement and the 'external' beam contribute to the charge density/currents ρ, j_x, j_y, j_z [*Deposition* (page 26)].

The fields are calculated from their derivatives. Theoretically, the equations are

$$\begin{aligned}
\Delta_\perp E_z &= \frac{\partial j_x}{\partial x} - \frac{\partial j_y}{\partial y} \\
\Delta_\perp B_z &= \frac{\partial j_x}{\partial y} - \frac{\partial j_y}{\partial x} \\
\Delta_\perp E_x &= \frac{\partial \rho}{\partial x} - \frac{\partial j_x}{\partial \xi} \\
\Delta_\perp E_y &= \frac{\partial \rho}{\partial y} - \frac{\partial j_y}{\partial \xi} \\
\Delta_\perp B_x &= \frac{\partial j_y}{\partial \xi} - \frac{\partial j_z}{\partial y} \\
\Delta_\perp B_y &= \frac{\partial j_z}{\partial x} - \frac{\partial j_x}{\partial \xi} \\
\Delta_\perp &= \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \\
\rho &= \rho_e + \rho_i + \rho_b \\
j &= j_e + j_i + j_b
\end{aligned}$$

where indices e, i, b represent electrons, ions and beam respectively.

Note: In reality, things are not that simple.

E_z and B_z calculations is relatively straightforward and boils down to solving the Laplace and Neumann equation with Dirichlet boundary conditions respectively.

The transverse fields are actually obtained by solving the Helmholtz equation with mixed boundary conditions, and then doing some more magic on top of that (so refer to *Ez* (page 15), *Ex, Ey, Bx, By* (page 16) and *Bz* (page 19) for the equations that we *really* solve).

Step

The ξ-cycle idea consists of looping these three actions:

- depositing plasma particles (and adding the beam density/current),
- calculating the new fields and
- moving plasma particles,

executed several times for each step in a predictor-corrector scheme [*Looping in xi* (page 31)].

1.2 Trickery index

Geometry

- *Quasistatic approximation* (page 3) reduces the problem dimensionality.

Todo: DOCS: write a separate page on the topic or link somewhere from the overview.

Numerical stability

- *Helmholtz equation* (page 16) increases numerical stability; optional, but highly recommended.
- *'Variant A'* (page 17) increases numerical stability; optional.
- *Coarse/fine plasma approach* (page 22) increases numerical stability; optional.
- *Offset-coordinate separation* (page 39) (probably) helps with float precision.

Simplifications

- *Reflection boundary* (page 6) is closer than the field calculation boundary to simplify boundary handling.

1.3 Simulation window and grids

Fields and densities grid

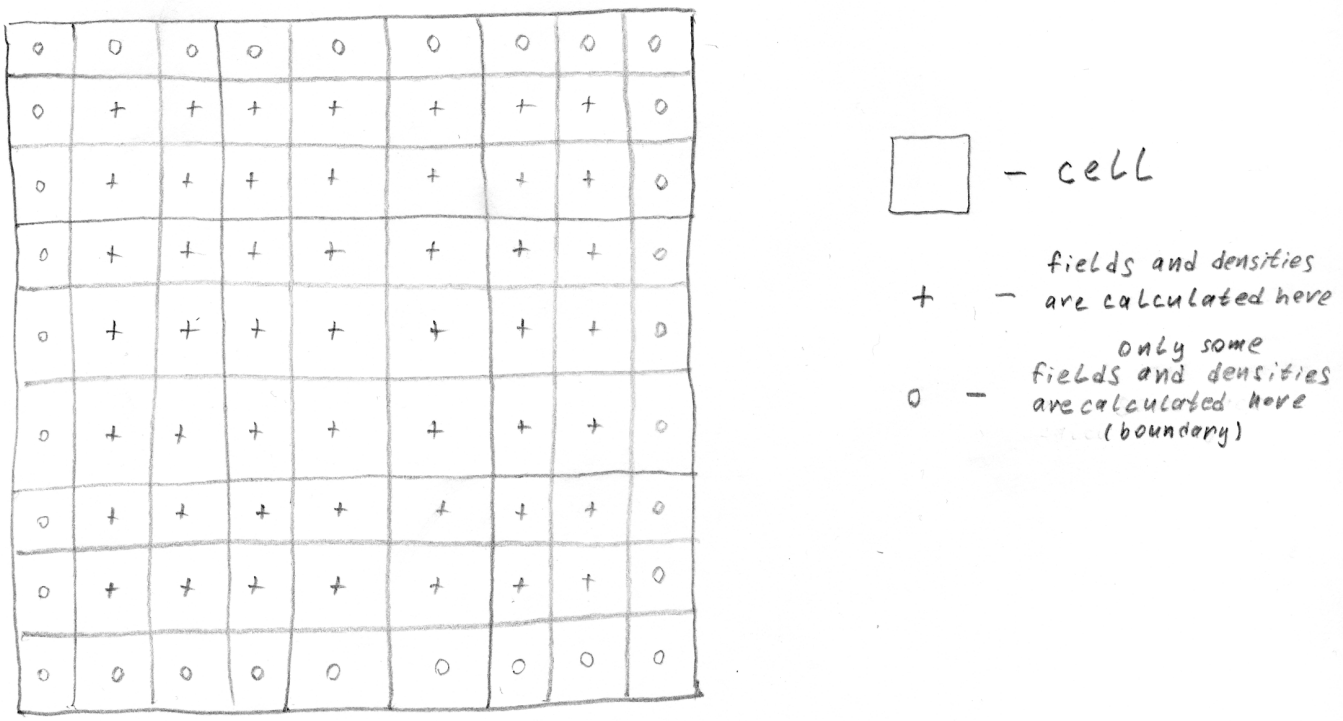


Fig. 1: The grid on which the fields and densities are calculated.

Fields and densities (ro, j) are calculated on a `config.grid_steps` x `config.grid_steps`-sized grid. This number must be odd in order to have an on-axis cell for on-axis diagnostics.

`config_example.`**`grid_steps = 641`**
: Transverse grid size in cells

`config_example.`**`grid_step_size = 0.025`**
: Transverse grid step size in plasma units

The fields are calculated at the centers of the grid cells.

Note: The muddy concept of 'window width' is no longer referenced in LCODE 3D to ease up the inevitable confusion about what it actually means and how it relates to `config.grid_step_size`. Please refrain from thinking in these terms and head over to the following subsection for more useful ones.

Reflect and 'plasma' boundaries

`config_example.`**`reflect_padding_steps = 5`**
: Plasma reflection <-> field calculation boundaries

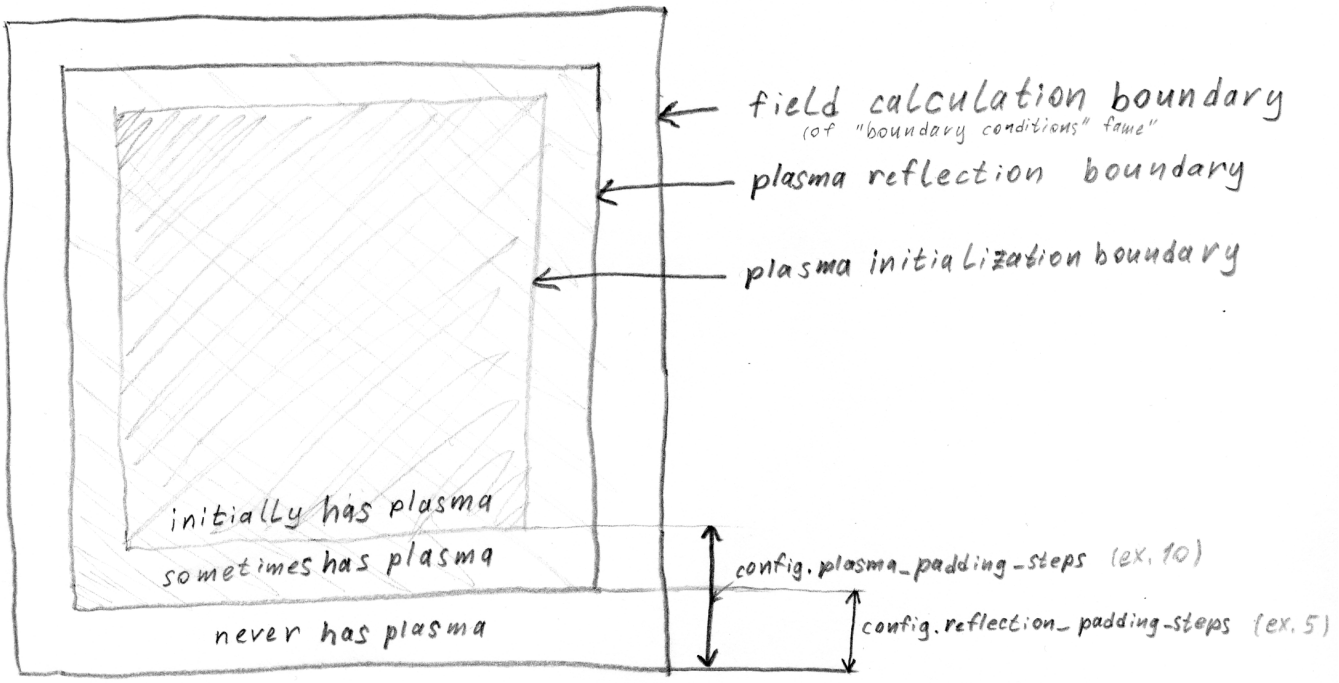


Fig. 2: The reflect and plasma boundaries illustrated.

`config_example.`**`plasma_padding_steps = 10`**
Plasma placement <-> field calculation boundaries

The plasma partlcles are not allowed to enter the outermost cells in order to simplify the treatment of boundary regions during interpolation, deposition and field calculation [*Zero special boundary treatment* (page 40)]. In order to achieve that, the reflection boundary is placed `config.reflection_padding_steps` steps deeper into the simulation area.

Note: While choosing the value for this parameter, one should take into account the particle size. Even a single fine [*Coarse/fine plasma* (page 22)] particle is three cells wide in deposition, [*Plasma* (page 20)], so the gap width should be wide enough to cover the entire coarse particle cloud size. Failure to meet this condition may result in a memory violation error. This could be solved by introducing fine particle reflection, but that'd be more resource-intensive.

Note that while it defines the area where plasma is allowed to be present, it must be larger than the area where the plasma is initially positioned. The size of the second area is controlled by the `config.plasma_padding_steps`, which puts a cut-off limit on the placement of both coarse and fine plasma particles.

Coarse and fine plasma grids

Finally, the plasma boundary hosts two grids of particles, the coarse grid and the fine grid [more info in *Coarse/fine plasma approach* (page 22)], which are coarser and finer than the field grid respectively.

1.4 Units

Todo: DOCS (Lotov)

CHAPTER 2

Usage

2.1 Installation

Common

LCODE 3D requires an NVIDIA GPU with CUDA support. CUDA Compute Capability 6+ is strongly recommended for accelerated atomic operations support.

On the Python front, it needs Python 3.6+ and the packages listed in `requirements.txt`:

```
cupy>=5.1
matplotlib>=1.4
numba>=0.41
numpy>=1.8
scipy>=0.14
```

Most of them are extremely popular and the only one that may be slightly problematic to obtain due to its 'teen age' is `cupy`.

Linux, distribution Python

All the dependencies, except for, probably, `cupy`, should be easily installable with your package manager.

Install NVIDIA drivers and CUDA packages according to your distribution documentation.

Install `cupy` according to the official installation guide[1], unless `5.1` or newer is already packaged by your distribution.

Linux, Anaconda

All dependencies, including `cupy`, are available from the official conda channels.

```
conda install cupy
```

or, if you are a miniconda user or a thorough kind of person,

```
while read req; do conda install --yes $req; done < requirements.txt
```

You probably still need to install NVIDIA drivers and CUDA packages, follow your distribution documentation.

Linux, NixOS

```
nix-shell
```

In case it's not enough, consider either switching to NVIDIA driver, or simply adding

[1] https://docs-cupy.chainer.org/en/stable/install.html

```
boot.kernelPackages = pkgs.linuxPackages;  # explicitly require stable kernel
boot.kernelModules = [ "nvidia-uvm" ];  # should bring just enough kernel␣
↪support for CUDA userspace
```

to `/etc/nixos/configuration.nix` and rebuilding the system.

Linux, locked-down environment

If want to, e.g., run LCODE 3D on a cluster without permissions to install software the proper way, please contact the administrator first and refer them to this page.

If you are sure about CUDA support and you absolutely want to install the dependencies yourself, then make sure you have Python 3.6+ and try to install `cupy` using the official installation guide. If you succeed, install all the other missing requirements with `pip`'s 'User Installs' feature. You mileage may vary. You're responsible for the cleanup.

Windows, Anaconda

If `cupy 5.1` or newer has already hit the channels, you're in luck. Just `conda install cupy`, and you should be good to go.

If https://anaconda.org/anaconda/cupy still shows 'win-64' at `v4.1.0`, please accept our condolences and proceed to the next subsection.

Windows, the hard way

- Ensure that you have Python 3.6+.
- Free up some 10 GiB of disk space or more.
- Verify that you're on good terms with the deity of your choice.
- Install Visual Studio (Community Edition is fine) with C++ support.
- Install NVIDIA CUDA Toolkit.
- Follow the `cupy` installation guide.
- Prefer installing precompiled packages, but you might also try installing from source.
- Verify that it works by executing `import cupy; (cupy.asarray([2])**2).get()` in Python shell.
- Install the other dependencies.

Known to work

As of early 2019, LCODE 3D is developed and known to work under:

- NixOS 19.03 "Koi"
- Debian 10 "Buster" + Anaconda 2019.03

- Windows 10 1903 + Anaconda 2019.03

2.2 Running and embedding

Only two files are required

LCODE 3D is a single-file module and you only need two files to execute it: `lcode.py` and `config.py`.

Installing LCODE into `PYTHONPATH` with the likes of `pip install .` is possible, but is not officially supported.

Configuration

LCODE 3D is configured by placing a file `config.py` into the current working directory. An example is provided as `config_example.py`.

The file gets imported by the standard Python importing mechanism, the resulting module is passed around internally as `config`.

One can use all the features of Python inside the configuration file, from arithmetic expressions and functions to other modules and metaprogramming.

Execution

`python3 lcode.py`, `python lcode.py` or `./lcode.py`

Todo: CODE: embedding

CHAPTER 3

Tour of the simulations

3.1 Ez

Equations

We want to solve

$$\Delta_\perp E_z = \frac{\partial j_x}{\partial x} - \frac{\partial j_y}{\partial y}$$

with Dirichlet (zero) boundary conditions.

Method

The algorithm can be succinctly written as `iDST2D(dirichlet_matrix * DST2D(RHS))`, where `DST2D` and `iDST2D` are Type-1 Forward and Inverse Discrete Sine 2D Trasforms respectively, `RHS` is the right-hand side of the equiation above, and `dirichlet_matrix` is a 'magical' matrix that does all the work.

`lcode.`**`dirichlet_matrix`**(*grid_steps*, *grid_step_size*)

Calculate a magical matrix that solves the Laplace equation if you elementwise-multiply the RHS by it "in DST-space". See Samarskiy-Nikolaev, p. 187.

