Runko: Modern multi-physics toolbox for plasma simulations

J. Näätäli\textsuperscript{1,2,3}

\textsuperscript{1} Physics Department and Columbia Astrophysics Laboratory, Columbia University, 538 West 120th Street, New York, NY 10027, USA
\textsuperscript{2} Center for Computational Astrophysics, Flatiron Institute, 162 Fifth Avenue, New York, NY 10010, USA
\textsuperscript{3} Nordita, KTH Royal Institute of Technology and Stockholm University, Roslagstullsbacken 23, SE-10691 Stockholm, Sweden. e-mail: jan2174@columbia.edu

Received XXX / Accepted XXX

ABSTRACT

Runko is a new open-source plasma simulation framework implemented in C++ and Python. It is designed to function as an easy-to-extract general toolbox for simulating astrophysical plasmas with different theoretical and numerical models. Computationally intensive low-level “kernels” are written in modern C++ taking advantage of polymorphic classes, multiple inheritance, and template metaprogramming. High-level functionality is operated with Python scripts. The hybrid program design ensures good code performance together with ease of use. The framework has a modular object-oriented design that allow the user to easily add new numerical algorithms to the system. The code can be run on various computing platforms ranging from laptops (shared-memory systems) to massively parallel supercomputer architectures (distributed-memory systems). The framework supports heterogeneous multi-physics simulations in which different physical solvers can be combined and run simultaneously. Here we showcase the usage of the framework’s relativistic particle-in-cell (PIC) module by presenting (i) 1D simulations of relativistic Weibel instability, (ii) 2D simulations of relativistic kinetic turbulence in a suddenly stirred magnetically-dominated pair plasma, and (iii) 3D simulations of collisionless shocks in an unmagnetized medium.

Key words. Plasmas – Turbulence – Methods: numerical

1. Introduction

Ever since the introduction of computers, numerical simulations have been used to study nonlinear behavior of plasma (see e.g., \textsuperscript{1}Buneman\textsuperscript{2}\textsuperscript{3}\textsuperscript{4}1959 \textsuperscript{5}Dawson\textsuperscript{6}\textsuperscript{7}\textsuperscript{8}1962 \textsuperscript{9}1964). In the early days, the research was mainly motivated by studies of basic plasma instabilities and confinement in fusion experiments but the method of computationally solving the dynamics of charged particles quickly gained popularity also in understanding plasma in space (see e.g., \textsuperscript{10}Tanaka\textsuperscript{11}\textsuperscript{12}\textsuperscript{13}1983 \textsuperscript{14}Langdon et al.\textsuperscript{15}1988 \textsuperscript{16}Buneman et al.\textsuperscript{17}1992).

Important factor in accelerating the usage of plasma simulations has been the ever-increasing computational speed and number of processors. In the recent years we have reached the petaflop supercomputing era ($10^{15}$ floating-point operations per second; FLOPS) and started to pave our way towards exascale computing systems ($10^{18}$ FLOPS): this technological surge (and of course ingenuity of the researchers themselves) has enabled us to shed light on many longstanding issues in, for example, high-energy astrophysics, including shocks (e.g., \textsuperscript{18}Frederiksen et al.\textsuperscript{19}2004 \textsuperscript{19}Spitkovsky\textsuperscript{20}2005), reconnection (e.g., \textsuperscript{21}Zenitani & Hoshino\textsuperscript{22}2001 \textsuperscript{23}Cerutti et al.\textsuperscript{24}2012 \textsuperscript{25}Kagan et al.\textsuperscript{26}2013 \textsuperscript{27}Sironi & Spitkovsky\textsuperscript{28}2014 \textsuperscript{28}Werner et al.\textsuperscript{29}2015), pulsar magnetospheres (e.g., \textsuperscript{30}Philippov & Spitkovsky\textsuperscript{31}2014 \textsuperscript{31}Chen & Beloborodov\textsuperscript{32}2014 \textsuperscript{32}Cerutti et al.\textsuperscript{33}2015), and kinetic turbulence (e.g., \textsuperscript{34}Howes et al.\textsuperscript{35}2008 \textsuperscript{36}Servidio et al.\textsuperscript{37}2014 \textsuperscript{37}Zhdankin et al.\textsuperscript{38}2017 \textsuperscript{38}Comisso & Sironi\textsuperscript{39}2018).

In order to keep riding this technological wave, successful numerical codes need to be highly scalable and modular in order to take advantage of the newest supercomputing architectures. These kinds of exascale-ready features include e.g., avoidance of global communication patterns and support of heterogeneous computing nodes with varying resources such as different number of cores, accelerators, memory (see also Nordlund et al.\textsuperscript{40}2018 for a discussion). The presented Runko framework is developed from scratch to support such features.

From the modeling perspective, dynamics of plasma can be described with many methods and assumptions (see e.g., \textsuperscript{41}Choudhuri\textsuperscript{42}1998 for an introduction). One of the most fundamental ways is to consider a fully-kinetic description. In this case we solve the time evolution of a distribution of individual particles interacting with each other and electromagnetic fields (Boltzmann equation). If the particle mean free path is long (i.e. collisions are practically nonexistent), only the collective fields created by the moving charged particles themselves are dynamically important (Vlasov-Maxwell system of equations). Such a system can be modeled by solving the partial differential equations directly on a grid (so-called Vlasov method; Cheng & Knorr\textsuperscript{43}1976) or by sampling the distribution with numerical superparticles (so-called particle-in-cell or PIC approach; Hockney & Eastwood\textsuperscript{44}1981 \textsuperscript{44}Birdsall & Langdon\textsuperscript{45}1985). In case of a strong (background) magnetic field, it is possible to average over the gyrorotation of the particle’s orbit. This is known as the gyrokinetic description (see e.g., \textsuperscript{46}Howes et al.\textsuperscript{47}2004).

The degrees of freedom in the kinetic system can be reduced further by using the Chapman-Enskog expansion (Chapman & Cowling\textsuperscript{48}1970) that provides a passage from...
a microscopic particle-based model to a continuum fluid-like description. This macro-scale description relies on moments of the distribution function that provide a closure for the plasma equations, and is related to the magnetohydrodynamic (MHD) description (see e.g., [Brandenburg 2003]). The MHD description itself can also be divided into multiple categories with different closure approximations or by following the evolution of not one but multiple fluids (so-called multifluid description). Finally, there exists a class of hybrid methods that combine different models from above, for example, by treating electrons with fluid description and protons with the kinetic description (see e.g., [Winske et al. 2003; Valentini et al. 2007; Haggerty & Caprioli 2019]).

From the theoretical point of view, it is clear that all the aforementioned descriptions connect to each other and form a physical hierarchy of systems—same is not true from the numerical modeling point of view as plethora of codes exists fixating on one specific description. In order to remedy this problem, a similar form of software abstraction is needed as in the mathematical formulation of the problem. This is the core idea of the runko simulation framework. runko is built using modern computer science methods and paradigms that allow sufficient degree of problem generalization to technically realize this.

From the technical perspective, the presented code design provides all the needed tools to create hierarchical modeling systems of (coupled) formalisms. Technically this is achieved by polymorphic multiple inheritance as supported by the usage of modern C++. The concept of object inheritance nicely reflects the mathematical way of constructing hierarchical models that build on top of the previous one. Another level of abstraction is obtained by a usage of template metaprogramming that allows, e.g., to parameterize the dimensionality of the system—again closely reflecting the mathematical form of the problem. Finally, in order to balance the computational efficiency and the user-friendliness of the code, the fast low-level C++ classes are exposed to high-level PYTHON3 scripting language.

The complete framework is provided as a free open source software for the community. It is available from a GitHub repository and the code design and structure are presented in detail in this article. Here we also present the PIC code of the framework and describe the implementation of the solvers needed to operate it. Firstly, the theoretical background of kinetic modeling of plasma is shortly reviewed in Sect. 2. The numerical implementation of the framework is then discussed in Sect. 3. As an example, the usage of the new PIC code is demonstrated on astrophysical plasma problems. Discussion of new features and future directions of the framework are given in Sect. 4. Lastly, the content is summarized in Sect. 5.

2. Theory

2.1. Kinetic plasma theory

Let us present the special relativistic formulation of the Vlasov/Boltzmann equation. Spatial coordinate location vector is given by \( \mathbf{x} \equiv (x, y, z) \) and coordinate time is measured with \( t \). Coordinate velocity (three-component) is given by \( \mathbf{v} \equiv \mathbf{dx} \) and the individual Cartesian components of the velocity are denoted as \( v = (v_x, v_y, v_z) \). Proper time (measured with a co-moving clock), \( \tau \), is connected to the coordinate time with the Lorentz factor \( \gamma \equiv d \tau / dt \). The proper velocity (spatial components of the four-velocity) is \( \mathbf{u} \equiv d \mathbf{x} / d \tau = \gamma \mathbf{v} \).

Lorentz factor and the velocities are connected by the expression

\[
\gamma^2 = 1 + (u/c)^2 = (1 - (v/c)^2)^{-1},
\]

where \( c \) is the speed of light, \( u = |\mathbf{u}| \) and \( v = |\mathbf{v}| \). Acceleration is denoted with \( \mathbf{a} \equiv d \mathbf{u} / dt \).

Six-dimensional phase-space density distribution for particle species \( s \) is given by \( f_s \equiv f_s(\mathbf{x}, \mathbf{u}, t) \). Thus, \( f_s d^3 x d^3 u \) is the number of particles in the six-dimensional differential phase space volume between \( \mathbf{x}, \mathbf{u} \) and \( \mathbf{x} + d \mathbf{x}, \mathbf{u} + d \mathbf{u} \).

Evolution of \( f_s \) is governed by the relativistic Boltzmann/Vlasov equation

\[
\frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \nabla f_s + \mathbf{a} \cdot \nabla u f_s = C,
\]

where \( \nabla_x \equiv \frac{\partial}{\partial x} \) and \( \nabla_u \equiv \frac{\partial}{\partial u} \) are the spatial and momentum parts of the differential operator \( \nabla \), respectively. The term in the right-hand side, defined as \( C \equiv \partial f_s / \partial \mathbf{u} \big|_{\text{coll}} \), is the collision operator. For Vlasov equation \( C = 0 \), i.e., the plasma is collisionless.

Acceleration of a charged particle, \( \mathbf{a}_s \), is governed by the Lorentz force

\[
\mathbf{a}_s \equiv \partial \mathbf{u} / \partial t = \frac{q_s}{m_s} \left( \mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B} \right) = \frac{q_s}{m_s} \left( \mathbf{E} + \frac{\mathbf{u}}{\gamma c} \times \mathbf{B} \right),
\]

where \( \mathbf{E} \) and \( \mathbf{B} \) are the electric and magnetic fields, \( q_s \) is the charge, and \( m_s \) is the mass of the particle of species \( s \).