In addition to the magic values, it also hosts the DST normalization multiplier.

Todo: DOCS: expand with method description (Kargapolov, Shalimova)

`lcode.`**`calculate_Ez`**(*config*, *jx*, *jy*)

Calculate Ez as iDST2D(dirichlet_matrix * DST2D(djx/dx + djy/dy)).

Note that the outer cells do not participate in the calculations, and the result is simply padded with zeroes in the end.

DST2D

`lcode.`**`dst2d`**(*a*)

Calculate DST-Type1-2D, jury-rigged from anti-symmetrically-padded rFFT.

As `cupy` currently ships no readily available function for calculating the DST2D on the GPU, we roll out our own FFT-based implementation.

We don't need to make a separate iDST2D function as (for Type-1) it matches DST2D up to the normalization multiplier, which is taken into account in *`dirichlet_matrix()`* (page 15).

3.2 Ex, Ey, Bx, By

Theory

We want to solve

$$\Delta_\perp E_x = \frac{\partial \rho}{\partial x} - \frac{\partial j_x}{\partial \xi}$$
$$\Delta_\perp E_y = \frac{\partial \rho}{\partial y} - \frac{\partial j_y}{\partial \xi}$$
$$\Delta_\perp B_x = \frac{\partial j_y}{\partial \xi} - \frac{\partial j_z}{\partial y}$$
$$\Delta_\perp B_y = \frac{\partial j_z}{\partial x} - \frac{\partial j_x}{\partial \xi}$$

with mixed boundary conditions.

Todo: DOCS: specify the boundary conditions.

Unfortunately, what we actually solve is no less than three steps away from these.

Helmholtz equations

The harsh reality of numerical stability forces us to solve these Helmholtz equations instead:

$$\Delta_\perp E_x - E_x = \frac{\partial \rho}{\partial x} - \frac{\partial j_x}{\partial \xi} - E_x$$
$$\Delta_\perp E_y - E_y = \frac{\partial \rho}{\partial y} - \frac{\partial j_y}{\partial \xi} - E_y$$
$$\Delta_\perp B_x - B_x = \frac{\partial j_y}{\partial \xi} - \frac{\partial j_z}{\partial y} - B_x$$
$$\Delta_\perp B_y - B_y = \frac{\partial j_z}{\partial x} - \frac{\partial j_x}{\partial \xi} - B_y$$

Note: The behaviour is actually configurable with `config.field_solver_subtraction_trick` (what a mouthful). `0` or `False` corresponds to Laplace equation, and while any floating-point values or whole matrices of them should be accepted, it's recommended to simply use `1` or `True` instead.

`config_example.`**`field_solver_subtraction_trick = 1`**
: 0 for Laplace eqn., Helmholtz otherwise

Method

The algorithm can be succinctly written as `iMIX2D(mixed_matrix * MIX2D(RHS))`, where `MIX2D` and `iMIX2D` are Type-1 Forward and Inverse Discrete Trasforms, Sine in one direction and Cosine in the other. `RHS` is the right-hand side of the equiation above, and `dirichlet_matrix` is a 'magical' matrix that does all the work.

lcode.**mixed_matrix**(*grid_steps*, *grid_step_size*, *subtraction_trick*)

Calculate a magical matrix that solves the Helmholtz or Laplace equation (subtraction_trick=True and subtraction_trick=False correspondingly) if you elementwise-multiply the RHS by it "in DST-DCT-transformed-space". See Samarskiy-Nikolaev, p. 189 and around.

Todo: DOCS: expand with method description (Kargapolov, Shalimova)

---

lcode.**calculate_Ex_Ey_Bx_By**(*config*, *Ex_avg*, *Ey_avg*, *Bx_avg*, *By_avg*, *beam_ro*, *ro*, *jx*, *jy*, *jz*, *jx_prev*, *jy_prev*)

Calculate transverse fields as iDST-DCT(mixed_matrix * DST-DCT(RHS.T)).T, with and without transposition depending on the field component.

Note that some outer cells do not participate in the calculations, and the result is simply padded with zeroes in the end. We don't define separate functions for separate boundary condition types and simply transpose the input and output data.

DST-DCT Hybrid

lcode.**mix2d**(*a*)

Calculate a DST-DCT-hybrid transform (DST in first direction, DCT in second one), jury-rigged from padded rFFT (anti-symmetrically in first direction, symmetrically in second direction).

As `cupy` currently ships no readily available function for calculating even 1D DST/DCT on the GPU, we, once again, roll out our own FFT-based implementation.

We don't need a separate function for the inverse transform, as it matches the forward one up to the normalization multiplier, which is taken into account in *mixed_matrix()* (page 16).

Variant B

But wait, the complications don't stop here.

While we do have a succesfully implemented $(\Delta_\perp - 1)$ inverse operator, there's still an open question of supplying an unknown value to the RHS.

The naive version (internally known as 'Variant B') is to pass the best known substitute to date, i.e. previous layer fields at the predictor phase and the averaged fields at the corrector phase. ξ-derivatives are taken at half-steps, transverse derivatives are averaged at half-steps or taken from the previous layer if not available.

$$
\begin{aligned}
(\Delta_\perp - 1)E_x^{next} &= \frac{\partial \rho^{prev}}{\partial x} - (\frac{\partial j_x}{\partial \xi})^{\text{halfstep}} - E_x^{avg} \\
(\Delta_\perp - 1)E_y^{next} &= \frac{\partial \rho^{prev}}{\partial y} - (\frac{\partial j_y}{\partial \xi})^{\text{halfstep}} - E_y^{avg} \\
(\Delta_\perp - 1)B_x^{next} &= (\frac{\partial j_y}{\partial \xi})^{\text{halfstep}} - \frac{\partial j_z^{prev}}{\partial y} - B_x^{avg} \\
(\Delta_\perp - 1)B_y^{next} &= \frac{\partial j_z^{prev}}{\partial x} - (\frac{\partial j_x}{\partial \xi})^{\text{halfstep}} - B_y^{avg}
\end{aligned}
$$

Variant A

The more correct version (known as 'Variant A') mutates the equations once again to take everything at half-steps:

$$
\begin{aligned}
(\Delta_\perp - 1) E_x^{\text{halfstep}} &= \frac{\partial \rho^{\text{avg}}}{\partial x} - (\frac{\partial j_x}{\partial \xi})^{\text{halfstep}} - E_x^{\text{avg}} \\
(\Delta_\perp - 1) E_y^{\text{halfstep}} &= \frac{\partial \rho^{\text{avg}}}{\partial y} - (\frac{\partial j_y}{\partial \xi})^{\text{halfstep}} - E_y^{\text{avg}} \\
(\Delta_\perp - 1) B_x^{\text{halfstep}} &= (\frac{\partial j_y}{\partial \xi})^{\text{halfstep}} - \frac{\partial j_z^{\text{avg}}}{\partial y} - B_x^{\text{avg}} \\
(\Delta_\perp - 1) B_y^{\text{halfstep}} &= \frac{\partial j_z^{\text{avg}}}{\partial x} - (\frac{\partial j_x}{\partial \xi})^{\text{halfstep}} - B_y^{\text{avg}}
\end{aligned}
$$

and calculates the fields at next step in the following fashion: $E_x^{next} = 2E_x^{avg} - E_x^{prev}$, e.t.c.

Solving these is equivalent to solving Variant B equations with averaged fields, ρ and j_z and applying the above transformation to the result. See `lcode.step()` (page 31) for the wrapping code that does that.

`config_example.`**`field_solver_variant_A = True`**

Use Variant A or Variant B for Ex, Ey, Bx, By

3.3 Bz

Equations

We want to solve

$$\Delta_\perp B_z = \frac{\partial j_x}{\partial y} - \frac{\partial j_y}{\partial x}$$

with Neumann boundary conditions (derivative = 0).

Method

The algorithm can be succinctly written as `iDCT2D(neumann_matrix * DCT2D(RHS))`, where `DCT2D` and `iDCT2D` are Type-1 Forward and Inverse Discrete Sine 2D Trasforms respectively, `RHS` is the right-hand side of the equiation above, and `neumann_matrix` is a 'magical' matrix that does all the work.

`lcode.`**`neumann_matrix`**(*grid_steps*, *grid_step_size*)

Calculate a magical matrix that solves the Laplace equation if you elementwise-multiply the RHS by it "in DST-space". See Samarskiy-Nikolaev, p. 187.

In addition to the magic values, it also hosts the DCT normalization multiplier.

Todo: DOCS: expand with method description (Kargapolov, Shalimova)

`lcode.`**`calculate_Bz`**(*config*, *jx*, *jy*)

Calculate Bz as iDCT2D(dirichlet_matrix * DCT2D(djx/dy - djy/dx)).

Note that this time the outer cells do not participate in the calculations, so the RHS derivatives are padded with zeroes in the beginning.

DCT2D

`lcode.`**`dct2d`**(*a*)

Calculate DCT-Type1-2D, jury-rigged from symmetrically-padded rFFT.

As `cupy` currently ships no readily available function for calculating the DCT2D on the GPU, we roll out our own FFT-based implementation.

We don't need to make a separate iDCT2D function as (for Type-1) it matches DCT2D up to the normalization multiplier, which is taken into account in *`neumann_matrix()`* (page 19).

3.4 Plasma

Characteristics

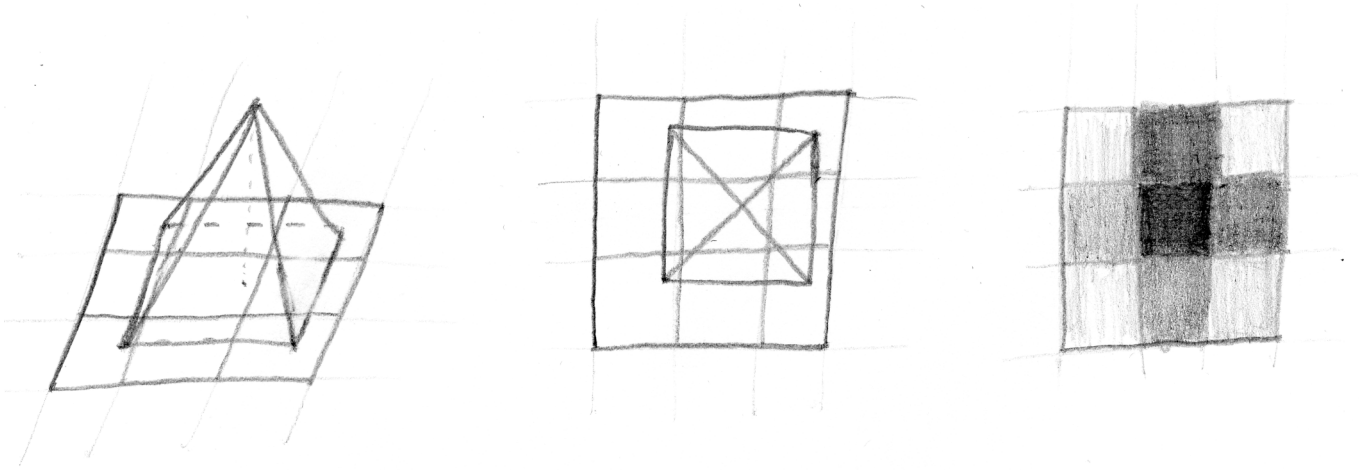
A plasma particle has these characteristics according to our model:

- Coordinates x and y, stored as `x_init + x_offt` and `y_init + y_offt` [*Offset-coordinate separation* (page 39)].
- Momenta p_x, p_y and p_z, stored as `px`, `py` and `pz`.
- Charge q, stored as `q`.
- Mass m, stored as `m`.

Shape

From the interpolation/deposition perspective, a plasma particle represents not a point in space, but a 2D Triangular-Shaped Cloud (TSC2D).

These clouds always (partially) cover an area the size of $3x3$ cells: the one where their center lies and eight neighouring ones.



Todo: DOCS: WRITE: write a nicer formula for the weights of each cell.

`lcode.`**`weights`**(*x*, *y*, *grid_steps*, *grid_step_size*)
Calculate the indices of a cell corresponding to the coordinates, and the coefficients of interpolation and deposition for this cell and 8 surrounding cells. The weights correspond to 2D triangluar shaped cloud (TSC2D).

The same coefficients are used for both deposition of the particle characterictics onto the grid [*Deposition* (page 26)] and interpolation of the fields in the particle center positions [*Plasma pusher* (page 28)].

`lcode.`**`deposit9`**(*a*, *i*, *j*, *val*, *wMP*, *w0P*, *wPP*, *wM0*, *w00*, *wP0*, *wMM*, *w0M*, *wPM*)
Deposit value into a cell and 8 surrounding cells (using *weights* output).

`lcode.`**`interp9`**(*a*, *i*, *j*, *wMP*, *w0P*, *wPP*, *wM0*, *w00*, *wP0*, *wMM*, *w0M*, *wPM*)
Collect value from a cell and 8 surrounding cells (using *weights* output).

The concept is orthogonal to the coarse plasma particle shape [*Coarse and fine plasma* (page 22)]. While a coarse particle may be considered to be a component of an elastic cloud of fine particles, each individial fine particle sports the same TSC2D shape.

3.5 Coarse and fine plasma

Concept

In order to increase stability and combat transverse grid noise, LCODE 3D utilises a dual plasma appoach.

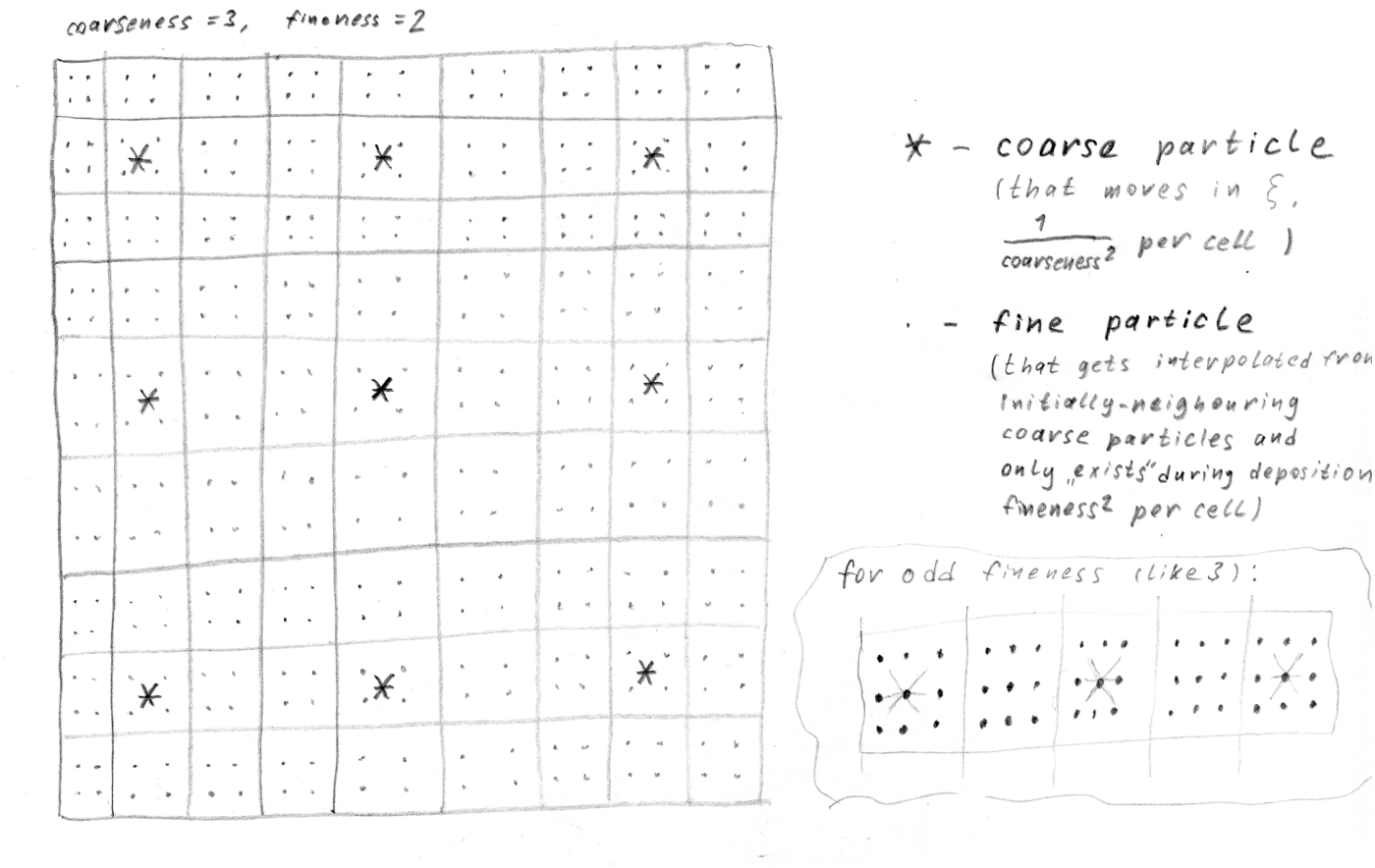


Fig. 1: Positioning of the coarse and fine particles in dual plasma approach.

Coarse particles are the ones that get tracked throughout the program, and pushed by the pusher. There coarse plasma grid is many times more sparse than the fields grid, think $\frac{1}{9}$ particles per cell.

`config_example.`**`plasma_coarseness = 3`**
: Square root of the amount of cells per coarse particle

Fine particles only exist inside the deposition phase. There are several fine particles per cell, think 4 or more. Their characteristic values are neither stored or evolved; instead they are intepolated from the coarse particle grid as a part of the the deposition process (and they don't 'exist' in any form outside of it).

`config_example.`**`plasma_fineness = 2`**
: Square root of the amount of fine particles per cell

Initialization

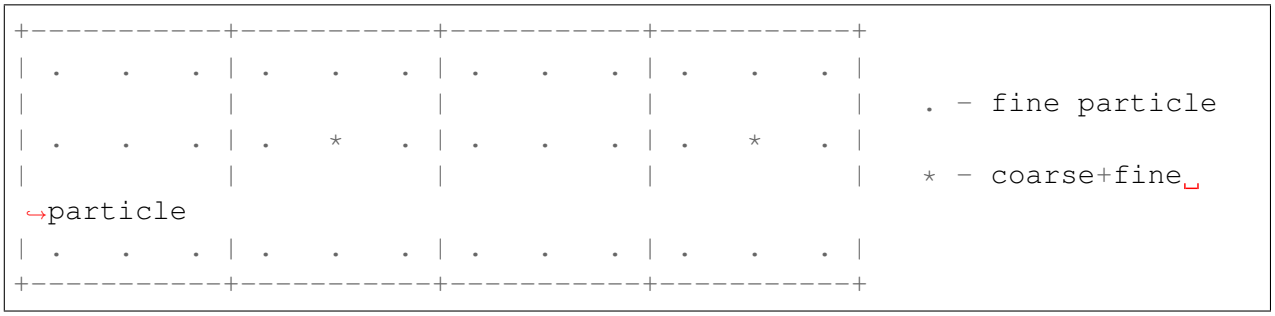
`lcode.`**`make_coarse_plasma_grid`**(*steps*, *step_size*, *coarseness=3*)
: Create initial coarse plasma particles coordinates (a single 1D grid for both x and y).

lcode.**make_fine_plasma_grid**(*steps*, *step_size*, *fineness=2*)

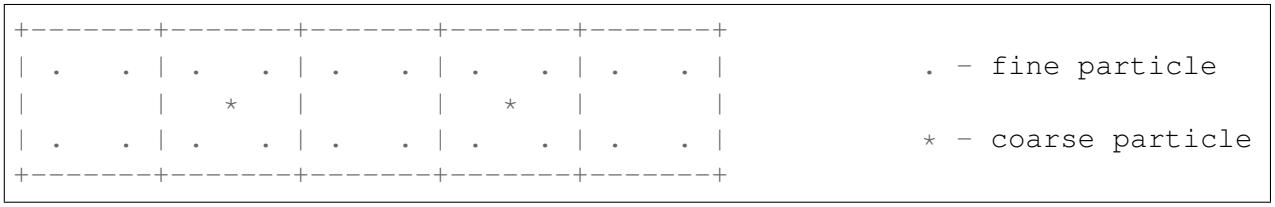
Create initial fine plasma particles coordinates (a single 1D grid for both x and y).

Avoids positioning particles at the cell edges and boundaries.

- `fineness=3` (and `coarseness=2`):



- `fineness=2` (and `coarseness=2`):



lcode.**make_plasma**(*steps*, *cell_size*, *coarseness=3*, *fineness=2*)

Make coarse plasma initial state arrays and the arrays needed to intepolate coarse plasma into fine plasma (`virt_params`).

Coarse is the one that will evolve and fine is the one to be bilinearly interpolated from the coarse one based on the initial positions (using 1 to 4 coarse plasma particles that initially were the closest).

Initializing coarse particles is pretty simple: `coarse_x_init` and `coarse_y_init` are broadcasted output of *`make_coarse_plasma_grid()`* (page 22). `coarse_x_offt` and `coarse_y_offt` are zeros and so are `coarse_px`, `coarse_py` and `coarse_pz`. `coarse_m` and `coarse_q` are constants divided by the factor of coarseness by fineness squared because fine particles represent smaller macroparticles.

Initializing fine particle boils down to calculating the interpolation coefficients (`influence_prev` and `influence_next`) and the indices of the coarse particles (`indices_prev`, `indices_next`) that the characteristics will be intepolated from.

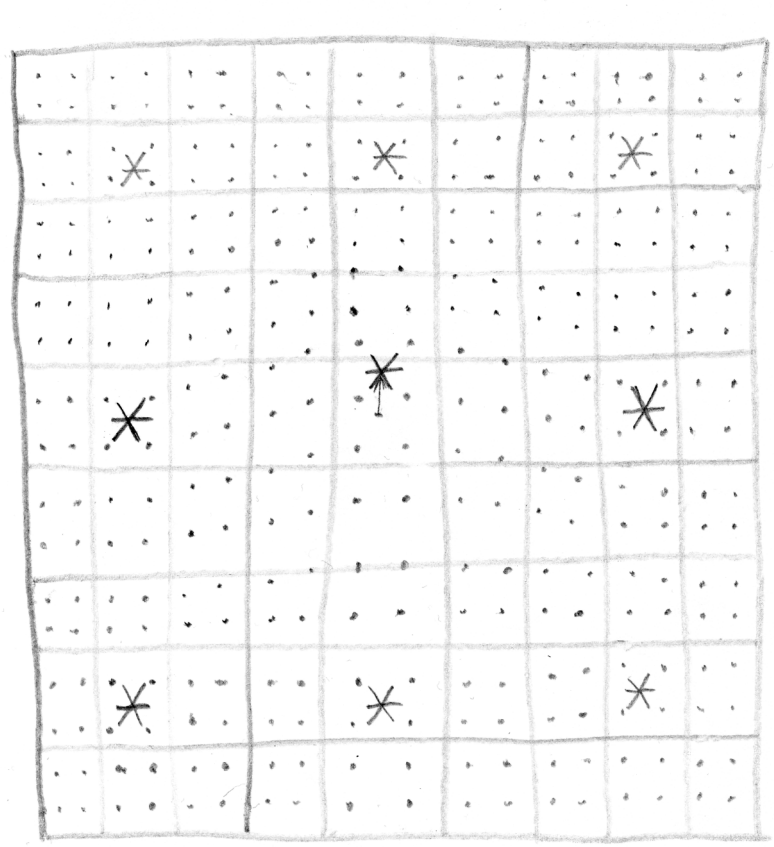
`influence_prev` and `influence_next` are linear interpolation coefficients based on the initial closest coarse particle positioning. Note that these are constant and do not change in ξ. The edges get special treatment later.

`indices_next` happens to be pretty much equal to `np.searchsorted(coarse_grid, fine_grid)` and `indices_prev` is basically `indices_next - 1`, except for the edges, where a fine particle can have less than four 'parent' coarse particles. For such 'outer' particles, existing coarse particles are used instead, so clipping the indices and fixing `influence`-arrays is carried out.

Note that these arrays are 1D for memory considerations [*Memory considerations* (page 40)].

The function returns the coarse particles and `virtparams`: a `GPUArrays` instance that conveniently groups the fine-particle related arrays, which only matter during deposition, under a single name.

Coarse-to-fine interpolation



`lcode.`**`mix`**(*coarse*, *A*, *B*, *C*, *D*, *pi*, *ni*, *pj*, *nj*)

Bilinearly interpolate fine plasma properties from four historically-neighbouring plasma particle property values:

```
B     D    #  y ^          A - bottom-left  neighbour, indices: pi, pj
   .       #    |          B - top-left     neighbour, indices: pi, nj
           #    +----->    C - bottom-right neighbour, indices: ni, pj
A     C     #         x    D - top-right    neighbour, indices: ni, nj
```

See the rest of the deposition and plasma creation for more info.

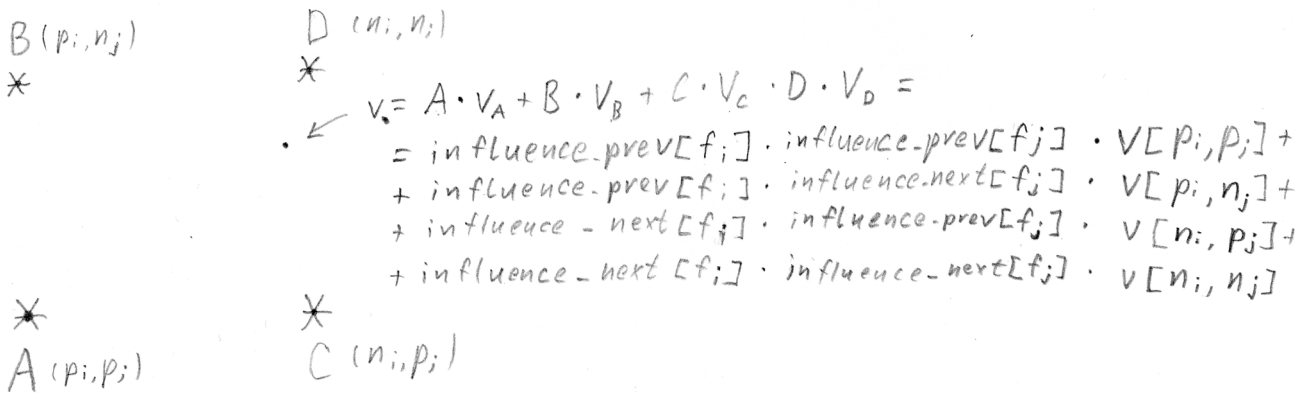
This is just a shorthand for the characteristic value mixing for internal use in `coarse_to_fine`.

`lcode.`**`coarse_to_fine`**(*fi*, *fj*, *c_x_offt*, *c_y_offt*, *c_m*, *c_q*, *c_px*, *c_py*, *c_pz*, *virt-plasma_smallness_factor*, *fine_grid*, *influence_prev*, *influence_next*, *indices_prev*, *indices_next*)

Bilinearly interpolate fine plasma properties from four historically-neighbouring plasma particle property values.

The internals are pretty straightforward once you wrap your head around the indexing.

A single fine particle with the indices `[fi, fj]` (in fine particles `virt_params` 1D arrays) is interpolated from four particles with indices `[pi, pj]`, `[pi, nj]`, `[ni, pj]`, `[ni, nj]` (in coarse particles `c_*` arrays) and four weights `A`, `B`, `C`, `D` respectively. The weights are, in turn, composed as a products of values from `influence_prev` and `influence_next` arrays, indiced, once again, with `[fi, fj]`. It would be convenient to calculate them beforehand, but they are recalculated instead as a result of time-memory tradeoff [*Memory considerations* (page 40)].



Finally, momenta, charge and mass are scaled according to the coarse-to-fine macrosity coefficient discussed above.

Alternative illustration

3.6 Deposition

Deposition operates on fine particles. Once the *coarse-to-fine interpolation* (page 22) is out of the picture, there isn't much left to discuss.

`lcode.`**`deposit_kernel`**(*grid_steps*, *grid_step_size*, *virtplasma_smallness_factor*, *c_x_offt*, *c_y_offt*, *c_m*, *c_q*, *c_px*, *c_py*, *c_pz*, *fine_grid*, *influence_prev*, *influence_next*, *indices_prev*, *indices_next*, *out_ro*, *out_jx*, *out_jy*, *out_jz*)

Interpolate coarse plasma into fine plasma and deposit it on the charge density and current grids.

First, the fine particle characteristics are interpolated from the coarse ones. Then the total contribution of the particles to the density and the currents is calculated and, finally, deposited on a grid in a 3x3 cell square with `i`, `j` as its center according to the weights calculated by `weights()` (page 20). Finally, the *ion background density* (page 27) is added to the resulting array.

The strange incantation at the top and the need to modify the output arrays instead of returning them are dictated by the fact that ihis is actually not a function, but a CUDA kernel (for more info, refer to *CUDA kernels with numba.cuda* (page 35)). It is launched in parallel for each fine particle, determines its 2D index (`fi`, `fj`), interpolates its characteristics from coarse particles and proceeds to deposit it.

`lcode.`**`deposit`**(*config*, *ro_initial*, *x_offt*, *y_offt*, *m*, *q*, *px*, *py*, *pz*, *virt_params*)

Interpolate coarse plasma into fine plasma and deposit it on the charge density and current grids. This is a convenience wrapper around the `deposit_kernel` CUDA kernel.

This function allocates the output arrays, unpacks the arguments from `config` and `virt_params`, calculates the kernel dispatch parameters (for more info, refer to *CUDA kernels with numba.cuda* (page 35)), and launches the kernel.

Todo: DOCS: explain deposition contribution formula (Lotov)

3.7 Background ions

LCODE 3D currently simulates only the movement of plasma electrons. The ions are modelled as a constant charge density distribution component that is calculated from the inital electron placement during the *initialization* (page 34).

`lcode.`**`initial_deposition`**(*config*, *x_offt*, *y_offt*, *px*, *py*, *pz*, *m*, *q*, *virt_params*)

Determine the background ion charge density by depositing the electrons with their initial parameters and negating the result.

For this initial deposition invocation, the ion density argument is specified as `0`.

The result is stored as `const.ro_initial` and passed to every consequtive *`lcode.deposit()`* (page 26) invocation.