Moments of the distribution function define macroscopic (bulk) quantities of the plasma. Zeroth moment of the distribution function \( f_s \) defines the charge density of species \( s \) as

\[
\rho_s \equiv q_s \int f_s \, d\mathbf{u}.
\]

Total charge density is \( \rho \equiv \sum_s \rho_s \). The first moment defines the current (charge flux) vector as

\[
\mathbf{J}_s \equiv q_s \int f_s \, \mathbf{u} \, d\mathbf{u} = q_s \int f_s \frac{\mathbf{u}}{\gamma} \, d\mathbf{u}.
\]

Total current is \( \mathbf{J} \equiv \sum_s \mathbf{J}_s \).

2.2. Maxwell’s equations

Evolution of electric field \( \mathbf{E} \) and magnetic field \( \mathbf{B} \) is governed by the Maxwell’s equations. These are the Gauss’ law,

\[
\nabla \cdot \mathbf{E} = 4\pi \rho,
\]

Gauss’ law for magnetism,

\[
\nabla \cdot \mathbf{B} = 0,
\]

Ampere’s law,

\[
\nabla \times \mathbf{E} = -\frac{1}{c} \frac{\partial \mathbf{B}}{\partial t},
\]

and Faraday’s law,

\[
\nabla \times \mathbf{B} = \frac{4\pi}{c} \mathbf{J} + \frac{1}{c} \frac{\partial \mathbf{E}}{\partial t}.
\]

Charge conservation follows from these by taking a divergence of Eq. 10 and substituting Eq. 6 to get

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{J} = 0.
\]
2.3. Dimensionless equations

Let us now describe the actual dimensionless equations that are solved numerically. Similar unit system as in TRISTAN and TRISTAN-MP (Buneman et al. 1993; Spitkovsky 2005; Sironi et al. 2013) is used. Derivation and more thorough discussion of this unit system is given in the Appendix A.

Many quantities, like the electromagnetic fields, are defined on a lattice (mesh). Discrete location of a point in this case is given as \( x^{i,j,k} \equiv x(i,j,k) = (i\Delta x, j\Delta y, k\Delta z) \) where \( i, j, k \) are the grid indices and \( \Delta x, \Delta y, \) and \( \Delta z \) are the grid point distances in each Cartesian dimension. Similarly, discretized time is given as \( t^n \equiv t(n) = n\Delta t \). Cells of the lattice are taken to have a fixed cube geometry, \( \Delta x = \Delta y = \Delta z \). Courant (or Courant-Friedrichs-Lewy; Courant et al. 1928) number is defined as

\[
\hat{c} = c \frac{\Delta t}{\Delta x}.
\]

representing a maximum numerical velocity of a signal traveling on the grid. For explicit time integration schemes \( \hat{c} \leq 1 \).

Electromagnetic fields are normalized with \( B_0 \) as \( \hat{E} = E/E_0 \) and \( \hat{B} = B/B_0 \). Similarly, current density is normalized with \( J_0 \) as \( \hat{J} = J/J_0 \). The value of these normalization factors are selected such that the numerically solved equations appear as if \( \Delta x = \Delta t = 1 \). This also means that the grid indices, \( i, j, k \), and \( \Delta x \) have the same numerical value as location \( x \).

2.3.1. Electromagnetic field module

Time evolution of electromagnetic fields is handled with a finite-difference-time-domain (FDTD) method. Yee lattice (Yee 1966) is used for \( E \) and \( B \) fields such that they are staggered both in space and in time,

\[
\hat{E} = (\hat{E}_{x,i,j,k}, \hat{E}_{y,i,j,k}, \hat{E}_{z,i,j,k})^T \quad (12)
\]

\[
\hat{B} = (\hat{B}_{x,i,j,k}, \hat{B}_{y,i,j,k}, \hat{B}_{z,i,j,k})^T \quad (13)
\]

where \( \hat{E} \) is located in the middle of the cell sides and \( \hat{B} \) in the center of the cell faces. This makes it easy to calculate the curl of the fields in the subsequent equations.

In the time domain we update \( \hat{E} \) and \( \hat{B} \) fields with discrete forms of Eqs. (8) and (9) given as

\[
\Delta[\hat{E}]_k = \hat{c} \Delta[\hat{B}]_k - \mathbf{J},
\]

and

\[
\Delta[\hat{B}]_k = -\hat{c} \Delta[\hat{E}]_k.
\]

where \( \Delta[Q]_k \) is the time differentiation or curl of a variable \( Q \) without the \( \Delta t \) or \( \Delta x \) denominator. Only normalization factor entering these equations is the Courant velocity, \( \hat{c} \), since everything else is absorbed in \( B_0 \) and \( J_0 \). There is no need to solve Eqs. (6) and (7) if charge-conserving scheme is used together with the Yee staggering of the fields (see Appendix B for more in depth discussion).

2.3.2. Particle-in-Cell module

Main purpose of the PIC module is to update particle velocities according to the Lorentz force. We express the four-velocity \( u \) in units of \( c \). Particle species specifier \( s \) is omitted in this section for brevity.

Lorentz force acting on a charged particle is

\[
m \frac{du}{dt} = q \left( \frac{E + \mathbf{v} \times \mathbf{B}}{c} \right). \quad (16)
\]

We simplify this expression again to appear as if unitless. Let us express charge and mass as \( q = q_0 \hat{q} \) and \( m = m_0 \hat{m} \).

The right-hand side of the Lorentz force equation is then

\[
\Delta[\hat{u}]_k = \frac{\hat{q}}{\hat{m}} \left( \hat{E} + \frac{\hat{v}}{\hat{c}} \times \hat{B} \right) \quad (17)
\]

The actual velocity update is done in parts since \( \hat{E} \) and \( \hat{B} \) are staggered in time. As an example of particle momentum update, the relativistic Boris scheme (Boris 1970) is presented in detail in Appendix C. In addition to the particle’s velocity update routines, we need a scheme to interpolate the (staggered) \( \hat{E} \) and \( \hat{B} \) fields to the arbitrary particle locations. This interpolation is typically implemented using a linear volume-averaging interpolation scheme although higher-order schemes can also be utilized.

After the velocity update, we advance the particle’s location. Particle’s position \( x \) is advanced in time with

\[
\Delta[x]_k = \frac{u_k}{\gamma} \left[ \sqrt{1 - \gamma^{-2}} \right]. \quad (18)
\]

The described particle propagation scheme is second order in time: the velocity is stored in middle of the time steps and updated as \( u_{n+\frac{1}{2}} = u_{n-\frac{1}{2}} + \Delta[\hat{u}]_{k,n} \); the location is stored at integer time steps and updated as \( x_{n+1} = x_n + \Delta[x]_{k,n+\frac{1}{2}} \).

Finally, when a charged particle moves on the grid it creates a current \( \mathbf{J} \) that induces an electric field via Eq. (9). We use the charge-conserving ZigZag current deposition scheme to deposit the current from the computational particles in to the mesh (Umeda et al. 2003). This scheme is summarized in Appendix D.

3. Numerical implementation

Let us now discuss the general design principles and structure of the framework itself. RUNKO is a hybrid C++/PYTHON code: low-level, computationally intensive “kernels” are written in C++ whereas the high-level functionality is operated by PYTHON scripts. The design heavily relies on object-oriented programming paradigm (supported naturally by both C++ and PYTHON) where data containers (attributes) and functions (methods) are grouped together into objects that interact with each other. In RUNKO we call a group of these objects modules (Sect. 3.2) and the objects themselves as solvers (Sect. 3.3). The solvers are operated by different kinds of drivers (Sect. 3.4). This kind of hybrid usage of the C++ and PYTHON ensures fast and well-optimized low-level code, whereas the PYTHON scripts allow for ease of usage and straightforward extensibility of the code.
The low-level C++ kernels heavily rely on template metaprogramming features of modern C++ (conforming currently to C++14 standard\(^2\)). In template metaprogramming a code template is provided to the compiler that then generates the actual source code based on it, given some extra information on how to fill the template parameters\(^3\). Many of the implemented C++ kernels take advantage of this by only having a \(D\)-dimensional general template of the algorithm. This template of an algorithm is then specialized to the required spatial dimension (typically \(D = 1, 2, \) or \(3\)) by the compiler during compile time. This translates to a more bug-free code as typically only one implementation of the algorithm is required.

The low-level C++ kernels are operated by user-friendly PYTHON scripts. Technically, the C++ kernels are exposed to PYTHON using the PYBIND1 library (Jukob et al. 2017). All the exposed C++ objects behave as if they are native PYTHON objects. No unnecessary memory copies are performed because PYBIND1 manages memory pointers. After installing RUNK0 these objects are available to PYTHON by importing them from the pyrunko package (and pycorgi as described in Sect. 3.1). Not only can these classes be easily used from the PYTHON scripts but they also support inheritance, in case the user wants to extend and modify the objects. This allows for fast prototyping as new classes can be first developed in PYTHON before implementing them in C++ (in case a better performance is needed). These PYTHON scripts also leverage NUMPY (van der Walt et al. 2011) and SCIPY for performance and convenience, together with MATPLOTLIB (Hunter 2007) for visualization.

### 3.1 Grid infrastructure

RUNK0 is built on top of the massively-parallel CORGI\(^4\)C++ template library (Nattilä, in prep.). CORGI is a modern parallelized grid infrastructure library that provides the (spatial) simulation grid. It is designed to be run on distributed memory systems (i.e., computing clusters and supercomputers) that require explicit inter-node communication. All inter-node communications are handled by the corgi library using the MPI (Message Passing Interface) library\(^5\). From hereafter, one MPI process is called a rank, to make a distinction from a processor because a rank does not necessarily have to coincide with a processor.

CORGI uses a patch-based super-decomposition strategy to partition the simulation grid between different ranks. This means that the entire simulation grid is split into small (preferably continuous) sub-regions, called tiles. One tile can, for e.g., host a \(10\times10\) sub-grid in a 2D simulation. If the complete simulation domain is composed of, for example, \(10^3\times10^3\) lattice, it can be tesselated with \(10^3\times10^3\) of such tiles.

\(^2\)The C++14 and 17 standards are generally considered as “transition standards” bridging the gap between the older C++11 and the newer C++20 version. In the future, RUNK0 will transition to fully conform to the C++20 standard as soon as a sufficient HPC compiler support can be guaranteed. At the moment, many of the newer C++20 template metaprogramming features are provided together with RUNK0 itself, implemented using C++11.