3.8 Plasma pusher

Without fields

The coordinate-evolving equations of motion are as follows:

$$\frac{dx}{d\xi} = -\frac{v_x}{1 - v_z}$$
$$\frac{dy}{d\xi} = -\frac{v_y}{1 - v_z}$$
$$\vec{v} = \frac{\vec{p}}{\sqrt{M^2 + p^2}}$$

`lcode.`**`move_estimate_wo_fields`**(*config*, *m*, *x_init*, *y_init*, *prev_x_offt*, *prev_y_offt*, *px*, *py*, *pz*)

Move coarse plasma particles as if there were no fields. Also reflect the particles from +-*reflect_boundary*.

This is used at the beginning of the *xi step* (page 31) to roughly estimate the half-step positions of the particles.

The reflection here flips the coordinate, but not the momenta components.

With fields

The coordinate-evolving equation of motion is as follows:

$$\frac{d\vec{p}}{d\xi} = -\frac{q}{1 - v_z}\left(\vec{E} + \left[\vec{v} \times \vec{B}\right]\right)$$

As the particle momentum is present at both sides of the equation (as p and v respectively), an iterative predictor-corrector scheme is employed.

The alternative is to use a symplectic solver that solves the resulting matrix equation (not mainlined at the moment, look for an alternative branch in `t184256`'s fork).

`lcode.`**`move_smart_kernel`**(*xi_step_size*, *reflect_boundary*, *grid_step_size*, *grid_steps*, *ms*, *qs*, *x_init*, *y_init*, *prev_x_offt*, *prev_y_offt*, *estimated_x_offt*, *estimated_y_offt*, *prev_px*, *prev_py*, *prev_pz*, *Ex_avg*, *Ey_avg*, *Ez_avg*, *Bx_avg*, *By_avg*, *Bz_avg*, *new_x_offt*, *new_y_offt*, *new_px*, *new_py*, *new_pz*)

Update plasma particle coordinates and momenta according to the field values interpolated halfway between the previous plasma particle location and the the best estimation of its next location currently available to us. Also reflect the particles from `+-reflect_boundary`.

The function serves as *the* coarse particle loop, fusing together midpoint calculation, field interpolation with *`interp9()`* (page 20) and particle movement for performance reasons.

The equations for half-step momentum are solved twice, with more precise momentum for the second time.

The particles coordinates are advanced using half-step momentum, and afterwards the momentum is advanced to the next step.

The reflection is more involved this time, affecting both the coordinates and the momenta.

Note that the reflected particle offsets are mixed with the positions, resulting in a possible float precision loss [*Offset-coordinate separation* (page 39)]. This effect probably negligible at this point, as the particle had to travel at least several cell sizes at this point. The only place where the separation really matters is the (final) coordinate addition (`x_offt += ...` and `y_offt += ...`).

The strange incantation at the top and the need to modify the output arrays instead of returning them is dictated by the fact that ihis is actually not a function, but a CUDA kernel (for more info, refer to *CUDA kernels with numba.cuda* (page 35)), It is launched in parallel for each coarse particle, determines its 1D index `k`, interpolates the fields at its position and proceeds to move and reflect it.

`lcode.`**`move_smart`**(*config*, *m*, *q*, *x_init*, *y_init*, *x_prev_offt*, *y_prev_offt*, *estimated_x_offt*, *estimated_y_offt*, *px_prev*, *py_prev*, *pz_prev*, *Ex_avg*, *Ey_avg*, *Ez_avg*, *Bx_avg*, *By_avg*, *Bz_avg*)

Update plasma particle coordinates and momenta according to the field values interpolated halfway between the previous plasma particle location and the the best estimation of its next location currently available to us. This is a convenience wrapper around the `move_smart_kernel` CUDA kernel.

This function allocates the output arrays, unpacks the arguments from `config` calculates the kernel dispatch parameters (for more info, refer to *CUDA kernels with numba.cuda* (page 35)), flattens the input and output array of particle characteristics (as the pusher does not care about the particle 2D indices) and launches the kernel.

3.9 Beam

The beam is currently simulated as a charge density function $\rho_b(\xi, x, y)$, and not with particles.

In the future, there will certainly be a way to define a beam with particles and simulate beam-plasma interaction both ways, but for now only simulating a plasma response to a rigid beam is possible.

`config_example.`**`beam`**(*xi_i*, *x*, *y*)

The user should specify the beam charge density as a function in the configuration file.

`xi_i` is not the value of the ξ coordinate, but the step index. Please use something in the lines of `xi = -xi_i * xi_step_size + some_offset`, according to where exactly in ξ do you define the beam density slices [*Integer xi steps* (page 41)].

`x` and `y` are `numpy` arrays, so one should use vectorized numpy operations to calculate the desired beam charge density, like `numpy.exp(-numpy.sqrt(x**2 + y**2))`.

The function should ultimately return an array with the same shape as `x` and `y`.

Todo: CODE: Simulate the beam with particles and evolve it according to the plasma response.

3.10 Looping in xi

Finally, here's the function that binds it all together, and currently makes up half of LCODE 3D API.

In short it: moves, deposits, estimates fields, moves, deposits, recalculates fields, moves and deposits.

`config_example.`**`xi_step_size = 0.005`**
Step size in time-space coordinate xi

`config_example.`**`xi_steps = 599999`**
Amount of xi steps

`lcode.`**`step`**(*config*, *const*, *virt_params*, *prev*, *beam_ro*)
Calculate the next iteration of plasma evolution and response. Returns the new state with the following attributes: `x_offt`, `y_offt`, `px`, `py`, `pz`, `Ex`, `Ey`, `Ez`, `Bx`, `By`, `Bz`, `ro`, `jx`, `jy`, `jz`. Pass the returned value as `prev` for the next iteration. Wrap it in `GPUArraysView` if you want transparent conversion to `numpy` arrays.

Input parameters

Beam density array *rho_b* (`beam_ro`) is copied to the GPU with `cupy.asarray`, as it is calculated with `numpy` in config-residing `beam()`.

All the other arrays come packed in `GPUArrays` objects [*GPU array conversion* (page 36)], which ensures that they reside in the GPU memory. These objects are:

- `const` and `virt_params`, which are constant at least for the ξ-step duration and defined during the *initialization* (page 34), and
- `prev`, which is usually obtained as the return value of the previous `step()` invocation, except for the very first step.

Initial half-step estimation

1. The particles are advanced according to their current momenta only (*`lcode.move_estimate_wo_fields()`* (page 28)).

Field prediction

While we don't know the fields on the next step:

2. The particles are advanced with the fields from **the previous step** using the coordinates **estimated at 1.** to calculate the half-step positions where the **previous step** fields should be interpolated at (*`lcode.move_smart()`* (page 29)).
3. The particles from **2.** are deposited onto the charge/current density grids (*`lcode.deposit()`* (page 26)).
4. The fields at the next step are calculated using densities from **3.** (*`lcode.calculate_Ez()`* (page 15), *`lcode.calculate_Ex_Ey_Bx_By()`* (page 17), *`lcode.calculate_Bz()`* (page 19)) and averaged with the previous fields.

This phase gives us an estimation of the fields at half-step, and the coordinate estimation at next step, while all other intermediate results are ultimately ignored.

Field correction

5. The particles are advanced with the **averaged** fields from **4.**, using the coordinates **from 2.** to calculate the half-step positions where the **averaged** fields from *4.* should be interpolated at (*lcode.move_smart()* (page 29)).
6. The particles from **5.** are deposited onto the charge/current density grids (*lcode.deposit()* (page 26)).
7. The fields at the next step are calculated using densities from **6.** (*lcode.calculate_Ez()* (page 15), *lcode.calculate_Ex_Ey_Bx_By()* (page 17), *lcode.calculate_Bz()* (page 19)) and averaged with the previous fields.

The resulting fields are far more precise than the ones from the prediction phase, but the coordinates and momenta are still pretty low-quality until we recalculate them using the new fields. Iterating the algorithm more times improves the stability, but it currently doesn't bring much to the table as the transverse noise dominates.

Final plasma evolution and deposition

8. The particles are advanced with the **averaged** fields from **7.**, using the coordinates **from 5.** to calculate the half-step positions where the **averaged** fields from **7.** should be interpolated at (*lcode.move_smart()* (page 29)).
9. The particles from **8.** are deposited onto the charge/current density grids (*lcode.deposit()* (page 26)).

The result, or the 'new prev'

The fields from 7., coordinates and momenta from 8., and densities from 9. make up the new `GPUArrays` collection that would be passed as `prev` to the next iteration of `step()`.

CHAPTER 4

Technicalities

4.1 Initialization

`lcode.`**`init`**(*config*)

Initialize all the arrays needed for `step` and `config.beam`.

This function performs quite a boring sequence of actions, outlined here for interlinking purposes:

- validates the oddity of `config.grid_steps` [*Fields and densities grid* (page 6)],
- validates that `config.reflect_padding_steps` is large enough [*Reflect and 'plasma' boundaries* (page 6)],
- calculates the `reflect_boundary` and monkey-patches it back into `config`,
- initializes the `x` and `y` arrays for use in *`config_example.beam()`* (page 30),
- calculates the plasma placement boundary,
- immediately passes it to *`make_plasma()`* (page 23), leaving it oblivious to the padding concerns,
- performs the initial electron deposition to obtain the background ions charge density [*Background ions* (page 27)],
- groups the constant arrays into a `GPUArray` instance `const`, and
- groups the evolving arrays into a `GPUArray` instance `state`.

4.2 GPU calculations pecularities

LCODE 3D performs most of the calculations on GPU using a mix of two approaches.

CUDA kernels with numba.cuda

One can use CUDA from Python more or less directly by writing and launching CUDA kernels with `numba.cuda`.

An example would be:

```
@numba.cuda.jit
def add_two_arrays_kernel(arr1, arr2, result):
    i = numba.cuda.grid(1)
    if i >= arr1.shape[0]:
        return
    result[i] = arr1[i] + arr2[i]
```

This function represents a loop body, launched in parallel with many threads at once. Each of them starts with obtaining the array index it is 'responsible' for with `cuda.grid(1)` and then proceeds to perform the required calculation. As it is optimal to launch them in 32-threaded 'warps', one also has to handle the case of having more threads than needed by making them skip the calculation.

No fancy Python operations are supported inside CUDA kernels, it is basically a way to write C-like bodies for hot loops without having to write actual C/CUDA code. You can only use simple types for kernel arguments and you cannot return anything from them.

To rub it in, this isn't even a directly callable function yet. To conceal the limitations and the calling complexity, it is convenient to write a wrapper for it.

```
def add_two_arrays(arr1, arr2):
    result = cp.zeros_like(arr1)  # uses cupy, see below
    warp_count = int(ceil(arr1.size / WARP_SIZE))
    add_two_arrays_kernel[warp_count, WARP_SIZE](arr1, arr2, result)
    return result
```

A pair of numbers (`warp_count`, `WARP_SIZE`) is required to launch the kernel. `warp_count` is chosen this way so that `warp_count * WARP_SIZE` would be larger than the problem size.

`lcode.`**`WARP_SIZE = 32`**

Should be detectable with newer `cupy` (>6.0.0b2) as `WARP_SIZE = cp.cuda.Device(config.gpu_index).attributes['WarpSize']`. As of 2019 it's equal to 32 for all CUDA-capable GPUs. It's even a hardcoded value in `cupy`.

Array-wise operations with cupy

`cupy` is a GPU array library that aims to implement a `numpy-like` interface to GPU arrays. It allows one to, e.g., add up two GPU arrays with a simple and terse `a + b`. Most of the functions in LCODE use vectorized operations and `cupy`. All memory management is done with `cupy` for consistency.

It's hard to underestimate the convenience of this approach, but sometimes expressing algorithms in vectorized notation is too hard or suboptimal. The only two times we're actually going for writing CUDA kernels are `deposit()` (our fine particle loop) and `move_smart()` (our coarse particle loop).

Copying is expensive

If the arrays were copied between GPU RAM and host RAM, the PCI-E bandwidth would become a bottleneck. The two most useful strategies to minimize excessive copying are

1. churning for several consecutive ξ-steps with no copying and no CPU-side data processing (with a notable exception of `beam()` and the resulting *beam_ro*); and
2. copying only the subset of the arrays that the outer diagnostics code needs.

GPU array conversion

In order for `a + b` to work in `cupy`, both arrays have to be copied to GPU (`cupy.asarray(a)`) and, in case you want the results back as `numpy` arrays, you have to explicitly copy them back (`gpu_array.get()`).

While for the LCODE 3D itself it's easier and quicker to stick to using GPU arrays exclusively, this means the only time when we want to do the conversion to `numpy` is when we are returning the results back to the external code.

There are two classes that assist in copying the arrays back and forth and conveniently as possible. The implementation looks a bit nightmarish, but using them is simple.

class lcode.**GPUArrays**(***kwargs*)

A convenient way to group several GPU arrays and access them with a dot. `x = GPUArrays(something=numpy_array, something_else=another_array)` will create `x` with `x.something` and `x.something_else` stored on GPU.

Do not add more attributes later, specify them all at construction time.

class lcode.**GPUArraysView**(*gpu_arrays*)

This is a magical wrapper around GPUArrays that handles GPU-RAM data transfer transparently. Accessing `view.something` will automatically copy array to host RAM, setting `view.something = ...` will copy the changes back to GPU RAM.

Usage: `view = GPUArraysView(gpu_arrays); view.something`

Do not add more attributes later, specify them all at construction time.

NOTE: repeatedly accessing an attribute will result in repeated copying!

This way we can wrap everything we need in GPUArrays with, e.g., `const = GPUArrays(x_init=x_init, y_init=y_init, ...)` and then access them as `const.x_init` from GPU-heavy code. For the outer code that does not care about GPU arrays at all, we can return a wrapped `const_view = GPUArraysView(const)` and access the arrays as `const_view.x_init`.

Copying will happen on-demand during the attribute access, intercepted by our `__getattr__` implementation, but beware!

Note: Repeatedly accessing `const_view.x_init` will needlessly perform the copying again, so one should bind it to the variable name (`x_init = const_view.x_init`) once and reuse the resulting `numpy` array.

Todo: CODE: wrap the returned array with GPUArraysView by default

Selecting GPU

`config_example.`**`gpu_index = 0`**
: Index of the GPU that should perform the calculations

LCODE 3D currently does not support utilizing several GPUs for one simulation, but once we switch to beam evolution calculation, processing several consecutive t-steps in a pipeline of several GPUs should be a low hanging fruit.

4.3 Optimal transverse grid sizes

FFT works best for grid sizes that are factorizable into small numbers. Any size will work, but the performance may vary dramatically.

FFTW documentation quotes the optimal size for their algorithm as $2^a 3^b 5^c 7^d 11^e 13^f$, where $e + f$ is either 0 or 1, and the other exponents are arbitrary.

While LCODE 3D does not use FFTW (it uses `cufft` instead, wrapped by `cupy`), the formula is still quite a good rule of thumb for calculating performance-friendly `config_example.grid_steps` values.

The employed FFT sizes for a grid sized N are $2N - 2$ for both DST (`dst2d()`, `mix2d()`) and DCT transforms (`dct2d()`, `mix2d()`) when we take padding and perimeter cell stripping into account.