\(^3\)We use the standard C++ syntax to present template classes: an object \(A\) with a template parameter \(X\) is given as \(A\langle X\rangle\).

\(^4\)https://github.com/natj/corgi

\(^5\)https://github.com/natj/mpi4cpp

Implementation of a physical simulation code works by inheriting all the grid infrastructure methods from a corgi::Tile\(<D>\) template class. This derived template class then automatically holds all the relevant physical attributes and methods needed to perform the simulation communications. The tile object provided by CORGI is a template class that is parameterized with the spatial dimension \(D\). Specializing this dimension parameter to any given value sets the simulation grid dimensionality.

All the tiles are stored in a container provided by CORGI called a grid. Each program rank has its own grid object corgi::Grid\(<D>\) (specialized for \(D\)) that holds a varying number of tiles. This allows to decompose the grid so that each rank will only hold and process a small part of the simulation domain (i.e., set of tiles). The grid class is responsible for providing all the spatial neighborhood information of the tiles and executing the inter-tile communications. These include both local shared-memory intra-rank communications and global inter-rank MPI communications. In practice, these methods are typically used to keep the halo regions of the tiles up-to-date.

There are no restrictions on the shape of the tile boundaries between different ranks. This allows using more complex tile ownership between ranks that aim to maximize the “volume” and minimize the “surface area” such as a honeycomb tesselation in 2D. Such configuration then translates directly to less communication needed between ranks.

Both the corgi::Tile and corgi::Grid C++ classes are exposed to PYTHON. Dimensionality, i.e., the \(D\) template parameter, is automatically set by loading a correct sub-package from the runk0 (and CORGI) python libraries: options are oneD for one-dimensional simulations \((N_x\text{ sized tiles}; N_x, \text{dimensional global grid configuration})\), twoD for two-dimensional simulations \((N_x\times N_y\text{ sized tiles}; N_x\times N_y, \text{dimensional global grid configuration})\), and threeD for three-dimensional \((N_x\times N_y\times N_z\text{ sized tiles}; N_x\times N_y\times N_z, \text{dimensional global grid configuration})\). Here \(N_{x,y,z}\) is the sub-grid resolution of tiles and \(N_{x,y,z}\) is the simulation grid resolution in units of the tiles.

### 3.2 Modules

The actual “physics” of the simulation is implemented in derived tiles that are based on the CORGI base tiles. We call these tiles and all the related objects and functions as modules\(^6\). An important part of the framework design is that different modules can be combined with polymorphic multiple inheritance, i.e., deriving methods and attributes from multiple other modules. This allows an efficient re-use of different modules when designing a new module with similar or overlapping physics concepts.

As an example, the PIC code (Sect. 2.3.2) uses the corgi::Tile as its first base class to inherit the grid infrastructure and fields::Tile as its second base class to inherit the electromagnetic FDTD routines (Sect. 2.3.1). The PIC module itself only defines the particle containers and methods to manipulate particle movement. The two modules are coupled together by the current \(j\) that is created by the moving charged particles. This current then induces a changing \(E\) and \(B\) fields that affect the Lorentz force experienced by the particles (see Fig. 1).

\(^6\)Technically a module corresponds to a namespace in C++ and a sub-package in PYTHON.
### Table 1. Time-advance loop of Particle-in-Cell module

| Description | Routine name | Module/Solver | Eqs. | Comments |
|-------------|--------------|---------------|------|----------|
| **Grid synchronization** | | | | |
| Communicate \( \hat{B} \) | mpi_b0 | CORGI | | global communication |
| Communicate \( \hat{E} \) | mpi_e0 | CORGI | | global communication |
| Update halo regions | upd_bc0 | FIELDS | | local communication |
| **First half of \( \hat{B} \) advance** | | | | |
| Advance \( \hat{B} \) | push_half_b1 | FIELDS/PROPAGATOR | \[15\] | 2nd order FDTD \( ^{†} \) |
| Communicate \( \hat{B} \) | mpi_b1 | CORGI | | global communication |
| Update \( \hat{B} \) halo regions | upd_bc1 | FIELDS | | local communication |
| **Particle advance** | | | | |
| Interpolate fields to particle locations | interp_em | PIC/INTERPOLATOR | Linear scheme \( ^{†} \) |
| Advance particle \( u \) and \( x \) | push | PIC/PUSHER | \(17\)–\(18\); Boris scheme: \( ^{†} \) \( C.5 \)–\( C.7 \) |
| **Second half of \( \hat{B} \) advance** | | | | |
| Advance \( \hat{B} \) | push_half_b2 | FIELDS/PROPAGATOR | \[15\] | 2nd order FDTD \( ^{†} \) |
| Communicate \( \hat{B} \) | mpi_b2 | CORGI | | global communication |
| Update \( \hat{B} \) halo regions | upd_bc2 | FIELDS | | local communication |
| **\( \hat{E} \) advance** | | | | |
| Advance \( \hat{E} \) | push_e | FIELDS/PROPAGATOR | \[14\] | 2nd order FDTD \( ^{†} \) |
| **Current \( J \) calculation** | | | | |
| Calculate \( J \) | comp_curr | PIC/DEPOSITER | ZigZag scheme \( ^{†} \) |
| Initialize \( J \) halo regions | clear_vir_cur | FIELDS | | |
| Communicate \( J \) | mpi_cur | CORGI | | global communication |
| Update \( J \) halo regions | cur_exchange | FIELDS | | local communication |
| **Particle communications** | | | | |
| Tag outflowing particles | check_outg_prtcls | PIC | | |
| Pack them into a message | pack_outg_prtcls | PIC | | |
| Communicate particles | mpi_prtcls | CORGI | | global communication |
| Unpack received messages | unpack_vir_prtcls | PIC | | |
| Receive incoming particles | get_inc_prtcls | PIC | | local communication |
| Remove outflowing particles | del_transfrc_prtcls | PIC | | |
| Clear global message arrays | del_vir_prtcls | PIC | | |
| **Deposit current \( J \) to \( \hat{E} \)** | | | | |
| Add \( J \) to \( \hat{E} \) | add_cur | FIELDS | \[14\] | |
| **I/O and analysis** | | | | |
| Write output files | io | io | | |

Notes: The listed routines are evaluated in order from top to down, with one loop corresponding to one time step. Changeable components and algorithms are marked with a \( ^{†} \) symbol.

### 3.3. Solvers

Each module can contain several solvers that operate on and/or change the internal attributes of the module’s tiles. Solvers are created by first defining an interface with an abstract class. The actual solver implementations that can be instantiated (i.e., the concrete classes) can then be implemented by creating a derived class from this abstract interface. These abstract interface classes are bound to PYTHON via the so-called trampoline classes. This automates the binding of any user-defined concrete solver implementations. A typical requirement of any solver is that it should provide a solve method that takes (at least) the module’s tile as an input.

As an example, the PIC module defines, e.g., an abstract INTERPOLATOR solver interface. The purpose of this solver object is to interpolate the \( \hat{E} \) and \( \hat{B} \) fields, defined on a lattice, to arbitrary particle locations. One of the currently provided INTERPOLATOR instances is the \( \text{LINEAR SOLVER} \) that implements a linear 1st order interpolation scheme to approximate the field values between the grid points. Thanks to this design pattern, it is very easy to extend the module to include other interpolation methods, like 2nd order polynomial or a spectral interpolation. The framework is agnostic about how the job of the solver is actually fulfilled: it is enough that it just conforms to the abstract interface class standard, and after that it can be automatically incorporated into the module routines.

### 3.4. Drivers

The actual simulation codes in RUNKO are executed by the so-called drivers. Drivers are short PYTHON scripts that use the C++ bindings of the tiles and solvers. They typically consist of an initialization section in the beginning where tiles of some specific module are loaded into the CORGI grid container. After this, the data attributes of the tiles are initialized. The drivers also include the main time prop-
Table 2. List of runko modules and solvers

| Module / Solver | Instance | Description | Eqs. | References |
|-----------------|----------|-------------|------|------------|
| FIELDS          | Maxwells field equations module | | | |
| PROPAGATOR      | Field solver | | | |
| FDTD2           | 2nd order FDTD solver | (14) and (15) | | Yee (1966) |
| FDTD4           | 4th order FDTD solver | | | Greenwood et al. (2004) |
| FDTDGen         | FDTD solver with free coefficients | | | Blinne et al. (2018) |
| FILTER          | Current smoothing | | | |
| Binomial2       | 3-point (Binomial) digital filter | | | Birdsall & Langdon (1985) |
| Compensator2    | 3-point “compensator” filter | | | ibid. |
| PIC             | Particle-in-Cell module | | | |
| Pusher          | Particle x and u propagation | | | |
| base class      | Simple velocity Verlet propagator | (15) | | Verlet (1967) |
| BorisPusher     | Relativistic Boris pusher | (C.3)-(C.7) | | Boris (1970) |
| VayPusher       | Vay pusher | | | Vay (2008) |
| HigueraCaryPusher | Higuera-Cary pusher | | | Higuera & Cary (2017) |
| rGCAPusher      | Guiding-center pusher | | | Bacchini et al. (2020) |
| INTERPOLATOR    | Field interpolator to particle location | | | |
| LinearInterpolator | Linear 1st order interpolator | | | |
| QuadraticInterpolator | Quadratic 2nd order interpolator | | | |
| CubicInterpolator | Cubic 3rd order interpolator | | | |
| QuarticInterpolator | Quartic 4th order interpolator | | | |
| DEPOSITER       | Current deposition | | | |
| ZigZag          | First order ZigZag scheme | (D.1)-(D.5) | | Umeda et al. (2003) |
| ZigZag_2nd      | 2nd order ZigZag scheme | | | Umeda et al. (2003) |
| Esikerpov_2nd   | 2nd order Esikerpov scheme | | | Esirkepov (2001) |
| Esikerpov_4th   | 4th order Esikerpov scheme | | | ibid. |

Fig. 1. Schematic presentation of the PIC code’s class relationship and connectivity. The FIELDS module is built on top of the CORGI module that provides the spatial simulation grid. It contains the electromagnetic fields, \( \hat{E} \) and \( \hat{B} \), and methods to advance them in time. The PIC module handles the computational particles. It is coupled to the FIELDS module via the current \( \hat{J} \) induced by the moving charged particles. Similarly, the FIELDS module is coupled to the PIC module by the Lorentz force.

The main advantage of using the low-level C++ classes via the PYTHON driver scripts is the ease of usage. Various complex initialization routines are, for example, rendered much easier to implement in a high-level language that also supports data visualization. It also opens up a possibility of performing (at least part of) the analysis on-the-fly during the simulation.