This leaves us to find such N that $N - 1$ satisfies the small-factor conditions.

If you don't mind arbitrary grid sizes, we suggest using

1. $N = 2^K + 1$, they always perform the best, or
2. one of the roundish 201, 301, 401, 501, 601, 701, 801, 901, 1001, 1101, 1201, 1301, 1401, 1501, 1601, 1801, 2001, 2101, 2201, 2401, 2501, 2601, 2701, 2801, 3001, 3201, 3301, 3501, 3601, 3901, 4001.

The code to check for the FFTW criteria above and some of the matching numbers are listed below.

```
def factorize(n, a=[]):
    if n <= 1:
        return a
    for i in range(2, n + 1):
        if n % i == 0:
            return factorize(n // i, a + [i])

def good_size(n):
    factors = factorize(n - 1)
    return (all([f in [2, 3, 4, 5, 7, 11, 13] for f in factors])
            and actors.count(11) + factors.count(13) < 2 and
            and n % 2)

', '.join([str(a) for a in range(20, 4100) if good_size(a)])
```

21, 23, 25, 27, 29, 31, 33, 37, 41, 43, 45, 49, 51, 53, 55, 57, 61, 65, 67, 71, 73, 79, 81, 85, 89, 91, 97, 99, 101, 105, 109, 111, 113, 121, 127, 129, 131, 133, 141, 145, 151, 155, 157, 161, 163, 169, 177, 181, 183, 193, 197, 199, 201, 209, 211, 217, 221, 225, 235, 241, 251, 253, 257, 261, 265, 271, 281, 289, 295, 301, 309, 313, 321, 325, 331, 337, 351, 353, 361, 365, 379, 385, 391, 393, 397, 401, 417, 421, 433, 441, 449, 451, 463, 469, 481, 487, 491, 501, 505, 513, 521, 529, 541, 547, 551, 561, 577, 589, 595, 601, 617, 625, 631, 641, 649, 651, 661, 673, 687, 701, 703, 705, 721, 729, 751, 757, 769, 771, 781, 785, 793, 801, 811, 833, 841, 865, 881, 883, 897, 901, 911, 925, 937, 961, 973, 981, 991, 1001, 1009, 1025, 1041, 1051, 1057, 1079, 1081, 1093, 1101, 1121, 1135, 1153, 1171, 1177, 1189, 1201, 1233, 1249, 1251, 1261, 1275, 1281, 1297, 1301, 1321, 1345, 1351, 1373, 1387, 1401, 1405, 1409, 1441, 1457, 1459, 1471, 1501, 1513, 1537, 1541, 1561, 1569, 1585, 1601, 1621, 1639, 1651, 1665, 1681, 1729, 1751, 1761, 1765, 1783, 1793, 1801, 1821, 1849, 1873, 1891, 1921, 1945, 1951, 1961, 1981, 2001, 2017, 2049, 2059, 2081, 2101, 2107, 2113, 2157, 2161, 2185, 2201, 2241, 2251, 2269, 2305, 2311, 2341, 2353, 2377, 2401, 2431, 2451, 2465, 2497, 2501, 2521, 2549, 2561, 2593, 2601, 2641, 2647, 2689, 2701, 2731, 2745, 2751, 2773, 2801, 2809, 2817, 2881, 2913, 2917, 2941, 2971, 3001, 3025, 3073, 3081, 3121, 3137, 3151, 3169, 3201, 3235, 3241, 3251, 3277, 3301, 3329, 3361, 3403, 3431, 3457, 3501, 3511, 3521, 3529, 3565, 3585, 3601, 3641, 3697, 3745, 3751, 3781, 3823, 3841, 3851, 3889, 3901, 3921, 3961, 4001, 4033, 4051, 4097

4.4 Offset-coordinate separation

Float precision loss

When floating point numbers of different magnitudes get added up, there is an inherent precision loss that grows with the magnitude disparity.

If a particle has a large coordinate (think `5.223426`), but moves for a small distance (think `7.139152e-4`) due to low `xi_step_size` and small momentum projection, calculating the sum of these numbers suffers from the precision loss due to the finite significand size:

An oversimplified illustration in decimal notation:

```
 5.223426
+0.0007139152
=5.224139LOST
```

We have not conducted extensive research on how detrimental this round-off accumulation is to LCODE 3D numerical stability in ξ. Currently the transverse noise dominates, but in order to make our implementation a bit more future-proof, we store the plasma particle coordinates separated into two floats: initial position (`x_init`, `y_init`) and accumulated offset (`x_offt`, `y_offt`) and do not mix them.

Mixing them all the time...

OK, we do mix them. Each and every function involving them adds them up at some point and even has the code like this:

```
x = x_init + x_offt
...
x_offt = x - x_init
```

to reconstruct `x_offt` from the 'dirty' sum values `x`.

We do that because we're fine with singular round-off errors until they don't propagate to the next step, accumulating for millions of ξ-steps ('Test 1' simulations were conducted for up to 1.5 million steps).

... but not where it really matters

This way the only places where the separation should be preserved is the path from `prev.x_offt` to `new_state.x_offt`. Several `x_offt` additions are performed and rolled back at each ξ-step, but only two kinds of them persist, both residing in `move_smart()`:

1. `x_offt += px / (gamma_m - pz) * xi_step_size` does no mixing with the coordinate values, and
2. `x = +2 * reflect_boundary - x` and the similar one for the left boundary only happen during particle reflection, which presumably happens rarely and only affects the particles that have already deviated at least several cells away from the initial position.

This way most particles won't experience this kind of rounding issues with their coordinates. On the flip side, splitting the coordinates makes working with them quite unwieldy.

4.5 Design decisions

Codebase complexity

The code strives to be readable and vaguely understandable by a freshman student with only some basic background in plasma physics and numerical simulations, to the point of being comfortable with modifying it.

Given the complexity of the physical problem behind it, this goal is, sadly, unattainable, but the authors employ several avenues to get as close as possible:

1. Abstaining from using advanced programming concepts. Cool things like aspect-oriented programming are neat, but keeping the code well under 1000 SLOC is even neater. The two classes we currently have is two classes over the ideal amount of them.
2. Preferring less code over extensibility.
3. Picking simpler code over performance tweaking.
4. Choosing malleability over user convenience.
5. Creating external modules or branches over exhaustive featureset.
6. Not shying away from external dependencies or unpopular technologies, even if this means sacrificing the portability.
7. Appointing a physics student with modest programming background as the maintainer and primary code reviewer.

Codebase size

LCODE 3D wasn't always around 500 SLOC. In fact, even as it got rewritten from C to Cython to numba to numba.cuda to cupy, it peaked at around 5000 SLOC, twice. And we don't even count its Fortran days.

In order to objectively curb the complexity, scope and malleability of the codebase, its size is limited to 1000 SLOC.

David Wheeler's SLOCCount is used for obtaining the metric. Empty lines, docstrings and comments don't count towards that limit.

Zero special boundary treatment

[*Not allowing the particles to reach the outer cells of the simulation window* (page 6)] slightly modifies the physical problem itself, but, in return blesses us with the ability to forego special boundary checks during deposition, interpolation and field calculation, simplifying the code and boosting the performance.

Memory considerations

LCODE 3D is observed to consume roughly the same amount of host and GPU RAM, hovering around 500 MiB for a 641x641 grid, coarseness=3 and fineness=2.

The size of the arrays processed by LCODE 3D depends on these parameters.

Let's label the field/densities grid size in a single direction as N, coarse plasma grid size as $N_c \approx \frac{N}{\text{coarseness}}$ and fine plasma grid size as $N_f \approx N * \text{fineness}$.

With about the same amount of arrays in scope for each of these three sizes, it is clear that the N_f^2-sized arrays would dominate the memory consumption. Fortunately, the arrays that contain fine plasma characteristics would be transient and only used during the deposition, while the interpolation indices and coefficients grouped under `virt_params` can be reduced to 1D arrays by exploiting the x/y symmetry of the coarse/fine plasma grids.

This way LCODE 3D stores only N_c^2- and N^2-sized arrays, with N_f-sized ones barely taking up any space thanks to the being 1D.

Also, all previous attempts to micromanaged the GPU memory allocations have been scraped in favor of blindly trusting the `cupy` on-demand allocation. Not only it is extremely convenient, it's even more performant than our own solutions.

Integer xi steps

ξ-steps are integer for the purpose of bypassing float precision-based errors. The task of converting it into the ξ-coordinate is placed within the usage context.

CHAPTER 5

Extras:

5.1 Contributing

When contributing to this repository, please discuss the change you wish to make via email (team@lcode.info), issue tracker (https://github.com/lotov/lcode3d/issues), personal communication or any other method with our team.

The suggested followup workflow for the implementor would be:

- choose the most suitable parent branch;
- fork https://github.com/lotov/lcode3d or its fork;
- check it out locally;
- install dependencies (see `requirements.txt`);
- verify that LCODE runs as-is;
- implement, test and commit changes;
- check that the code is still under 1000 SLOC;
- try to strip all the complex programming concepts and clever hacks;
- rebase it if the parent branch advances;
- submit a pull request;
- wait for it to be rebased-and-merged.

By submitting patches to this project, you agree them to be redistributed under the project's license according to the normal forms and usages of the open-source community.

5.2 Example configuration file

```
grid_steps = 641  #: Transverse grid size in cells
grid_step_size = .025  #: Transverse grid step size in plasma units

xi_step_size = .005  #: Step size in time-space coordinate xi
xi_steps = int(3000 // xi_step_size)  #: Amount of xi steps

diagnostics_each_N_steps = int(1 / xi_step_size)

field_solver_subtraction_trick = 1  #: 0 for Laplace eqn., Helmholtz otherwise
field_solver_variant_A = True  #: Use Variant A or Variant B for Ex, Ey, Bx, By

reflect_padding_steps = 5  #: Plasma reflection <-> field calculation␣
↪boundaries
plasma_padding_steps = 10  #: Plasma placement <-> field calculation boundaries

plasma_coarseness = 3  #: Square root of the amount of cells per coarse␣
↪particle
plasma_fineness = 2  #: Square root of the amount of fine particles per cell


from numpy import cos, exp, pi, sqrt

def beam(xi_i, x, y):
    xi = -xi_i * xi_step_size
    COMPRESS, BOOST, SIGMA, SHIFT = 1, 1, 1, 0
    if xi < -2 * sqrt(2 * pi) / COMPRESS:
        return 0
    r = sqrt(x**2 + (y - SHIFT)**2)
    return (.05 * BOOST * exp(-.5 * (r / SIGMA)**2) *
            (1 - cos(xi * COMPRESS * sqrt(pi / 2))))

gpu_index = 0  #: Index of the GPU that should perform the calculations
```

5.3 The complete LCODE 3D source code

```
#!/usr/bin/env python3

# Copyright (c) 2016-2019 LCODE team <team@lcode.info>.

# LCODE is free software: you can redistribute it and/or modify
# it under the terms of the GNU Affero General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# LCODE is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU Affero General Public License for more details.
#
# You should have received a copy of the GNU Affero General Public License
# along with LCODE.  If not, see <http://www.gnu.org/licenses/>.


from math import sqrt, floor

import os
import sys

import matplotlib.pyplot as plt

import numpy as np

import numba
import numba.cuda

import cupy as cp

import scipy.ndimage
import scipy.signal


# Prevent all CPU cores waiting for the GPU at 100% utilization (under conda).
# os.environ['OMP_NUM_THREADS'] = '1'

#: Should be detectable with newer ``cupy`` (>6.0.0b2) as
#: ``WARP_SIZE = cp.cuda.Device(config.gpu_index).attributes['WarpSize']``.
#: As of 2019 it's equal to 32 for all CUDA-capable GPUs.
#: It's even a hardcoded value in ``cupy``.
WARP_SIZE = 32

ELECTRON_CHARGE = -1
ELECTRON_MASS = 1


# Grouping GPU arrays, with optional transparent RAM<->GPU copying #
```

(continues on next page)

(continued from previous page)

```
class GPUArrays:
    """
    A convenient way to group several GPU arrays and access them with a dot.
    ``x = GPUArrays(something=numpy_array, something_else=another_array)`` will
    create ``x`` with ``x.something`` and ``x.something_else`` stored on GPU.

    Do not add more attributes later, specify them all at construction time.
    """
    def __init__(self, **kwargs):
        """
        Convert the keyword arguments to ``cupy`` arrays and assign them
        to the object attributes.
        Amounts to, e.g., ``self.something = cp.asarray(numpy_array)``,
        and ``self.something_else = cp.asarray(another_array)``,
        see class doctring.
        """
        for name, array in kwargs.items():
            setattr(self, name, cp.asarray(array))


# NOTE: The implementation may be complicated, but the usage is simple.
class GPUArraysView:
    """
    This is a magical wrapper around GPUArrays that handles GPU-RAM data
    transfer transparently.
    Accessing ``view.something`` will automatically copy array to host RAM,
    setting ``view.something = ...`` will copy the changes back to GPU RAM.

    Usage: ``view = GPUArraysView(gpu_arrays); view.something``

    Do not add more attributes later, specify them all at construction time.

    NOTE: repeatedly accessing an attribute will result in repeated copying!
    """
    def __init__(self, gpu_arrays):
        """
        Wrap ``gpu_arrays`` and transparently copy data to/from GPU.
        """
        # Could've been written as ``self._arrs = gpu_arrays``
        # if only ``__setattr__`` was not overwritten!
        # ``super(GPUArraysView)`` is the proper way to obtain the parent class
        # (``object``), which has a regular boring and usable ``__setattr__``.
        super(GPUArraysView, self).__setattr__('_arrs', gpu_arrays)

    def __dir__(self):
        """
        Make ``dir()`` also show the wrapped ``gpu_arrays`` attributes.
        """
        # See ``GPUArraysView.__init__`` for the explanation how we access the
        # parent's plain ``__dir__()`` implementation (and avoid recursion).
        return list(set(super(GPUArraysView, self).__dir__() +
```

(continues on next page)