3.5. Currently implemented modules

Currently implemented modules include the FIELDS and PIC modules. These and the corresponding solvers are listed in Table 2.

The FIELDS module (Sect. 2.3.1) is responsible for advancing the electromagnetic fields via the Maxwell’s equations. This module inherits its spatial dimensionality \( D \) from the CORGI tiles. Internally it has a mesh container that stores the field components as \( D \) dimensional lattices. It also holds methods to propagate these fields in time using Eqs. (14) and (15). The module is agnostic about the form of the incoming current—it can be coupled to any other module to provide this closure.

The PIC module (Sect. 2.3.2) handles the particle propagation and provides the current closure for the FIELDS module (see also Fig. 1). Attributes of the electromagnetic field lattices are inherited from the fields::Tile. In addition to the spatial dimensionality template parameter \( D \), the PIC module has another template parameter for the velocity space dimensionality, \( V \), i.e., \( \text{pic::Tile<}D, V> \). By specializing the \( V \) template parameter, the solvers acting on the \( \text{pic::Tile} \) take into account the corresponding velocity space dimensionality. For example, setting \( V = 2 \) equals to manipulating the velocity vector as \( \mathbf{u} = (u_x, u_y) \) whereas \( V = 3 \) modifies a full three-dimensional velocity vector \( \mathbf{u} = (u_x, u_y, u_z) \). PIC module also defines various solvers that operate on the \( \text{pic::Tiles} \). See Table 2 for all the solvers implemented in the framework.
4. Problem showcase

In this section we showcase the usage of the new PIC code implemented into the runko framework. First, we demonstrate the code validity by comparing simulation results of relativistic Weibel instability to the known analytical growth rates. Next, we demonstrate the same instability on a larger-scale by presenting results from an unmagnetized collisionless shock simulation. Finally, we measure the numerical performance of the code by using results from relativistic turbulence simulations.

4.1. Relativistic Weibel instability

We start showcasing the code capabilities by first simulating a "standard" astrophysics-motivated plasma instability, commonly known as the Weibel instability (Weibel 1959, Fried 1959, Bret et al. 2010). Relativistic Weibel instability in a collisionless plasma is triggered when two counter-streaming plasma populations start to experience small charge-dependent deflections, seeded by the simulation noise. The resulting modes are electromagnetic so perturbations in both $E$ and $B$ are important. Therefore, the presented simulations also help us probe the validity of the solutions to the full Vlasov-Maxwell’s equations.

4.1.1. Problem setup

We simulate the linear growth phase of the instability by following two counter-streaming plasma populations in a 2D simulation domain. The plasma is modelled with 32 particles per cell per species and is initialized to have a small thermal spread of $\Delta T = kT/m_e c^2 = 10^{-3}$ and flowing relativistically along $\hat{x}$ as $u = \pm \beta \hat{x}$. Electromagnetic fields are propagated in time with the standard FDTD solver, current is deposited with the BorisPusher scheme, particle interpolation is done with the LinearInterpolator method, and particle propagation is performed with BorisPusher solver. The simulation domain is captured with a grid of $320 \times 80$ and the skin-depth is resolved with 10 cells. We use a standard time step of $\Delta t = 0.45 \Delta x$. We use symmetric beams with Lorentz factors of $\gamma_b = 3, 10, 30$, and 100.

4.1.2. Comparison to analytical estimates

The analytic linear growth rates of the Weibel instability are known for symmetric beams (Bret et al. 2010) presents the filamentation instability growth rate $\delta$ as

$$\frac{\delta}{\omega_p} = \beta_b \sqrt{\frac{2}{\gamma_b}}$$

where $\gamma_b = (1 - \beta_b^2)^{-1/2}$ is the bulk Lorentz factor of the beams, $\omega_p = \sqrt{4\pi n_e e^2 / m_e}$ is the plasma frequency, $n_e$ is the plasma number density (of electrons). We have compared the simulation results to these theoretical estimates in Fig. 2 and found the linear instability growth rate to be perfectly matched with the expectations.

These simulations are not particularly demanding and can therefore serve as a helpful introduction to the code usage. The discussed setup is also general enough that it could be easily reformulated to study other analytically intractable plasma beam instabilities.

4.2. Formation of collisionless shocks

The relativistic Weibel filamentation setup can be expanded to a realistic 3D simulation and used to model the plasma physics of collisionless shocks. The shock is formed by the growth and nonlinear saturation of the same instability.

We have found this setup to be one of the most demanding numerical tests for PIC codes because the numerical algorithms need to be able to both sustain the relativistic upstream bulk flow and correctly model the kinetic shock physics. The upstream is unstable to numerical Cherenkov instability (Godfrey 1974) and is, in addition, sensitive to any numerical error in the current deposition; even tiny perturbations in the relativistically advected upstream plasma can trigger an isotropization of the bulk flow. The shock front, on the other hand, is sustained by a strong electric current that is sensitive to charge conservation. We use this setup to stress test the full 3D PIC routines and their correctness.

4.2.1. Problem setup

The numerical setup consists of an elongated shock "tube" along the $\hat{x}$-axis and a reflecting wall at the left side of the domain. Initially, we set all the plasma in the domain...
to move from right to left with a velocity \( \mathbf{u} = -\Gamma \hat{x} \). The plasma is reflected from the left simulation domain and forms a counter-propagating beam. This causes a formation of a collisionless shock.

We simulate the shock in an elongated box with a size \( L_x \times L_z = 512 \times 64^3 (c/\omega_p)^3 \) using 10 cells per skin depth so that the full domain is covered with a grid of \( 5120 \times 64^2 \). The plasma is flowing with a relativistic velocity \( \Gamma = 10 \) and is modeled with 64 particles per cell per species. Electromagnetic fields are propagated in time with the 4th-order FDTD4 scheme. The resulting current is calculated using the ZigZag scheme, particle interpolation is done via the LinearInterpolator, and the current is smoothed with 8 passes of binomial filter (Binomial12). We use a time step of \( \Delta t = 0.45 \Delta x \). The pair plasma has an initial thermal spread of \( \theta = kT/m_e c^2 \approx 2 \times 10^{-5} \) and is updated using the Hi guar-aCar y Pusher. We start the simulation with no background magnetic field corresponding to an unmagnetized shock.

The plasma is injected into the domain from a receding right wall that moves with a speed of light, from left to the right; this enables longer simulations both because the particles are not "alive" as long as they normally would (suppressing numerical Cherenkov instability) and by saving computational time (since we do not need to initially propagate all the particles in the domain). The shock front is adaptively tracked on-the-fly and only the high-resolution grid data around it is saved. The grid is decomposed into cyclic MPI domains of length 1280 cells in the \( \hat{x} \) direction (i.e. 5 cycles per the full box); this results in a more even memory load for the simulation.

### 4.2.2. Physical interpretation and numerical tests

A collisionless shock is quickly formed when the simulation is started. The shock is mediated by a formation of Weibel filaments that grow, saturate, and merge; this stops the counter-streaming plasma (see Fig. 3).

The simulated shock structure and dynamics are well-described by the theoretical expectations and previous numerical experiments (e.g., Sironi & Spitkovsky 2009, Plotnikov et al. 2018). The filaments form and induce the shock in about \( 100 \omega_p^{-1} \). The upstream flow is compressed into a more dense downstream with a compression ratio \( n_{\text{down}}/n_{\text{up}} \approx 3.65 \pm 0.15 \) and the shock propagates into the upstream with a mildly relativistic velocity, \( \beta_{\text{sh}} \approx 0.36 \pm 0.02 \) (measured for \( 200 \leq \omega_p \leq 1000 \)). Unmagnetized MHD jump conditions, in comparison, predict, \( n_{\text{down}}/n_{\text{up}} = 1 + (\Gamma + 1)/(\Gamma \gamma_{\text{AD}} - 1) \approx 4.3 \) for the compression ratio, and \( \beta_{\text{sh}} = (\gamma_{\text{AD}} - 1)/(\Gamma - 1) \approx 0.3 \), given the 3D adiabatic index \( \gamma_{\text{AD}} = 4/3 \) (see e.g., Plotnikov et al. 2018). The shock ramp itself has a length of roughly \( 50 c/\omega_p \) and is accommodated by an increase in the electric field energy density just ahead of the moving front.

We have also specifically tested that the upstream remains smooth and stable for an extended period of time, \( \geq 1000 \omega_p^{-1} \) with electric and magnetic field fluctuations of \( \delta E, \delta B \lesssim 10^{-6} \) (in absolute code units; the value corresponds to floating point accuracy). This is a stringent test that can be used to verify that the particle-to-grid and grid-to-particle routines are all correctly implemented, and no charge or electric current is lost; in the opposite case any numerical error will quickly grow and isotropize the flow. Similarly, the formation of the shock front supports the no-
tion that all the 3D routines are correct. We emphasize that the shock simulation is a particularly demanding code test because of the above details.

4.3. Decaying relativistic kinetic turbulence

In this section we use a so-called relativistic decaying turbulence setup to showcase the code performance and scalability metrics. The setup is presented and discussed in detail in Comisso & Sironi (2018); Comisso & Sironi (2019); Nättiä & Beloborodov (2020). Similar driven turbulence simulations are performed for e.g., in Zhdankin et al. (2017).

4.3.1. Problem setup

Consider a uniform pair plasma that is suddenly stirred. Due to this large-scale driving motion, a turbulent cascade quickly develops. We study so-called 2D3V setups where the cascade forms in the planar x − y direction and is captured by the spatial 2D simulation domain. All three velocity components are, however, considered, including the out-of-the-plane z direction. To physically mimic confinement of the cascade to a 2D plane, a guide field is imposed in the z direction as \( B_0 = B_0 \hat{z} \) (\( \hat{z} \) is the unit vector along z axis).

Presence of the strong guide-field renders the plasma magnetically dominated. The plasma magnetization parameter we use is

\[
\sigma = \frac{B_0^2}{4\pi n_0 m_e c^2},
\]

where \( n_0 \) is the initial number density and \( \bar{y} \) is the mean thermal Lorentz factor of the plasma. Here we focus on a strongly magnetized case of \( \sigma = 10 \), only. Initially the plasma is defined to be warm with a temperature of \( \theta = kT/m_e c^2 = 0.3 \) (\( \bar{y} = 1.6 \)).

Spatial resolution of our fiducial simulation is 5120\(^2\) cells (512 × 512 tiles with internal tile resolution of 10 × 10). Initial plasma skindepth, \( c/\omega_p \), is resolved with 5 cells. This determines our box size as \( L = 1000 c/\omega_p \). We use 64 particles per cell per species totaling to 128 particles per cell for both electron and positron species. In total, we then propagate \( \sim 10^5 \) particles in the simulation.

The sudden large-scale stirring is mimicked by perturbing the planar magnetic field \( B_L \) with strong \( (B_0^2) \equiv \delta B^2 \sim B_0^2 \) uncorrelated fluctuations in x and y directions following Comisso & Sironi (2018). This drives strong turbulence with semi-relativistic bulk velocities. The forcing modes, \( m, n \in \{1, \ldots, 8\} \), are initialized with equal power as

\[
\delta B_x = \sum_{m,n} \beta_{mn} n \sin(k_m x + \phi_{mn}) \cos(k_n y + \psi_{mn}),
\]

\[
\delta B_y = -\sum_{m,n} \beta_{mn} n \cos(k_m x + \phi_{mn}) \sin(k_n y + \psi_{mn}),
\]

where \( k_m = 2\pi m/L \) and \( k_n = 2\pi n/L \) are the wave numbers along x and y dimensions, respectively. The forcing is uncorrelated so \( \phi_{mn}, \psi_{mn} \in [0, 2\pi] \), correspond to random phases. The forcing amplitude is

\[
\beta_{mn} = 0.002 \frac{N B}{N m^2 + r^2},
\]

where \( N = 8 \) in our case. The largest forcing mode \( (k_N = 2\pi N/L) \) defines the energy-carrying scale and the characteristic size of the largest magnetic eddies, \( \ell_0 = 2\pi/k_N \).

4.3.2. Physical interpretation

The turbulence cascade is quickly formed as the plasma relaxes from the initially non-equilibrium state (see Fig. 4 bottom row). The cascade is characterized by magnetic islands (would-be magnetic flux tubes in 3D) that collide and merge. This forms a self-similar chain of islands to smaller and smaller scales. In between the islands, thin current sheets are formed. When these current sheets become thin enough, they undergo magnetic reconnection as they tear. This produces small plasmoids that are expelled from their initial location in the sheets in to the surrounding plasma.

The resulting particle spectra has a thermal part and a power-law tail of non-thermal particles. Since a small particle Lorentz factor corresponds to a small Larmor gyration radius \( (\gamma \sim \sigma) \), the thermal population can depict a fluid-like behavior at scales larger than the characteristic Larmor scale. As a manifestation of this fluid-like behavior, a large number of these thermal particles is shown to organize into coherent flows that create random turbulent bulk flows in the plasma. These bulk motions then support the turbulent cascade.

The turbulent plasma shows copious numbers of current sheets. Particles in these current sheets are shown to experience a strong injection (up to \( \gamma \sim \sigma \)) that leads to an increase in the particle’s velocity and Larmor radius (Comisso & Sironi 2018; Comisso & Sironi 2019). This causes these injected particle to decouple from the thermal pool. After decoupling, these non-thermal particles interact with the turbulence only in a random stochastic manner. Therefore, the non-thermal particle population acts as an efficient energy sink for the kinetic turbulence. This is an interesting first-principles mechanism for providing energy dissipation to collisionless plasmas; it can also give rise to anomalous viscosity and resistivity that is strongly localized into the intermittent current sheet structures (see also e.g., Groselj et al. 2017; Cerri et al. 2019).

4.3.3. Numerical efficiency and parallel scaling

We use the turbulence simulation setup to probe the numerical performance and parallel scaling of the code. This setup is advantageous not only because it is a real physical problem but also because we can probe both load-balanced and load-imbalanced performance of the PIC code. Initially the simulations are in the load-balanced state (computing time being dominated by particle push time) but after the cascade forms we observe a strong numerical imbalance between different ranks because of the strong fluctuations in particle number densities (computing time dominated by communication routines). No dynamic load-balancing algorithm is used. Here we mainly focus on these transition scales of around \( \sim 10^5 \) processors where both of these regimes can be simulated, leaving extremely large simulations with \( \geq 10^7 \) processors and dynamic load balancing for future work. The scaling measurement reported here are performed in the Kebnekaise computing cluster. We use the second-order FDTD solver (FDTD2), Boris pusher (BorisPusher), linear interpolation (LinearInterpolator), and first-order ZigZag scheme.
Fig. 4. Visualizations of the kinetic turbulence simulations. Top row shows the honey-comb-like domain decomposition that partitions the grid between different processors with the corgi library. Bottom row shows the plasma density (in units of the initial plasma mass density, $\rho_0$). Left panels show the full simulation domain whereas the right panels focuses on a smaller sub-region that better illustrate the small-scale structures.

(ZigZag) to perform the simulations. Calculations are done using the double floating point precision.

Good numerical performance is obtained for the PIC code. In an ideal completely load-balanced state the average total particle update time per processor is of around 0.7 $\mu$s (see Fig. 5 around $n \lesssim 200$). In the strongly imbalanced simulation state (dominated mainly by inter-rank particle communications) the particle update time per processor is of around 1.1 $\mu$s when the turbulent cascade has developed (Fig. 5 for $n \gtrsim 4000$). The decrease in speed is caused by increase in the evaluation time of routines communicating the fields and particles to the neighboring ranks.

We also monitor all the separate routines (see Table 1) independently to probe the numerical performance of the code in more detail. Note, however, that this monitoring is only done for one particular rank so the component-wise evaluation times reported here might fluctuate in between different ranks experiencing differing loads. The numbers are, however, indicative of where most of the evaluation time is spent. As seen from Fig. 5 in the ideal case most of the evaluation time (around 0.3 $\mu$s per particle per processor) is spent in the interpolation routine (interp_em: pink solid line) where $\hat{B}$ and $\hat{E}$ are computed at the particle locations in-between the grid points. The large cost of interpolation step is caused by frequent cache misses: interpolation routine consists of many random access operations of the field arrays because particles are not sorted by their location; this prevents the processor cache prefetching from correctly predicting the incoming requested values. The second most expensive routine is the current calculation (comp_curr; green dashed line) with a typical evaluation time of $0.06 \mu$s per particle per processor. It has a similar problem with the cache memory because writing the current to the array is preformed with an unpredictable access pattern. When communication costs increase, the MPI messaging routines start to be as costly as the current de-
position with an average time of about 0.1 µs per particle per processor.

The code shows good weak (Fig. 6) and strong scaling (Fig. 7) when the simulations transform from cpu-bound to communication-bound. Weak scaling proceeds as 1.0 : 1.10 : 1.40 : 1.93 : 2.09 for every quadrupling of number of processors, 16 : 64 : 256 : 1024 : 4096. Strong scaling proceeds as 1.0 : 1.18 : 1.48 : 2.36 : 1.93 against an ideal scaling, for every doubling of number of processors for 256 : 512 : 1024 : 2048 : 4096. The increase in the evaluation time when the number of processors is increased can indeed be attributed to the increase in the evaluation time of the communication routines (field updates: mpi_b1, mpi_b2, mpi_e1, mpi_cur, clear_vir_cur; and particle update: mpi_prtcls, unpack_vir_prtcls, del_vir_prtcls). A similar test performed in the load-balanced state yields almost ideal scaling up to ~10^3 processors (scaling of roughly 1.10 of the ideal) with a total particle update time of about 0.7 µs.

Finally, we note that the extra costs originating from the intra-rank updates of the tile boundaries are negligible. This extra cost is introduced because of the patch-based grid infrastructure where the smallest unit of calculation is one tile (10 × 10 lattice with about 10^5 particles per species per tile in this particular case). Therefore, some communication is always needed to keep these individual units of calculation in sync (field update routines: upd_bc0, upd_bc1, cur_exchange; and particle update routines: check_outg_prtcls, pack_outg_prtcls, get_inc_prtcls, pack_outg_prtcls, del_transf_prtcls) even in systems completely relying on shared-memory parallelism. These routines are, however, typically 10–100x cheaper than the costs of the field interpolation or MPI communication tasks.

5. Discussion

5.1. Computational advantages

RUNKO is a modern numerical toolbox tailored for astrophysical plasma simulations. The framework is designed using recent computer science methods that rely on multiple levels of code abstraction. These in turn, help to create a general and easily extensible code framework. Furthermore, in order to encourage all kinds of usage, RUNKO is provided as an open source software for the community.

The code is written as a hybrid C++/PYTHON program. The low-level C++ kernels are currently designed to conform to a C++14 standard taking a benefit of plethora of modern computational methods. We, for example, heavily
rly on recent C++ template metaprogramming features to simplify the design of the code. This ensures a truly modern numerical code that will not be outdated immediately upon publication.

In addition to the underlying numerically efficient C++ implementation, all the classes have a PYTHON interface. This allows the user to take full advantage of all the benefits of using a high-level language without sacrificing the actual run-time speed of the code. This way, many of simulation setups can be quickly prototyped in a laptop and then scaled up to supercomputer platforms for the actual production runs. Furthermore, it allows designing and implementing very complex initial conditions for simulations because the initial routines can be created using PYTHON.

The modular design allows RUNKO to function as a fully agnostic framework for various different modeling formulations of simulating plasma with computers. The “physics” is implemented via the so-called modules that, in practice, often solve some partial differential equation or propagate Lagrangian particles species. This means that we are not locked into a one predefined theoretical formalism of describing the evolution of the plasma. The high degree of modularity also allows the user to easily change the numerical solvers and algorithms depending on the problem at hand.

The implementation relies on modern C++ features such as polymorphism and template metaprogramming to simplify the code and make it easily extensible. Different solvers inherit their standard interface from an abstract base class implementation so that introduction and implementation of new physical algorithms is straightforward. User-definable dimensionality of different solvers is handled by template specializations. Additionally, most of the solvers are agnostic to the actual floating-point accuracy (i.e., single or double precision) because the working precision is typically also specified as a template variable.

Another technical advantage of the framework is the patch-based domain decomposition strategy that is built-in to the grid infrastructure routines. As shown by the performance analysis, the cost of the additional updates needed to keep the individual tiles in sync are negligible in comparison to the evaluation costs of the “physics” in modules. In fact, having a larger number of moderately-sized tiles is found to be faster than partitioning the domain with fewer but larger tiles. The benefits of this system are that

- it automatically introduces some level of load-balancing into the simulations (each rank has \( \sim 10^6 \) tiles so load imbalances at the level of one tile are effectively smoothed out);
- it helps in data cache locality (array sizes and particle containers remain small and easily movable in memory), and
- it simplifies the global communication routines (since these communication routines are anyway needed).  