(continued from previous page)

```
                        dir(self._arrs)))

    def __getattr__(self, attrname):
        """
        Intercept access to (missing) attributes, access the wrapped object
        attributes instead and copy the arrays from GPU to RAM.
        """
        return getattr(self._arrs, attrname).get()  # auto-copies to host RAM

    def __setattr__(self, attrname, value):
        """
        Intercept setting attributes, access the wrapped object attributes
        instead and reassign their contents, copying the arrays from RAM
        to GPU in the process.
        """
        getattr(self._arrs, attrname)[...] = value  # copies to GPU RAM
        # TODO: just copy+reassign it without preserving identity and shape?


# Solving Laplace equation with Dirichlet boundary conditions (Ez) #

def dst2d(a):
    """
    Calculate DST-Type1-2D, jury-rigged from anti-symmetrically-padded rFFT.
    """
    assert a.shape[0] == a.shape[1]
    N = a.shape[0]
    #                                    / 0  0  0  0  0  0 \
    #  0  0  0  0                       |  0 /1  2\ 0 -2 -1  |
    #  0 /1  2\ 0   anti-symmetrically   |  0 \3  4/ 0 -4 -3  |
    #  0 \3  4/ 0       padded to        |  0  0  0  0  0  0  |
    #  0  0  0  0                       |  0 -3 -4  0 +4 +3  |
    #                                    \ 0 -1 -2  0 +2 +1 /
    p = cp.zeros((2 * N + 2, 2 * N + 2))
    p[1:N+1, 1:N+1], p[1:N+1, N+2:] = a,              -cp.fliplr(a)
    p[N+2:,  1:N+1], p[N+2:,  N+2:] = -cp.flipud(a), +cp.fliplr(cp.flipud(a))

    # after padding: rFFT-2D, cut out the top-left segment, take -real part
    return -cp.fft.rfft2(p)[1:N+1, 1:N+1].real


@cp.memoize()
def dirichlet_matrix(grid_steps, grid_step_size):
    """
    Calculate a magical matrix that solves the Laplace equation
    if you elementwise-multiply the RHS by it "in DST-space".
    See Samarskiy-Nikolaev, p. 187.
    """
    # mul[i, j] = 1 / (lam[i] + lam[j])
    # lam[k] = 4 / h**2 * sin(k * pi * h / (2 * L))**2, where L = h * (N - 1)
    k = cp.arange(1, grid_steps - 1)
    lam = 4 / grid_step_size**2 * cp.sin(k * cp.pi / (2 * (grid_steps - 1)))**2
```

(continues on next page)

(continued from previous page)

```
    lambda_i, lambda_j = lam[:, None], lam[None, :]
    mul = 1 / (lambda_i + lambda_j)
    return mul / (2 * (grid_steps - 1))**2  # additional 2xDST normalization


def calculate_Ez(config, jx, jy):
    """
    Calculate Ez as iDST2D(dirichlet_matrix * DST2D(djx/dx + djy/dy)).
    """
    # 0. Calculate RHS (NOTE: it is smaller by 1 on each side).
    # NOTE: use gradient instead if available (cupy doesn't have gradient yet).
    djx_dx = jx[2:, 1:-1] - jx[:-2, 1:-1]
    djy_dy = jy[1:-1, 2:] - jy[1:-1, :-2]
    rhs_inner = -(djx_dx + djy_dy) / (config.grid_step_size * 2)  # -?

    # 1. Apply DST-Type1-2D (Discrete Sine Transform Type 1 2D) to the RHS.
    f = dst2d(rhs_inner)

    # 2. Multiply f by the special matrix that does the job and normalizes.
    f *= dirichlet_matrix(config.grid_steps, config.grid_step_size)

    # 3. Apply iDST-Type1-2D (Inverse Discrete Sine Transform Type 1 2D).
    #    We don't have to define a separate iDST function, because
    #    unnormalized DST-Type1 is its own inverse, up to a factor 2(N+1)
    #    and we take all scaling matters into account with a single factor
    #    hidden inside dirichlet_matrix.
    Ez_inner = dst2d(f)
    Ez = cp.pad(Ez_inner, 1, 'constant', constant_values=0)
    numba.cuda.synchronize()
    return Ez


# Solving Laplace or Helmholtz equation with mixed boundary conditions #

# jury-rigged from padded rFFT
def mix2d(a):
    """
    Calculate a DST-DCT-hybrid transform
    (DST in first direction, DCT in second one),
    jury-rigged from padded rFFT
    (anti-symmetrically in first direction, symmetrically in second direction).
    """
    # NOTE: LCODE 3D uses x as the first direction, thus the confision below.
    M, N = a.shape
    #                                  /(0  1  2  0)-2 -1 \       +----->  x
    #  / 1  2 \                       | (0  3  4  0)-4 -3  |      |       (M)
    #  | 3  4 |  mixed-symmetrically  | (0  5  6  0)-6 -5  |      |
    #  | 5  6 |       padded to       | (0  7  8  0)-8 -7  |      v
    #  \ 7  8 /                       |  0 +5 +6  0 -6 -5  |
    #                                  \ 0 +3 +4  0 -4 -3 /       y (N)
    p = cp.zeros((2 * M + 2, 2 * N - 2))  # wider than before
    p[1:M+1, :N] = a
```

(continues on next page)

(continued from previous page)

```
    p[M+2:2*M+2, :N] = -cp.flipud(a)  # flip to right on drawing above
    p[1:M+1, N-1:2*N-2] = cp.fliplr(a)[:, :-1]  # flip down on drawing above
    p[M+2:2*M+2, N-1:2*N-2] = -cp.flipud(cp.fliplr(a))[:, :-1]
    # Note: the returned array is wider than the input array, it is padded
    # with zeroes (depicted above as a square region marked with round braces).
    return -cp.fft.rfft2(p)[:M+2, :N].imag  # FFT, cut a corner with 0s, -imag


@cp.memoize()
def mixed_matrix(grid_steps, grid_step_size, subtraction_trick):
    """
    Calculate a magical matrix that solves the Helmholtz or Laplace equation
    (subtraction_trick=True and subtraction_trick=False correspondingly)
    if you elementwise-multiply the RHS by it "in DST-DCT-transformed-space".
    See Samarskiy-Nikolaev, p. 189 and around.
    """
    # mul[i, j] = 1 / (lam[i] + lam[j])
    # lam[k] = 4 / h**2 * sin(k * pi * h / (2 * L))**2, where L = h * (N - 1)
    # but k for lam_i spans from 1..N-2, while k for lam_j covers 0..N-1
    ki, kj = cp.arange(1, grid_steps - 1), cp.arange(grid_steps)
    li = 4 / grid_step_size**2 * cp.sin(ki * cp.pi / (2 * (grid_steps - 1)))**2
    lj = 4 / grid_step_size**2 * cp.sin(kj * cp.pi / (2 * (grid_steps - 1)))**2
    lambda_i, lambda_j = li[:, None], lj[None, :]
    mul = 1 / (lambda_i + lambda_j + (1 if subtraction_trick else 0))
    return mul / (2 * (grid_steps - 1))**2  # additional 2xDST normalization


def dx_dy(arr, grid_step_size):
    """
    Calculate x and y derivatives simultaneously (like np.gradient does).
    NOTE: use gradient instead if available (cupy doesn't have gradient yet).
    NOTE: arrays are assumed to have zeros on the perimeter.
    """
    dx, dy = cp.zeros_like(arr), cp.zeros_like(arr)
    dx[1:-1, 1:-1] = arr[2:, 1:-1] - arr[:-2, 1:-1]  # arrays have 0s
    dy[1:-1, 1:-1] = arr[1:-1, 2:] - arr[1:-1, :-2]  # on the perimeter
    return dx / (grid_step_size * 2), dy / (grid_step_size * 2)


def calculate_Ex_Ey_Bx_By(config, Ex_avg, Ey_avg, Bx_avg, By_avg,
                          beam_ro, ro, jx, jy, jz, jx_prev, jy_prev):
    """
    Calculate transverse fields as iDST-DCT(mixed_matrix * DST-DCT(RHS.T)).T,
    with and without transposition depending on the field component.
    """
    # NOTE: density and currents are assumed to be zero on the perimeter
    # (no plasma particles must reach the wall, so the reflection boundary
    #  must be closer to the center than the simulation window boundary
    #  minus the coarse plasma particle cloud width).

    # 0. Calculate gradients and RHS.
    dro_dx, dro_dy = dx_dy(ro + beam_ro, config.grid_step_size)
```

(continues on next page)

(continued from previous page)

```
    djz_dx, djz_dy = dx_dy(jz + beam_ro, config.grid_step_size)
    djx_dxi = (jx_prev - jx) / config.xi_step_size  # - ?
    djy_dxi = (jy_prev - jy) / config.xi_step_size  # - ?

    # Are we solving a Laplace equation or a Helmholtz one?
    subtraction_trick = config.field_solver_subtraction_trick
    Ex_rhs = -((dro_dx - djx_dxi) - Ex_avg * subtraction_trick)  # -?
    Ey_rhs = -((dro_dy - djy_dxi) - Ey_avg * subtraction_trick)
    Bx_rhs = +((djz_dy - djy_dxi) + Bx_avg * subtraction_trick)
    By_rhs = -((djz_dx - djx_dxi) - By_avg * subtraction_trick)

    # Boundary conditions application (for future reference, ours are zero):
    # rhs[:, 0] -= bound_bottom[:] * (2 / grid_step_size)
    # rhs[:, -1] += bound_top[:] * (2 / grid_step_size)

    # 1. Apply our mixed DCT-DST transform to RHS.
    Ey_f = mix2d(Ey_rhs[1:-1, :])[1:-1, :]

    # 2. Multiply f by the magic matrix.
    mix_mat = mixed_matrix(config.grid_steps, config.grid_step_size,
                           config.field_solver_subtraction_trick)
    Ey_f *= mix_mat

    # 3. Apply our mixed DCT-DST transform again.
    Ey = mix2d(Ey_f)

    # Likewise for other fields:
    Bx = mix2d(mix_mat * mix2d(Bx_rhs[1:-1, :])[1:-1, :])
    By = mix2d(mix_mat * mix2d(By_rhs.T[1:-1, :])[1:-1, :]).T
    Ex = mix2d(mix_mat * mix2d(Ex_rhs.T[1:-1, :])[1:-1, :]).T

    return Ex, Ey, Bx, By


# Solving Laplace equation with Neumann boundary conditions (Bz) #

def dct2d(a):
    """
    Calculate DCT-Type1-2D, jury-rigged from symmetrically-padded rFFT.
    """
    assert a.shape[0] == a.shape[1]
    N = a.shape[0]
    #                                  //1  2  3  4\ 3  2 \
    # /1  2  3  4\                    | |5  6  7  8| 7  6  |
    # |5  6  7  8|     symmetrically    | |9  A  B  C| B  A  |
    # |9  A  B  C|      padded to       | \D  E  F  G/ F  E  |
    # \D  E  F  G/                    |  9  A  B  C  B  A  |
    #                                  \ 5  6  7  8  7  6 /
    p = cp.zeros((2 * N - 2, 2 * N - 2))
    p[:N, :N] = a
    p[N:, :N] = cp.flipud(a)[1:-1, :]  # flip to right on drawing above
    p[:N, N:] = cp.fliplr(a)[:, 1:-1]  # flip down on drawing above
```

(continues on next page)

(continued from previous page)

```
    p[N:, N:] = cp.flipud(cp.fliplr(a))[1:-1, 1:-1]  # bottom-right corner
    # after padding: rFFT-2D, cut out the top-left segment, take -real part
    return -cp.fft.rfft2(p)[:N, :N].real


@cp.memoize()
def neumann_matrix(grid_steps, grid_step_size):
    """
    Calculate a magical matrix that solves the Laplace equation
    if you elementwise-multiply the RHS by it "in DST-space".
    See Samarskiy-Nikolaev, p. 187.
    """
    # mul[i, j] = 1 / (lam[i] + lam[j])
    # lam[k] = 4 / h**2 * sin(k * pi * h / (2 * L))**2, where L = h * (N - 1)
    k = cp.arange(0, grid_steps)
    lam = 4 / grid_step_size**2 * cp.sin(k * cp.pi / (2 * (grid_steps - 1)))**2
    lambda_i, lambda_j = lam[:, None], lam[None, :]
    mul = 1 / (lambda_i + lambda_j)  # WARNING: zero division in mul[0, 0]!
    mul[0, 0] = 0  # doesn't matter anyway, just defines constant shift
    return mul / (2 * (grid_steps - 1))**2  # additional 2xDST normalization


def calculate_Bz(config, jx, jy):
    """
    Calculate Bz as iDCT2D(dirichlet_matrix * DCT2D(djx/dy - djy/dx)).
    """
    # 0. Calculate RHS.
    # NOTE: use gradient instead if available (cupy doesn't have gradient yet).
    djx_dy = jx[1:-1, 2:] - jx[1:-1, :-2]
    djy_dx = jy[2:, 1:-1] - jy[:-2, 1:-1]
    djx_dy = cp.pad(djx_dy, 1, 'constant', constant_values=0)
    djy_dx = cp.pad(djy_dx, 1, 'constant', constant_values=0)
    rhs = -(djx_dy - djy_dx) / (config.grid_step_size * 2)  # -?

    # As usual, the boundary conditions are zero
    # (otherwise add them to boundary cells, divided by grid_step_size/2

    # 1. Apply DST-Type1-2D (Discrete Sine Transform Type 1 2D) to the RHS.
    f = dct2d(rhs)

    # 2. Multiply f by the special matrix that does the job and normalizes.
    f *= neumann_matrix(config.grid_steps, config.grid_step_size)

    # 3. Apply iDCT-Type1-2D (Inverse Discrete Cosine Transform Type 1 2D).
    #    We don't have to define a separate iDCT function, because
    #    unnormalized DCT-Type1 is its own inverse, up to a factor 2(N+1)
    #    and we take all scaling matters into account with a single factor
    #    hidden inside neumann_matrix.
    Bz = dct2d(f)
    numba.cuda.synchronize()

    Bz -= Bz.mean()  # Integral over Bz must be 0.
```

(continues on next page)

(continued from previous page)

```
    return Bz


# Pushing particles without any fields (used for initial halfstep estimation) #

def move_estimate_wo_fields(config,
                            m, x_init, y_init, prev_x_offt, prev_y_offt,
                            px, py, pz):
    """
    Move coarse plasma particles as if there were no fields.
    Also reflect the particles from `+-reflect_boundary`.
    """
    x, y = x_init + prev_x_offt, y_init + prev_y_offt
    gamma_m = cp.sqrt(m**2 + pz**2 + px**2 + py**2)

    x += px / (gamma_m - pz) * config.xi_step_size
    y += py / (gamma_m - pz) * config.xi_step_size

    reflect = config.reflect_boundary
    x[x >= +reflect] = +2 * reflect - x[x >= +reflect]
    x[x <= -reflect] = -2 * reflect - x[x <= -reflect]
    y[y >= +reflect] = +2 * reflect - y[y >= +reflect]
    y[y <= -reflect] = -2 * reflect - y[y <= -reflect]

    x_offt, y_offt = x - x_init, y - y_init

    numba.cuda.synchronize()
    return x_offt, y_offt


# Deposition and interpolation helper functions #

@numba.jit(inline=True)
def weights(x, y, grid_steps, grid_step_size):
    """
    Calculate the indices of a cell corresponding to the coordinates,
    and the coefficients of interpolation and deposition for this cell
    and 8 surrounding cells.
    The weights correspond to 2D triangluar shaped cloud (TSC2D).
    """
    x_h, y_h = x / grid_step_size + .5, y / grid_step_size + .5
    i, j = int(floor(x_h) + grid_steps // 2), int(floor(y_h) + grid_steps // 2)
    x_loc, y_loc = x_h - floor(x_h) - .5, y_h - floor(y_h) - .5
    # centered to -.5 to 5, not 0 to 1, as formulas use offset from cell center
    # TODO: get rid of this deoffsetting/reoffsetting festival

    wx0, wy0 = .75 - x_loc**2, .75 - y_loc**2  # fx1, fy1
    wxP, wyP = (.5 + x_loc)**2 / 2, (.5 + y_loc)**2 / 2  # fx2**2/2, fy2**2/2
    wxM, wyM = (.5 - x_loc)**2 / 2, (.5 - y_loc)**2 / 2  # fx3**2/2, fy3**2/2

    wMP, w0P, wPP = wxM * wyP, wx0 * wyP, wxP * wyP
```