As a one particular test we have partitioned a 3D PIC shock simulation grid of 7680 \( \times \) 120\(^2\) with tiles of sizes 8\(^3\), 15\(^3\), or 30\(^3\) (keeping the number of processors fixed at 1280 cores) and found that the absolute wall-clock time per lap varies as 1.34, 0.66, and 1.22s. Smaller but more numerous tiles are found to have more expensive communication routines whereas larger tiles have worse solver performance. Optimal tile size is hence, in this particular case, somewhere around tiles of size 15\(^3\). Another benefit is the possibility to optimize the MPI domain boundaries with more complex edge shapes. Interesting example of this is the honey-comb-like tile memory configuration (see Fig. 1, top row). We also note that a similar decomposition technique seems to be the current state-of-the-art technical choice of many other recent high-performance PIC codes too, such as VPIC (Bowers et al. 2008) and SMILEI (Deronillat et al. 2018).

5.2. Future directions

The presented framework offers a unique possibility to start exploring and experimenting with multi-physics simulations. These are new kinds of numerical simulations that evolve multiple physical formalism simultaneously or in an adaptive-fashion selecting the most accurate (or relevant) solver on-the-fly.

These heterogeneous simulations will enable us to simulate larger—and therefore more realistic—systems. In the future, this can enable, for example, novel plasma simulations where the majority of the simulation domain is modeled with a force-free electrodynamic description of the plasma (Komissarov 2002). However, in some small regions with extreme conditions (e.g., current sheets, shocks, etc.) the simulation can be adaptively refined (on-the-fly) to the fully-kinetic description that enables a realistic in-situ modeling of particle acceleration. Another possibility could be to use a multiple-domain-multiple-physics approach where some fixed part of the simulation is, e.g., described with kinetic description whereas some other part is modeled with MHD. Here the domains could be divided based on, for example, some strong density gradient in the problem setup like those found in systems with diluted non-thermal particles on top of a more dense fluid-like plasma; physically these can model astrophysical systems like corona on top of stars and accretion disks.

Another important focus is the code performance and scalability. In practice, this mostly means minimizing the global communication costs. One possibility of decreasing the communication costs is the usage of more complex hybrid parallelization schemes. As an example, the usage of the patch-based domain super-decomposition encourages the use of a task-based hybrid parallelization scheme similar to e.g., DISPATCH framework (Nordlund et al. 2018). Since most of the updates of the tiles are numerically independent, these operations can be easily performed in parallel with shared-memory parallelization strategies. This, in turn, allows to increase the number of tiles per MPI rank which acts as an extra buffer against sudden load-imbalance fluctuations. Such a strategy also allows performing interleaved global (non-blocking) communications and normal solver updates simultaneously. This allows hiding almost all the (MPI) communications since we can prioritize the evaluation of tiles such that the boundary tiles are always computed first and then sent to the neighbors, whereas the calculation of the inner tiles are continued independently by other threads while waiting for the halo tiles. We are also currently experimenting with dynamical load balancing methods where the ownership of the tiles is changing during the simulation, depending on the computational load.

Finally, since PYTHON is becoming the new lingua franca of astronomy and astrophysics, it is important to stress that problem initialization in RUNKO can be done purely via PYTHON scripts. Not only does this mean that
new users are immediately familiar with the rough usage of the code but also that initialization of very complex simulation setups can be done by a high-level language. We hope that this will accelerate the usage of plasma simulations to study many yet unresolved astrophysical problems.

6. Summary

We started by reviewing the kinetic plasma theory and the numerically solved Vlasov-Maxwell system of equations (Sect. 2). The numerical implementation of the framework is discussed in Sect. 3. We focused, in particular, on the design of the code and introduced the different numerical components of the framework. Firstly, we discussed the CORGI grid infrastructure and the related patch-based domain super decomposition strategy where the computational grid is partitioned into separate tiles (Sect. 3.1). Secondly, we presented the physical building blocks of the code, modules (Sect. 3.2). Different physical formulations of modeling the evolution of the plasma and e.g., electromagnetic fields are encapsulated in different modules. Thirdly, each module can contain several numerical algorithms called solvers that are short C++ classes that operate and modify the content of different tiles (Sect. 3.3). Lastly, these C++ solvers are applied to the tiles and operated by PYTHON scripts, so-called drivers, ensuring easy of use (Sect. 3.4).

As our first task, we have implemented a new particle-in-cell module into the framework. The implementation is discussed in detail in Sects. 2.3.1 and 2.3.2. The PIC code demonstrates good numerical performance with a mean particle push time per processor of about 0.7 µs in an ideal load-balanced state and 1.1 µs in a strongly load-imbalanced state. Furthermore, the code is shown to scale well up to ~10 000 processors.

We showcased the usage of this PIC code by three different, complex, astrophysically-motivated plasma simulations. First, we simulated the linear growth and nonlinear saturation of the Weibel instability in a relativistically streaming pair plasma. Second, we performed a mock simulation of a current-filamentation-mediated relativistic collisionless shock in full 3D. Last, we tested the code performance by modeling a decaying relativistic kinetic turbulence in magnetized pair plasma. These simulations demonstrated the ease of use, extendability, and flexibility of the code. All setups are commonly used to study astrophysical phenomena and can therefore serve as a useful starting point for new users.

Acknowledgments

JN would like to thank Andrei Beloborodov, Axel Brandenburg, Luca Comisso, Troels Haugbølle, Druba Mitra, Ake Nordlund, Lorenzo Sironi, Anatoly Spitkovsky, and Alexander Veledina for stimulating discussions on numerics and plasma simulations. We would also like to thank the referee for many helpful comments and suggestions. The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC and HPC2N.

References

Bacchini, F., Ripperda, B., Philippov, A. A., & Parfrey, K. 2020, The Astrophysical Journal Supplement Series, 251, 10
Bret, A., Grenillet, L., & Dieckmann, M. E. 2010, Physics of Plasmas, 17, 120501
Birdsall, C. & Langdon, A. 1985, Plasma physics via computer simulation, The Adam Hilger series on plasma physics (McGraw-Hill, New York)
Blinne, A., Schinkel, D., Kuschel, S., et al. 2018, Computer Physics Communications, 224, 730
Boris, J. P. 1970, in Proc. Fourth Conf. Num. Sim. Plasmas, Naval Res. Lab, Wash, DC, 3–67
Bowers, K. J., Albright, B. J., Yin, L., Bergen, B., & Kwan, T. J. T. 2008, Physics of Plasmas, 15, 055703
Brandenburg, A. 2003, in Advances in Nonlinear Dynamos (CRC Press), 269–344
Buneman, O. 1959, Physical Review Letters, 115, 503
Buneman, O., Neubert, T., & Nishikawa, K. I. 1992, IEEE Transactions on Plasma Science, 20, 810
Buneman, O., Nishikawa, K.-I., & Neubert, T. 1993, in Plasma Physics and Controlled Nuclear Fusion (ITC-4), ed. H. T. D. Guyenne & J. J. Hunt
Ceri, S. S., Groselj, D., & Franci, L. 2019, Frontiers in Astronomy and Space Sciences, 6
Cerutti, B., Philippov, A., Parfrey, K., & Spitkovsky, A. 2015, Monthly Notices of the Royal Astronomical Society, 448, 606
Cerutti, B., Werner, G. R., Udansky, D. A., & Begelman, M. C. 2012, The Astrophysical Journal, 754, L33
Chapman, S. & Cowling, T. G. 1970, The Mathematical Theory of Non-uniform Gases (Cambridge University Press, Cambridge)
Chen, A. Y. & Beloborodov, A. M. 2014, The Astrophysical Journal, 795, L22
Cheng, C. Z. & Knorr, G. 1976, Journal of Computational Physics, 22, 330
Choudhuri, A. R. 1998, The Physics of Fluids and Plasma: An Introduction for Astrophysicists (Cambridge University Press, Cambridge)
Comisso, L. & Sironi, L. 2018, Physical Review Letters, 121
Comisso, L. & Sironi, L. 2019, ApJ, 886, 122
Courant, R., Friedrichs, K., & Lewy, H. 1928, Mathematische Annalen, 100, 32
Dawson, J. 1962, Physics of Fluids, 5, 445
Dawson, J. 1966, Physics of Fluids, 7, 419
Deroiullat, J., Beck, A., Pérez, F., et al. 2018, Computer Physics Communications, 222, 351
Esinkepov, T. Z. 2001, Computer Physics Communications, 135, 144
Frederiksen, J. T., Hededal, C. B., Haugbølle, T., & Nordlund, Å. 2004, The Astrophysical Journal, 608, L13
Fried, B. D. 1959, Physics of Fluids, 2, 337
Goodfyl, B. B. 1974, Journal of Computational Physics, 15, 504
Groselj, D., Cerri, S. S., Navarro, A. B., et al. 2017, The Astrophysical Journal, 847, 28
Greenwood, A. D., Cartwright, K. L., Luginsland, J., Baca, E. A. 2004, Journal of Computational Physics, 201, 665
Haggerty, C. C. & Caprioli, D. 2019, The Astrophysical Journal, 887, 167
Higuera, A. V. & Cary, J. R. 2017, Physics of Plasmas, 24, 052104
Hockney, R. W. & Eastwood, J. W. 1981, Computer Simulation Using Particles (McGraw-Hill, New York)
Howes, G. G., Cowley, S. C., Dorland, W., et al. 2006, The Astrophysical Journal, 651, 590
Howes, G. G., Dorland, W., Cowley, S. C., et al. 2008, Physical Review Letters, 100
Hunter, J. D. 2007, Computing In Science & Engineering, 9, 90
Jackson, J. D. 1975, Classical electrodynamics (Wiley, New York)
Kagan, D., Milosavljević, M., & Spitkovsky, A. 2013, The Astrophysical Journal, 774, 41
Komissarov, S. S., 2002, MNRAS, 336, 759
Kumar, P. & Narayan, R. 2009, MNRAS, 395, 472
Langdon, A. B., Arons, J., & Max, C. E. 1988, Physical Review Letters, 61, 779
MacDonald, N. R. & Marscher, A. P. 2018, ApJ, 862, 58
Nordlund, Å., Ramsey, J. P., Popovas, A., & Küffmeier, M. 2018, Monthly Notices of the Royal Astronomical Society, 477, 624
Philippov, A. A. & Spitkovsky, A. 2014, The Astrophysical Journal, 785, L33
Philippov, A. A. & Spitkovsky, A. 2014, The Astrophysical Journal, 785, L33
Philippov, A. A. & Spitkovsky, A. 2014, The Astrophysical Journal, 785, L33
Plotnikov, I., Grassi, A., & Grech, M. 2018, Monthly Notices of the Royal Astronomical Society, 477, 5238
Servidio, S., Valentini, F., Perrone, D., et al. 2014, Journal of Plasma Physics, 81
Sironi, L. & Spitkovsky, A. 2009, ApJ, 698, 1523
Sironi, L. & Spitkovsky, A. 2014, ApJ, 783, L21
Sironi, L., Spitkovsky, A., & Arons, J. 2013, The Astrophysical Journal, 771, 54
Spitkovsky, A. 2005, in American Institute of Physics Conference Series, Vol. 801, Astrophysical Sources of High Energy Particles and Radiation, ed. T. Bulik, B. Rudak, & G. Madejski, 345–350
Tanaka, M. 1983, Physics of Fluids, 26, 1697
Umeda, T., Omura, Y., Tominaga, T., & Matsumoto, H. 2003, Computer Physics Communications, 156, 73
Umeda, Y., Omura, T. T. H. M. 2005, ISSS-7, 26
Valentini, F., Trávníček, P., Califano, F., Hellinger, P., & Mangeney, A. 2007, Journal of Computational Physics, 225, 753
van der Walt, S., Colbert, S. C., & Varoquaux, G. 2011, Computing in Science Engineering, 13, 22
Vay, J.-L. 2008, Physics of Plasmas, 15, 056701
Verlet, L. 1967, Physical Review, 159, 98
Weibel, E. S. 1959, Physical Review Letters, 2, 83
Werner, G. R., Uzdensky, D. A., Cerutti, B., Nalewajko, K., & Begelman, M. C. 2015, The Astrophysical Journal, 816, L8
Winske, D., Yin, L., Omidi, N., & al., e. 2003, Hybrid Simulation Codes: Past, Present and Future - A Tutorial, ed. J. Büchner, C. Dum, & M. Scholer, Vol. 615, 136–165
Yee, K. 1966, IEEE Transactions on Antennas and Propagation, 14, 302
Yuan, F. & Narayan, R. 2014, ARA&A, 52, 529
Zenitani, S. & Hoshino, M. 2001, The Astrophysical Journal, 562, L63
Zhdankin, V., Uzdensky, D. A., Werner, G. R., & Begelman, M. C. 2017a, Monthly Notices of the Royal Astronomical Society, 474, 2514
Zhdankin, V., Werner, G. R., Uzdensky, D. A., & Begelman, M. C. 2017b, Physical Review Letters, 118
Appendix A: Non-dimensionalization of the Vlasov-Maxwell equations

Following Jackson (1975) the Maxwell’s equations in an arbitrary system of units can be written as

\[ \nabla \cdot E = 4 \pi k_1 \rho \]  
(A.1)

\[ \frac{\partial E}{\partial t} = c^2 \nabla \times B - 4 \pi k_1 J \]  
(A.2)

\[ \nabla \cdot B = 0 \]  
(A.3)

\[ \frac{\partial B}{\partial t} = -k_2 \nabla \times E, \]  
(A.4)

where \( k_1 \) and \( k_2 \) are constants that define the unit system. For Gaussian system we have \( k_1 = 1 \) and \( k_2 = c \) whereas for rationalized MKSA \( k_1 = 1/4 \pi \epsilon_0 \) and \( k_2 = 1 \). Here \( \epsilon_0 \) is the vacuum permittivity (and \( \mu_0 \) is the vacuum permeability found from relation \( c^2 = 1/(\epsilon_0 \mu_0) \)).

Here we will present the unit system in use (originally by Buneman et al., 1993 and described in detail also in TRISTAN-MP user-manual by A. Spitkovsky). We select the Gaussian system here by setting \( k_1 = 1 \) and \( k_2 = c \). Our Maxwell’s equations are then simplified to

\[ \nabla \cdot E = 4 \pi \rho \]  
(A.5)

\[ \frac{\partial E}{\partial t} = c \nabla \times B - 4 \pi J \]  
(A.6)

\[ \nabla \cdot B = 0 \]  
(A.7)

\[ \frac{\partial B}{\partial t} = -c \nabla \times E, \]  
(A.8)

and the fields appear symmetrical as if \( E \leftrightarrow B \). Lorentz force in this unit system is

\[ \frac{\partial \tilde{u}}{\partial t} = \frac{q}{m} \left( \tilde{E} \left/ \hat{c} \right. \times \hat{B} \right), \]  
(A.9)

Let us next normalize all the variables with some fiducial values. Most peculiar of these is our selection of distance and time scalings: we express the distance in units of the grid spacing, \( x = \hat{x} \Delta x \), and time in units of the time step, \( t = \hat{t} \Delta t \). Coordinate-velocity is then \( \hat{v} = \hat{v} \Delta x / \Delta t = \hat{v} c / \hat{c} \) and four-velocity is \( \hat{u} = \hat{u} c / \hat{c} = \hat{u} \gamma c / \hat{c} \). The fields are scaled with some fiducial field \( B_0 \) such that \( E = \hat{E}B_0 \) and \( B = \hat{B}B_0 \). Similarly, the charge, mass, and current density are also presented such that \( q = \hat{q}q_0, m = \hat{m}m_0 \), and \( J = J_0 \hat{J} \), respectively. This way, all of our numerical quantities are denoted with a hat.

In numerical (code) units the equations that are being solved are then

\[ \Delta [\tilde{E}]_x = \frac{\hat{c}}{c} \Delta x [\tilde{B}]_x - \frac{4 \pi J_0}{B_0} \Delta \hat{J} = \hat{c} [\Delta \tilde{B}]_x - \hat{J} \]  
(A.10)

\[ \Delta [\tilde{B}]_x = -\frac{c}{\hat{c}} \Delta \tilde{E} - \hat{c} [\Delta \tilde{J}]_x \]  
(A.11)

\[ \Delta [\tilde{u}]_x = \frac{\hat{q} q_0 \hat{c}}{\hat{m} \hat{c} \hat{c}} B_0 \Delta t \left( \tilde{E} + \hat{v} \hat{c} c \hat{c} \times \hat{B} \right) = \frac{\hat{q}}{\hat{m}} \left( \tilde{E} + \hat{v} \hat{c} c \hat{c} \times \hat{B} \right), \]  
(A.12)

where in the last steps we have defined

\[ B_0 = \frac{m_0 c}{\hat{q} q_0 \hat{c} \hat{c} \hat{c}}, \]  
(A.13)

\[ J_0 = \frac{1}{4 \pi \hat{c} \hat{c} \hat{c}}, \]  
(A.14)

so that Eqs. \( \text{A.10}-\text{A.12} \) appear as if unitless.

From Eqs. \( \text{A.13} \) and \( \text{A.14} \) we obtain a connection between the grid spacing, particle mass, and charge. Note first that current density is

\[ J = J_0 \hat{J} = \hat{q} q_0 \hat{c} \hat{c} \Delta x \Delta \hat{v} = \hat{q} \hat{v} - \frac{q_0}{\hat{c} \hat{c} \hat{c}} \Delta \hat{x} \hat{v}, \]  
(A.15)

i.e., \( J_0 = q_0 / \Delta \hat{x} \Delta \hat{v} \Delta t \). This implies that

\[ \frac{q_0}{\Delta \hat{x}^2} = \frac{m_0 c}{4 \pi q_0 \hat{c} \hat{c} \hat{c} \hat{c}} \]  
(A.16)

i.e.,

\[ \Delta x = \frac{4 \pi q_0^2}{m_0 c^2} \left( \frac{\hat{c}}{c} \right)^2. \]  
(A.17)
Assuming that one computational macroparticle contains $N$ electrons (or positrons) we can write
\[ \Delta x = 4\pi N \frac{\varrho^2}{m_e} \left( \frac{\dot{c}}{c} \right)^2 = 4\pi N c^2 \Delta \rho, \]  
where we have set $m_0 = m_e$ ($m_e$ is the electron rest mass), $q_0 = e$ ($e$ is the electron charge), and $r = c^2 / m_e c^2 \approx 2.82 \times 10^{-13}$ cm is the classical electron radius. This enables us to express the field scaling, Eq. (A.13), in physical (Gaussian) units
\[ B_0 = \frac{m_e c}{e c} \left( \frac{c}{\Delta \rho c^2} \right) = \frac{e}{r_c c^2 \Delta x}. \]  
The conversions from code units to Gaussian units can then be performed by selecting a length-scale, $\Delta x$, as
\[ \dot{B} = B_0 \dot{B} \approx 1.705 \times 10^3 \frac{\dot{B}}{\dot{c}} \frac{1}{c^2} \frac{1}{\Delta x} \ G, \]  
\[ \dot{E} = B_0 \dot{E} \approx 1.705 \times 10^3 \frac{\dot{E}}{\dot{c}} \frac{1}{c^2} \frac{1}{\Delta x} \ \text{statvolt cm}^{-1}, \]  
\[ \dot{J} = \frac{10^{12}}{4\pi \Delta \rho} \dot{J} = \frac{e c}{4\pi r_c c^2 \Delta x^2} \dot{J} \approx 4.056 \times 10^{12} \frac{\dot{J}}{c^3} \left( \frac{1}{\Delta x} \right)^2 \ \text{statcoul s}^{-1}, \]  
\[ q = \dot{q} \left( \frac{e \Delta x}{4\pi r_c c^2} \right) \approx 1.356 \times 10^2 \frac{\dot{q}}{c^2} \frac{\Delta x}{1 \text{ cm}} \ \text{statcoul}. \]  

Let us next discuss some derived plasma quantities in the simulation. The total relativistic plasma frequency is given as
\[ \omega_p^2 = \omega_{p,-}^2 + \omega_{p,+}^2 = \frac{4\pi q^2 n}{(\gamma) m_e} \left( 1 + \frac{m_-}{m_e} \right), \]  
where $\omega_{p,-}$ and $\omega_{p,+}$ are the plasma frequencies of electrons and ions (or electrons and positrons), respectively, and where we have assumed $n = n_+ = n_-$. For electron-ion plasma $m_+ = 1$ and $m_- = 1836$ (or some smaller reduced proton mass) whereas for electron-positron pair plasma $m_+ = 1$, $m_- = 1$, and $1 + m_- / m_+ = 2$. Time dilatation from a relativistic bulk-motion (if any) is corrected by the mean Lorentz factor of that flow, $\langle \gamma \rangle$. The counterpart in the code units is
\[ \omega_{p,0} = \dot{\omega}_{p,0} \Delta x^2 = \frac{\dot{\omega}_{p,0}}{\Delta x}. \]  

We can then fix the value of charge, $|\dot{q}|$ (and $\dot{m}$), such that the initial plasma frequency for the electrons in the beginning of the simulation is (i.e., requiring $\Delta t = \omega_{p,0}$)
\[ |\dot{q}| = \frac{\omega_{p,0}^2}{N_{ppc} \left( 1 + \frac{m_-}{m_+} \right)}. \]  