(continues on next page)

(continued from previous page)

```
    wM0, w00, wP0 = wxM * wy0, wx0 * wy0, wxP * wy0
    wMM, w0M, wPM = wxM * wyM, wx0 * wyM, wxP * wyM

    return i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM


@numba.jit(inline=True)
def interp9(a, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM):
    """
    Collect value from a cell and 8 surrounding cells (using `weights` output).
    """
    return (
        a[i - 1, j + 1] * wMP + a[i + 0, j + 1] * w0P + a[i + 1, j + 1] * wPP +
        a[i - 1, j + 0] * wM0 + a[i + 0, j + 0] * w00 + a[i + 1, j + 0] * wP0 +
        a[i - 1, j - 1] * wMM + a[i + 0, j - 1] * w0M + a[i + 1, j - 1] * wPM
    )


@numba.jit(inline=True)
def deposit9(a, i, j, val, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM):
    """
    Deposit value into a cell and 8 surrounding cells (using `weights` output).
    """
    # This is like a[i - 1, j + 1] += val * wMP, except it is atomic
    # and incrementing the same cell by several threads will add up correctly.
    # CUDA Compute Capability 6.0+ is recommended for hardware atomics support.
    numba.cuda.atomic.add(a, (i - 1, j + 1), val * wMP)
    numba.cuda.atomic.add(a, (i + 0, j + 1), val * w0P)
    numba.cuda.atomic.add(a, (i + 1, j + 1), val * wPP)
    numba.cuda.atomic.add(a, (i - 1, j + 0), val * wM0)
    numba.cuda.atomic.add(a, (i + 0, j + 0), val * w00)
    numba.cuda.atomic.add(a, (i + 1, j + 0), val * wP0)
    numba.cuda.atomic.add(a, (i - 1, j - 1), val * wMM)
    numba.cuda.atomic.add(a, (i + 0, j - 1), val * w0M)
    numba.cuda.atomic.add(a, (i + 1, j - 1), val * wPM)


# Coarse and fine plasma initialization #

def make_coarse_plasma_grid(steps, step_size, coarseness=3):
    """
    Create initial coarse plasma particles coordinates
    (a single 1D grid for both x and y).
    """
    assert coarseness == int(coarseness)  # TODO: why?
    plasma_step = step_size * coarseness
    right_half = np.arange(steps // (coarseness * 2)) * plasma_step
    left_half = -right_half[:0:-1]  # invert, reverse, drop zero
    plasma_grid = np.concatenate([left_half, right_half])
    assert(np.array_equal(plasma_grid, -plasma_grid[::-1]))
    return plasma_grid
```

(continues on next page)

(continued from previous page)

```
def make_fine_plasma_grid(steps, step_size, fineness=2):
    """
    Create initial fine plasma particles coordinates
    (a single 1D grid for both x and y).

    Avoids positioning particles at the cell edges and boundaries.

    .. See docs/how/fine_and_coarse_plasma for illustrations.
    """
    assert fineness == int(fineness)
    plasma_step = step_size / fineness
    if fineness % 2:  # some on zero axes, none on cell corners
        right_half = np.arange(steps // 2 * fineness) * plasma_step
        left_half = -right_half[:0:-1]  # invert, reverse, drop zero
    else:  # none on zero axes, none on cell corners
        right_half = (.5 + np.arange(steps // 2 * fineness)) * plasma_step
        left_half = -right_half[::-1]  # invert, reverse
    plasma_grid = np.concatenate([left_half, right_half])
    assert(np.array_equal(plasma_grid, -plasma_grid[::-1]))
    return plasma_grid


def make_plasma(steps, cell_size, coarseness=3, fineness=2):
    """
    Make coarse plasma initial state arrays and the arrays needed to intepolate
    coarse plasma into fine plasma (``virt_params``).

    Coarse is the one that will evolve and fine is the one to be bilinearly
    interpolated from the coarse one based on the initial positions
    (using 1 to 4 coarse plasma particles that initially were the closest).
    """
    coarse_step = cell_size * coarseness

    # Make two initial grids of plasma particles, coarse and fine.
    # Coarse is the one that will evolve and fine is the one to be bilinearly
    # interpolated from the coarse one based on the initial positions.

    coarse_grid = make_coarse_plasma_grid(steps, cell_size, coarseness)
    coarse_grid_xs, coarse_grid_ys = coarse_grid[:, None], coarse_grid[None, :]

    fine_grid = make_fine_plasma_grid(steps, cell_size, fineness)

    Nc = len(coarse_grid)

    # Create plasma electrons on the coarse grid, the ones that really move
    coarse_x_init = cp.broadcast_to(cp.asarray(coarse_grid_xs), (Nc, Nc))
    coarse_y_init = cp.broadcast_to(cp.asarray(coarse_grid_ys), (Nc, Nc))
    coarse_x_offt = cp.zeros((Nc, Nc))
    coarse_y_offt = cp.zeros((Nc, Nc))
    coarse_px = cp.zeros((Nc, Nc))
    coarse_py = cp.zeros((Nc, Nc))
```

(continues on next page)

(continued from previous page)

```
    coarse_pz = cp.zeros((Nc, Nc))
    coarse_m = cp.ones((Nc, Nc)) * ELECTRON_MASS * coarseness**2
    coarse_q = cp.ones((Nc, Nc)) * ELECTRON_CHARGE * coarseness**2

    # Calculate indices for coarse -> fine bilinear interpolation

    # Neighbour indices array, 1D, same in both x and y direction.
    indices = np.searchsorted(coarse_grid, fine_grid)
    # example:
    #     coarse:  [-2., -1.,  0.,  1.,  2.]
    #     fine:    [-2.4, -1.8, -1.2, -0.6,  0. ,  0.6,  1.2,  1.8,  2.4]
    #     indices: [ 0  ,  1  ,  1  ,  2  ,  2  ,  3  ,  4  ,  4  ,  5 ]
    # There is no coarse particle with index 5, so clip it to 4:
    indices_next = np.clip(indices, 0, Nc - 1)  # [0, 1, 1, 2, 2, 3, 4, 4, 4]
    # Clip to zero for indices of prev particles as well:
    indices_prev = np.clip(indices - 1, 0, Nc - 1)  # [0, 0, 0, 1 ... 3, 3, 4]
    # mixed from: [ 0&0 , 0&1 , 0&1 , 1&2 , 1&2 , 2&3 , 3&4 , 3&4, 4&4 ]

    # Calculate weights for coarse->fine interpolation from initial positions.
    # The further the fine particle is from closest right coarse particles,
    # the more influence the left ones have.
    influence_prev = (coarse_grid[indices_next] - fine_grid) / coarse_step
    influence_next = (fine_grid - coarse_grid[indices_prev]) / coarse_step
    # Fix for boundary cases of missing cornering particles.
    influence_prev[indices_next == 0] = 0   # nothing on the left?
    influence_next[indices_next == 0] = 1   # use right
    influence_next[indices_prev == Nc - 1] = 0  # nothing on the right?
    influence_prev[indices_prev == Nc - 1] = 1  # use left
    # Same arrays are used for interpolating in y-direction.

    # The virtualization formula is thus
    # influence_prev[pi] * influence_prev[pj] * <bottom-left neighbour value> +
    # influence_prev[pi] * influence_next[nj] * <top-left neighbour value> +
    # influence_next[ni] * influence_prev[pj] * <bottom-right neighbour val> +
    # influence_next[ni] * influence_next[nj] * <top-right neighbour value>
    # where pi, pj are indices_prev[i], indices_prev[j],
    #       ni, nj are indices_next[i], indices_next[j] and
    #       i, j are indices of fine virtual particles

    # This is what is employed inside mix() and deposit_kernel().

    # An equivalent formula would be
    # inf_prev[pi] * (inf_prev[pj] * <bot-left> + inf_next[nj] * <bot-right>) +
    # inf_next[ni] * (inf_prev[pj] * <top-left> + inf_next[nj] * <top-right>)

    # Values of m, q, px, py, pz should be scaled by 1/(fineness*coarseness)**2

    virt_params = GPUArrays(
        influence_prev=influence_prev, influence_next=influence_next,
        indices_prev=indices_prev, indices_next=indices_next,
        fine_grid=fine_grid,
    )
```

(continues on next page)

(continued from previous page)

```
    return (coarse_x_init, coarse_y_init, coarse_x_offt, coarse_y_offt,
            coarse_px, coarse_py, coarse_pz, coarse_m, coarse_q, virt_params)


@numba.jit(inline=True)
def mix(coarse, A, B, C, D, pi, ni, pj, nj):
    """
    Bilinearly interpolate fine plasma properties from four
    historically-neighbouring plasma particle property values::
        B    D   #  y ^         A - bottom-left  neighbour, indices: pi, pj
           .     #    |         B - top-left     neighbour, indices: pi, nj
                 #    +---->    C - bottom-right neighbour, indices: ni, pj
        A    C   #          x   D - top-right    neighbour, indices: ni, nj
    See the rest of the deposition and plasma creation for more info.
    """
    return (A * coarse[pi, pj] + B * coarse[pi, nj] +
            C * coarse[ni, pj] + D * coarse[ni, nj])


@numba.jit(inline=True)
def coarse_to_fine(fi, fj, c_x_offt, c_y_offt, c_m, c_q, c_px, c_py, c_pz,
                   virtplasma_smallness_factor, fine_grid,
                   influence_prev, influence_next, indices_prev, indices_next):
    """
    Bilinearly interpolate fine plasma properties from four
    historically-neighbouring plasma particle property values.
    """
    # Calculate the weights of the historically-neighbouring coarse particles
    A = influence_prev[fi] * influence_prev[fj]
    B = influence_prev[fi] * influence_next[fj]
    C = influence_next[fi] * influence_prev[fj]
    D = influence_next[fi] * influence_next[fj]
    # and retrieve their indices.
    pi, ni = indices_prev[fi], indices_next[fi]
    pj, nj = indices_prev[fj], indices_next[fj]

    # Now we're ready to mix the fine particle characteristics
    x_offt = mix(c_x_offt, A, B, C, D, pi, ni, pj, nj)
    y_offt = mix(c_y_offt, A, B, C, D, pi, ni, pj, nj)
    x = fine_grid[fi] + x_offt  # x_fine_init
    y = fine_grid[fj] + y_offt  # y_fine_init

    # TODO: const m and q
    m = virtplasma_smallness_factor * mix(c_m, A, B, C, D, pi, ni, pj, nj)
    q = virtplasma_smallness_factor * mix(c_q, A, B, C, D, pi, ni, pj, nj)

    px = virtplasma_smallness_factor * mix(c_px, A, B, C, D, pi, ni, pj, nj)
    py = virtplasma_smallness_factor * mix(c_py, A, B, C, D, pi, ni, pj, nj)
    pz = virtplasma_smallness_factor * mix(c_pz, A, B, C, D, pi, ni, pj, nj)
    return x, y, m, q, px, py, pz
```

(continues on next page)

(continued from previous page)

```
# Deposition #

@numba.cuda.jit
def deposit_kernel(grid_steps, grid_step_size, virtplasma_smallness_factor,
                   c_x_offt, c_y_offt, c_m, c_q, c_px, c_py, c_pz,  # coarse
                   fine_grid,
                   influence_prev, influence_next, indices_prev, indices_next,
                   out_ro, out_jx, out_jy, out_jz):
    """
    Interpolate coarse plasma into fine plasma and deposit it on the
    charge density and current grids.
    """
    # Do nothing if our thread does not have a fine particle to deposit.
    fk = numba.cuda.grid(1)
    if fk >= fine_grid.size**2:
        return
    fi, fj = fk // fine_grid.size, fk % fine_grid.size

    # Interpolate fine plasma particle from coarse particle characteristics
    x, y, m, q, px, py, pz = coarse_to_fine(fi, fj, c_x_offt, c_y_offt,
                                            c_m, c_q, c_px, c_py, c_pz,
                                            virtplasma_smallness_factor,
                                            fine_grid,
                                            influence_prev, influence_next,
                                            indices_prev, indices_next)

    # Deposit the resulting fine particle on ro/j grids.
    gamma_m = sqrt(m**2 + px**2 + py**2 + pz**2)
    dro = q / (1 - pz / gamma_m)
    djx = px * (dro / gamma_m)
    djy = py * (dro / gamma_m)
    djz = pz * (dro / gamma_m)

    i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM = weights(
        x, y, grid_steps, grid_step_size
    )
    deposit9(out_ro, i, j, dro, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    deposit9(out_jx, i, j, djx, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    deposit9(out_jy, i, j, djy, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    deposit9(out_jz, i, j, djz, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)


def deposit(config, ro_initial, x_offt, y_offt, m, q, px, py, pz, virt_params):
    """
    Interpolate coarse plasma into fine plasma and deposit it on the
    charge density and current grids.
    This is a convenience wrapper around the ``deposit_kernel`` CUDA kernel.
    """
    virtplasma_smallness_factor = 1 / (config.plasma_coarseness *
                                       config.plasma_fineness)**2
    ro = cp.zeros((config.grid_steps, config.grid_steps))
```

(continues on next page)

(continued from previous page)

```
    jx = cp.zeros((config.grid_steps, config.grid_steps))
    jy = cp.zeros((config.grid_steps, config.grid_steps))
    jz = cp.zeros((config.grid_steps, config.grid_steps))
    cfg = int(np.ceil(virt_params.fine_grid.size**2 / WARP_SIZE)), WARP_SIZE
    deposit_kernel[cfg](config.grid_steps, config.grid_step_size,
                        virtplasma_smallness_factor,
                        x_offt, y_offt, m, q, px, py, pz,
                        virt_params.fine_grid,
                        virt_params.influence_prev, virt_params.influence_next,
                        virt_params.indices_prev, virt_params.indices_next,
                        ro, jx, jy, jz)
    # Also add the background ion charge density.
    ro += ro_initial  # Do it last to preserve more float precision
    numba.cuda.synchronize()
    return ro, jx, jy, jz


def initial_deposition(config, x_offt, y_offt, px, py, pz, m, q, virt_params):
    """
    Determine the background ion charge density by depositing the electrons
    with their initial parameters and negating the result.
    """
    ro_electrons_initial, _, _, _ = deposit(config, 0, x_offt, y_offt,
                                            m, q, px, py, pz, virt_params)
    return -ro_electrons_initial  # Right on the GPU, huh


# Field interpolation and particle movement (fused) #

@numba.cuda.jit
def move_smart_kernel(xi_step_size, reflect_boundary,
                      grid_step_size, grid_steps,
                      ms, qs,
                      x_init, y_init,
                      prev_x_offt, prev_y_offt,
                      estimated_x_offt, estimated_y_offt,
                      prev_px, prev_py, prev_pz,
                      Ex_avg, Ey_avg, Ez_avg, Bx_avg, By_avg, Bz_avg,
                      new_x_offt, new_y_offt, new_px, new_py, new_pz):
    """
    Update plasma particle coordinates and momenta according to the field
    values interpolated halfway between the previous plasma particle location
    and the the best estimation of its next location currently available to us.
    Also reflect the particles from ``+-reflect_boundary``.
    """
    # Do nothing if our thread does not have a coarse particle to move.
    k = numba.cuda.grid(1)
    if k >= ms.size:
        return

    m, q = ms[k], qs[k]
```