Similarly, the relativistic species-specific cyclotron frequency is
\[ \omega_R = \frac{q B}{m_e c \gamma}, \]  
that in code units is simply
\[ \omega_{R \Delta t} = \dot{q} \dot{B} \frac{m_-}{m_e c \gamma m_+}. \]  
where $\dot{m}_+ = m_0$ is the numerical mass of the particle (i.e., 1$\dot{m}$ for pair plasma or 1836$\dot{m}$ for protons). Note that the critical electron cyclotron frequency for $B = B_0$ (i.e., $\dot{B} = 1$) corresponds to $\omega_{R \Delta t} = 1 / \dot{c}$. Both Eqs. (A.25) and (A.29) can be translated to physical units by simply inserting numerical values to the time step, $\Delta x^{-1} = c / \dot{c} \Delta x$. By comparing Eqs. (A.24) vs. (A.25) and Eqs. (A.28) vs. (A.29) we also see that numerical quantities are conveniently obtained by just replacing the physical quantities with the code variables.
Finally, the total plasma magnetization is given as
\[ \sigma = \frac{B^2}{4\pi nm_e c^2(\gamma)} = \left(\frac{\omega_B}{\omega_p}\right)^2 \frac{\hat{B}^2}{N_{ppc} m_e c^2(\gamma)} \left(1 + \frac{m_e}{m_p}\right)^{-1}. \]  
\hspace{1cm} (A.30)

This demonstrates the usefulness of the unit system: numerically we perform less floating-point operations because formulas appear unitless, our typical values are of same order of magnitude reducing floating-point round-off errors, and physical quantities are easily transformed to the code unit system by replacing variables with code quantities (i.e., \(x \to \hat{x}\) and \(n = N_{ppc}\), etc.) and dropping \(4\pi\) factors from Maxwell’s equations.

**Appendix B: Time and space staggering with finite difference method**

Numerical charge conservation and divergence-free \(B\) field preservation follow from the usage of a charge conserving current deposition strategy and of staggered Yee lattice (Yee 1966). See Esirkepov (2001) for more in-depth discussion on the topic.

Let us define discrete versions of the \(\nabla\) operator as
\[ \nabla^+ A_{i,j,k} = \left(\frac{A_{i+1,j,k} - A_{i,j,k}}{\Delta x}, \frac{A_{i,j+1,k} - A_{i,j,k}}{\Delta y}, \frac{A_{i,j,k+1} - A_{i,j,k}}{\Delta z}\right) \]  
and
\[ \nabla^- A_{i,j,k} = \left(\frac{A_{i+1,j,k} - A_{i,j,k}}{\Delta x}, \frac{A_{i,j+1,k} - A_{i,j,k}}{\Delta y}, \frac{A_{i,j,k+1} - A_{i,j,k}}{\Delta z}\right), \]
where \(\Delta x, \Delta y,\) and \(\Delta z\) are the grid spacings. These operators have the following properties
\[ \nabla^- \cdot \nabla^+ \times = \nabla^+ \cdot \nabla^- = 0 \quad \text{and} \quad \nabla^- \cdot \nabla^+ = \nabla^+ \cdot \nabla^- = \hat{\Lambda}, \]
where \(\hat{\Lambda}\) is the discrete Poisson operator in central differences such that
\[ \hat{\Lambda} A_{i,j,k} = \frac{A_{i-1,j,k} - 2A_{i,j,k} + A_{i+1,j,k}}{\Delta x^2} + \frac{A_{i,j-1,k} - 2A_{i,j,k} + A_{i,j+1,k}}{\Delta y^2} + \frac{A_{i,j,k-1} - 2A_{i,j,k} + A_{i,j,k+1}}{\Delta z^2}. \]  
\hspace{1cm} (B.1)

Let us stagger our variables in space and time such that
\[ x = x^n, \]
\[ u = u^{n+\frac{1}{2}}, \]
\[ \rho = \rho^{n+\frac{1}{2}}, \]
\[ J = (J_x; i+\frac{1}{2}, j+\frac{1}{2}, k; J_y; i+\frac{1}{2}, j+\frac{1}{2}, k; J_z; i+\frac{1}{2}, j+\frac{1}{2}, k)^{n+\frac{1}{2}}, \]
\[ E = (E_x; i+\frac{1}{2}, j+\frac{1}{2}, k; E_y; i+\frac{1}{2}, j+\frac{1}{2}, k; E_z; i+\frac{1}{2}, j+\frac{1}{2}, k)^{n}, \]
\[ B = (B_x; i+\frac{1}{2}, j+\frac{1}{2}, k; B_y; i+\frac{1}{2}, j+\frac{1}{2}, k; B_z; i+\frac{1}{2}, j+\frac{1}{2}, k)^{n+\frac{1}{2}}. \]
\hspace{1cm} (B.2)

Maxwell’s equations for \(E\) and \(B\) are given as
\[ \frac{E^{n+\frac{1}{2}} - E^n}{\Delta t} = \nabla^+ \times B^{n+\frac{1}{2}} - 4\pi J^{n+\frac{1}{2}}, \]  
\hspace{1cm} (B.11)
\[ \frac{B^{n+\frac{1}{2}} - B^{n-\frac{1}{2}}}{\Delta t} = -\nabla^- \times E^n, \]  
\hspace{1cm} (B.12)
\[ \nabla^+ \cdot E^n = 4\pi \rho^n, \]  
\hspace{1cm} (B.13)
\[ \nabla^- \cdot B^{n+\frac{1}{2}} = 0. \]  
\hspace{1cm} (B.14)

Discrete form of the charge conservation then follows by operating with \(\nabla^+\) on Eq. (B.11) and substituting (B.13) to the equation that follows. We then obtain
\[ \frac{\rho^{n+\frac{1}{2}} - \rho^n}{\Delta t} + \nabla^+ \cdot J^{n+\frac{1}{2}} = 0. \]  
\hspace{1cm} (B.15)
Similarly, operating on Eq. (B.12) with $\nabla^\cdot$ we obtain divergence free nature of $B$ field

$$\frac{\nabla^\cdot B^{n+\frac{1}{2}} - \nabla^\cdot B^{n-\frac{1}{2}}}{\Delta t} = 0.$$  \hspace{1cm} (B.16)

Thus if

$$\nabla^\cdot E = 4\pi \rho$$  \hspace{1cm} (B.17)

and

$$\nabla^\cdot B = 0$$  \hspace{1cm} (B.18)

at $t = 0$, then divergence of $E$ is always equal to the charge density and $B$ will retain its divergence free nature up to machine precision during the simulation.

**Appendix C: Relativistic Boris scheme**

Let us present the full relativistic Boris scheme here that updates particle four-velocity one time step forwards. This can be found as `Pusher` solver named `BorisPusher` in the `pic` module. In the beginning we have a particle with a four-velocity $u^0$ in units of $c$. The Lorentz update is done via velocities in units of $\hat{c}$ since this is the effective speed-of-light experienced by the electromagnetic fields. To differentiate between these two units we use primed quantities to mark velocities in units of $\hat{c}$.

The scheme begins with a first half of electric field acceleration as

$$u'_1 = \hat{c} u_0 + \frac{1}{2} \hat{s} \hat{E},$$  \hspace{1cm} (C.1)

where

$$\hat{s} = \frac{\hat{q}}{m_z m\pm} = \text{sign}(\hat{q}_z) \frac{m_z}{m\pm},$$  \hspace{1cm} (C.2)

and $m\pm = m/m_e$ is the mass of the particle in units of electron rest mass $m_e$. This four-velocity in units of $\hat{c}$ corresponds to a Lorentz factor of

$$\gamma_1 = \frac{\sqrt{\hat{c}^2 + u'^2_1}}{\hat{c}} = \sqrt{1 + u'^2_1/\hat{c}^2}.$$  \hspace{1cm} (C.3)

We then proceed with a first half of magnetic rotation

$$u'_2 = u'_1 f + \hat{s} \frac{u'_1}{\gamma_1} \times \frac{1}{2} \hat{B} f,$$  \hspace{1cm} (C.4)

where

$$f = \frac{2}{1 + \left(\frac{\hat{B}}{2\gamma_1}\right)^2}.$$  \hspace{1cm} (C.5)

As a final step, we combine the second half of magnetic rotation and second half of the electric field acceleration as

$$u'_3 = u'_1 + \hat{s} \frac{u'_2}{\gamma_1} \times \frac{1}{2} \hat{B} + \frac{1}{2} \hat{s} \hat{E}.$$  \hspace{1cm} (C.6)

Final particle four-velocity in units of $c$ is then

$$u'^{n+1} = \frac{u'_3}{\hat{c}},$$  \hspace{1cm} (C.7)

summarizing a recipe for updating the particle velocity from an initial time $u^n$ to $u'^{n+1}$, i.e., one time step of $\Delta t$ forward.
Appendix D: Current deposition

Let us summarize the charge-conserving ZigZag current deposition scheme [Umeda et al. 2003] implemented as ZigZag solver in RUNKO. First we compute the relay point $x_r$ between previous particle position $x_1$ (and closest grid index $i_1 = (i_1, j_1, k_1)$) and the current position $x_2$ (and closest grid index $i_2$) as

$$x_r = \min \left\{ \min \{i_1, i_2\} + 1, \max \{ \max \{i_1, i_2\}, \frac{1}{2} (x_1 + x_2) \} \right\}.$$ (D.1)

Here we assume element-wise min and max operators as $\min\{a, b\} \equiv (\min\{a_x, b_x\}, \min\{a_y, b_y\}, \min\{a_z, b_z\})$.

The current fluxes are then obtained as

$$F_1 = \hat{q}(x_r - x_1),$$ (D.2)
$$F_2 = \hat{q}(x_2 - x_r).$$ (D.3)

We assume first-order shape functions for the particles. This is related to particle weights defined at the mid points $(x_r + x_1)/2$ and $(x_2 + x_r)/2$ as

$$W_1 = \frac{1}{2} (x_1 + x_r) - i_1,$$ (D.4)
$$W_2 = \frac{1}{2} (x_2 + x_r) - i_2.$$ (D.5)

Depending on the dimensionality, the weights are described as $W_{1,2} = (W_{1,2,x})$ for $D = 1$, $W_{1,2} = (W_{1,2,x}, W_{1,2,y})$ for $D = 2$, or $W_{1,2} = (W_{1,2,x}, W_{1,2,y}, W_{1,2,z})$ for $D = 3$. Finally, the current density, $J = FW$, needs to be injected into adjacent grid points following a charge-conserving deposition strategy. This can be implemented following [Umeda et al. 2003].