(continues on next page)

(continued from previous page)

```
    opx, opy, opz = prev_px[k], prev_py[k], prev_pz[k]
    px, py, pz = opx, opy, opz
    x_offt, y_offt = prev_x_offt[k], prev_y_offt[k]

    # Calculate midstep positions and fields in them.
    x_halfstep = x_init[k] + (prev_x_offt[k] + estimated_x_offt[k]) / 2
    y_halfstep = y_init[k] + (prev_y_offt[k] + estimated_y_offt[k]) / 2
    i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM = weights(
        x_halfstep, y_halfstep, grid_steps, grid_step_size
    )
    Ex = interp9(Ex_avg, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    Ey = interp9(Ey_avg, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    Ez = interp9(Ez_avg, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    Bx = interp9(Bx_avg, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    By = interp9(By_avg, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)
    Bz = interp9(Bz_avg, i, j, wMP, w0P, wPP, wM0, w00, wP0, wMM, w0M, wPM)

    # Move the particles according the the fields
    gamma_m = sqrt(m**2 + pz**2 + px**2 + py**2)
    vx, vy, vz = px / gamma_m, py / gamma_m, pz / gamma_m
    factor_1 = q * xi_step_size / (1 - pz / gamma_m)
    dpx = factor_1 * (Ex + vy * Bz - vz * By)
    dpy = factor_1 * (Ey - vx * Bz + vz * Bx)
    dpz = factor_1 * (Ez + vx * By - vy * Bx)
    px, py, pz = opx + dpx / 2, opy + dpy / 2, opz + dpz / 2

    # Move the particles according the the fields again using updated momenta
    gamma_m = sqrt(m**2 + pz**2 + px**2 + py**2)
    vx, vy, vz = px / gamma_m, py / gamma_m, pz / gamma_m
    factor_1 = q * xi_step_size / (1 - pz / gamma_m)
    dpx = factor_1 * (Ex + vy * Bz - vz * By)
    dpy = factor_1 * (Ey - vx * Bz + vz * Bx)
    dpz = factor_1 * (Ez + vx * By - vy * Bx)
    px, py, pz = opx + dpx / 2, opy + dpy / 2, opz + dpz / 2

    # Apply the coordinate and momenta increments
    gamma_m = sqrt(m**2 + pz**2 + px**2 + py**2)

    x_offt += px / (gamma_m - pz) * xi_step_size  # no mixing with x_init
    y_offt += py / (gamma_m - pz) * xi_step_size  # no mixing with y_init

    px, py, pz = opx + dpx, opy + dpy, opz + dpz

    # Reflect the particles from `+-reflect_boundary`.
    # TODO: avoid branching?
    x = x_init[k] + x_offt
    y = y_init[k] + y_offt
    if x > +reflect_boundary:
        x = +2 * reflect_boundary - x
        x_offt = x - x_init[k]
        px = -px
    if x < -reflect_boundary:
```

(continues on next page)

(continued from previous page)

```
        x = -2 * reflect_boundary - x
        x_offt = x - x_init[k]
        px = -px
    if y > +reflect_boundary:
        y = +2 * reflect_boundary - y
        y_offt = y - y_init[k]
        py = -py
    if y < -reflect_boundary:
        y = -2 * reflect_boundary - y
        y_offt = y - y_init[k]
        py = -py

    # Save the results into the output arrays  # TODO: get rid of that
    new_x_offt[k], new_y_offt[k] = x_offt, y_offt
    new_px[k], new_py[k], new_pz[k] = px, py, pz


def move_smart(config,
               m, q, x_init, y_init, x_prev_offt, y_prev_offt,
               estimated_x_offt, estimated_y_offt, px_prev, py_prev, pz_prev,
               Ex_avg, Ey_avg, Ez_avg, Bx_avg, By_avg, Bz_avg):
    """
    Update plasma particle coordinates and momenta according to the field
    values interpolated halfway between the previous plasma particle location
    and the the best estimation of its next location currently available to us.
    This is a convenience wrapper around the ``move_smart_kernel`` CUDA kernel.
    """
    x_offt_new = cp.zeros_like(x_prev_offt)
    y_offt_new = cp.zeros_like(y_prev_offt)
    px_new = cp.zeros_like(px_prev)
    py_new = cp.zeros_like(py_prev)
    pz_new = cp.zeros_like(pz_prev)
    cfg = int(np.ceil(x_init.size / WARP_SIZE)), WARP_SIZE
    move_smart_kernel[cfg](config.xi_step_size, config.reflect_boundary,
                           config.grid_step_size, config.grid_steps,
                           m.ravel(), q.ravel(),
                           x_init.ravel(), y_init.ravel(),
                           x_prev_offt.ravel(), y_prev_offt.ravel(),
                           estimated_x_offt.ravel(), estimated_y_offt.ravel(),
                           px_prev.ravel(), py_prev.ravel(), pz_prev.ravel(),
                           Ex_avg, Ey_avg, Ez_avg, Bx_avg, By_avg, Bz_avg,
                           x_offt_new.ravel(), y_offt_new.ravel(),
                           px_new.ravel(), py_new.ravel(), pz_new.ravel())
    numba.cuda.synchronize()
    return x_offt_new, y_offt_new, px_new, py_new, pz_new


# The scheme of a single step in xi #

def step(config, const, virt_params, prev, beam_ro):
    """
    Calculate the next iteration of plasma evolution and response.
```

(continues on next page)

(continued from previous page)

```
    Returns the new state with the following attributes:
    ``x_offt``, ``y_offt``, ``px``, ``py``, ``pz``,
    ``Ex``, ``Ey``, ``Ez``, ``Bx``, ``By``, ``Bz``,
    ``ro``, ``jx``, ``jy``, ``jz``.
    Pass the returned value as ``prev`` for the next iteration.
    Wrap it in ``GPUArraysView`` if you want transparent conversion
    to ``numpy`` arrays.
    """
    beam_ro = cp.asarray(beam_ro)  # copy the array is on GPU if it's not there

    # Estimate the midpoint particle position without knowing the fields yet
    # TODO: use regular pusher and pass zero fields? previous fields?
    x_offt, y_offt = move_estimate_wo_fields(config, const.m,
                                             const.x_init, const.y_init,
                                             prev.x_offt, prev.y_offt,
                                             prev.px, prev.py, prev.pz)

    # Interpolate fields in midpoint and move particles with previous fields.
    x_offt, y_offt, px, py, pz = move_smart(
        config, const.m, const.q, const.x_init, const.y_init,
        prev.x_offt, prev.y_offt, x_offt, y_offt, prev.px, prev.py, prev.pz,
        # no halfstep-averaged fields yet
        prev.Ex, prev.Ey, prev.Ez, prev.Bx, prev.By, prev.Bz
    )
    # Recalculate the plasma density and currents.
    ro, jx, jy, jz = deposit(
        config, const.ro_initial, x_offt, y_offt, const.m, const.q, px, py, pz,
        virt_params
    )

    # Calculate the fields.
    ro_in = ro if not config.field_solver_variant_A else (ro + prev.ro) / 2
    jz_in = jz if not config.field_solver_variant_A else (jz + prev.jz) / 2
    Ex, Ey, Bx, By = calculate_Ex_Ey_Bx_By(config,
                                           prev.Ex, prev.Ey, prev.Bx, prev.By,
                                           # no halfstep-averaged fields yet
                                           beam_ro, ro_in, jx, jy, jz_in,
                                           prev.jx, prev.jy)
    if config.field_solver_variant_A:
        Ex, Ey = 2 * Ex - prev.Ex, 2 * Ey - prev.Ey
        Bx, By = 2 * Bx - prev.Bx, 2 * By - prev.By

    Ez = calculate_Ez(config, jx, jy)
    Bz = calculate_Bz(config, jx, jy)

    Ex_avg = (Ex + prev.Ex) / 2
    Ey_avg = (Ey + prev.Ey) / 2
    Ez_avg = (Ez + prev.Ez) / 2
    Bx_avg = (Bx + prev.Bx) / 2
    By_avg = (By + prev.By) / 2
    Bz_avg = (Bz + prev.Bz) / 2
```

(continues on next page)

(continued from previous page)

```
    # Repeat the previous procedure using averaged fields.
    x_offt, y_offt, px, py, pz = move_smart(
        config, const.m, const.q, const.x_init, const.y_init,
        prev.x_offt, prev.y_offt, x_offt, y_offt,
        prev.px, prev.py, prev.pz,
        Ex_avg, Ey_avg, Ez_avg, Bx_avg, By_avg, Bz_avg
    )
    ro, jx, jy, jz = deposit(config, const.ro_initial, x_offt, y_offt,
                             const.m, const.q, px, py, pz, virt_params)

    ro_in = ro if not config.field_solver_variant_A else (ro + prev.ro) / 2
    jz_in = jz if not config.field_solver_variant_A else (jz + prev.jz) / 2
    Ex, Ey, Bx, By = calculate_Ex_Ey_Bx_By(config,
                                           Ex_avg, Ey_avg, Bx_avg, By_avg,
                                           beam_ro, ro_in, jx, jy, jz_in,
                                           prev.jx, prev.jy)
    if config.field_solver_variant_A:
        Ex, Ey = 2 * Ex - prev.Ex, 2 * Ey - prev.Ey
        Bx, By = 2 * Bx - prev.Bx, 2 * By - prev.By

    Ez = calculate_Ez(config, jx, jy)
    Bz = calculate_Bz(config, jx, jy)

    Ex_avg = (Ex + prev.Ex) / 2
    Ey_avg = (Ey + prev.Ey) / 2
    Ez_avg = (Ez + prev.Ez) / 2
    Bx_avg = (Bx + prev.Bx) / 2
    By_avg = (By + prev.By) / 2
    Bz_avg = (Bz + prev.Bz) / 2

    # Repeat the previous procedure using averaged fields once again.
    x_offt, y_offt, px, py, pz = move_smart(
        config, const.m, const.q, const.x_init, const.y_init,
        prev.x_offt, prev.y_offt, x_offt, y_offt,
        prev.px, prev.py, prev.pz,
        Ex_avg, Ey_avg, Ez_avg, Bx_avg, By_avg, Bz_avg
    )
    ro, jx, jy, jz = deposit(config, const.ro_initial, x_offt, y_offt,
                             const.m, const.q, px, py, pz, virt_params)

    # TODO: what do we need that roj_new for, jx_prev/jy_prev only?

    # Return the array collection that would serve as `prev` for the next step.
    new_state = GPUArrays(x_offt=x_offt, y_offt=y_offt, px=px, py=py, pz=pz,
                          Ex=Ex.copy(), Ey=Ey.copy(), Ez=Ez.copy(),
                          Bx=Bx.copy(), By=By.copy(), Bz=Bz.copy(),
                          ro=ro, jx=jx, jy=jy, jz=jz)

    return new_state


# Array initialization #
```

(continues on next page)

(continued from previous page)

```
def init(config):
    """
    Initialize all the arrays needed for ``step`` and ``config.beam``.
    """

    assert config.grid_steps % 2 == 1

    # virtual particles should not reach the window pre-boundary cells
    assert config.reflect_padding_steps > config.plasma_coarseness + 1
    # the (costly) alternative is to reflect after plasma virtualization

    config.reflect_boundary = config.grid_step_size * (
        config.grid_steps / 2 - config.reflect_padding_steps
    )

    grid = ((np.arange(config.grid_steps) - config.grid_steps // 2)
            * config.grid_step_size)
    xs, ys = grid[:, None], grid[None, :]

    x_init, y_init, x_offt, y_offt, px, py, pz, m, q, virt_params = \
        make_plasma(config.grid_steps - config.plasma_padding_steps * 2,
                    config.grid_step_size,
                    coarseness=config.plasma_coarseness,
                    fineness=config.plasma_fineness)

    ro_initial = initial_deposition(config, x_offt, y_offt,
                                    px, py, pz, m, q, virt_params)

    const = GPUArrays(m=m, q=q, x_init=x_init, y_init=y_init,
                      ro_initial=ro_initial)

    def zeros():
        return cp.zeros((config.grid_steps, config.grid_steps))

    state = GPUArrays(x_offt=x_offt, y_offt=y_offt, px=px, py=py, pz=pz,
                      Ex=zeros(), Ey=zeros(), Ez=zeros(),
                      Bx=zeros(), By=zeros(), Bz=zeros(),
                      ro=zeros(), jx=zeros(), jy=zeros(), jz=zeros())

    return xs, ys, const, virt_params, state


# Some really sloppy diagnostics #

max_zn = 0
def diags_ro_zn(config, ro):
    global max_zn

    sigma = 0.25 / config.grid_step_size
    blurred = scipy.ndimage.gaussian_filter(ro, sigma=sigma)
    hf = ro - blurred
```

(continues on next page)

(continued from previous page)

```
    zn = np.abs(hf).mean() / 4.23045376e-04
    max_zn = max(max_zn, zn)
    return max_zn


def diags_peak_msg(Ez_00_history):
    Ez_00_array = np.array(Ez_00_history)
    peak_indices = scipy.signal.argrelmax(Ez_00_array)[0]

    if peak_indices.size:
        peak_values = Ez_00_array[peak_indices]
        rel_deviations_perc = 100 * (peak_values / peak_values[0] - 1)
        return (f'{peak_values[-1]:0.4e} '
                f'{rel_deviations_perc[-1]:+0.2f}%')
    else:
        return '...'


def diags_ro_slice(config, xi_i, xi, ro):
    if xi_i % int(1 / config.xi_step_size):
        return
    if not os.path.isdir('transverse'):
        os.mkdir('transverse')

    fname = f'ro_{xi:+09.2f}.png' if xi else 'ro_-00000.00.png'
    plt.imsave(os.path.join('transverse', fname), ro.T,
               origin='lower', vmin=-0.1, vmax=0.1, cmap='bwr')


def diagnostics(view_state, config, xi_i, Ez_00_history):
    xi = -xi_i * config.xi_step_size

    Ez_00 = Ez_00_history[-1]
    peak_report = diags_peak_msg(Ez_00_history)

    ro = view_state.ro
    max_zn = diags_ro_zn(config, ro)
    diags_ro_slice(config, xi_i, xi, ro)

    print(f'xi={xi:+.4f} {Ez_00:+.4e}|{peak_report}|zn={max_zn:.3f}')
    sys.stdout.flush()


# Main loop #

def main():
    import config
    with cp.cuda.Device(config.gpu_index):

        xs, ys, const, virt_params, state = init(config)
        Ez_00_history = []
```

(continues on next page)

(continued from previous page)

```
        for xi_i in range(config.xi_steps):
            beam_ro = config.beam(xi_i, xs, ys)

            state = step(config, const, virt_params, state, beam_ro)
            view_state = GPUArraysView(state)

            ez = view_state.Ez[config.grid_steps // 2, config.grid_steps // 2]
            Ez_00_history.append(ez)

            time_for_diags = xi_i % config.diagnostics_each_N_steps == 0
            last_step = xi_i == config.xi_steps - 1
            if time_for_diags or last_step:
                diagnostics(view_state, config, xi_i, Ez_00_history)


if __name__ == '__main__':
    main()
```

Index