Tutorial for 1D hybrid (kinetic ions/massless fluid electrons) simulation code

Kyungguk Min[1] & Kaijun Liu[2]
[1]Korea Astronomy & Space Science Institute, Korea
[2]Southern University of Science and Technology, China

Materials found at:
https://gitlab.com/kyungguk/hybrid-code-tutorial

- Introduction
- Hybrid Scheme & Demo Code Description
- Examples

Hierarchy

1) Individual particles: Klimontovich equation

2) Kinetic: $f = f(\mathbf{v}, \mathbf{x}, t)$

   - Eulerian: Full Vlasov solvers

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} + \frac{q}{m}\left(\mathbf{E} + \frac{\mathbf{v}}{c} \times \mathbf{B}\right) \cdot \frac{\partial f}{\partial \mathbf{v}} = 0$$

   - Lagrangian: Particle-in-cell codes (where PSD is represented by simulation "marker" particles)

   - Darwin, gyro-kinetic, drift kinetic, etc.

3) Hybrid

   - One species kinetic, the other fluid; low-frequency approx. of EM fields

   - Typically kinetic ions and inertial-less, quasi-neutral fluid electrons

4) Fluid

   - MHD

Source: https://www.youtube.com/watch?v=P2Tk61UZr2I

Hybrid Codes

- Hybrid codes arise from the need to model phenomena that do not resolve processes that occur on electron scales.

- Relevant scales: Ion gyro-radius, ion inertial spatial scales, and inverse ion gyro-frequency time scale.

- A full-dynamics PIC code must resolve electron dynamics at scales far smaller than that for ions, even though it is not relevant to the physical problems to be addressed.

- The hybrid approach differs from PIC in that
  - The displacement current is neglected in Ampère-Maxwell's law, eliminating the propagation of light waves;
  - The electrons are modeled as a massless charge-neutralizing fluid; and
  - The electric field is calculated from the generalized Ohm's law.

Governing Equations

- Ion dynamics

$$m_i \frac{\mathrm{d}\mathbf{v}_p}{\mathrm{d}t} = q_i \left(\mathbf{E} + \frac{\mathbf{v}_p}{c} \times \mathbf{B} \right)$$

$$\frac{\mathrm{d}\mathbf{x}_p}{\mathrm{d}t} = \mathbf{v}_p$$

- Electron dynamics –> $\mathbf{E}(n_\mathrm{e}, \mathbf{V}_\mathrm{e}, P_\mathrm{e}, \mathbf{B})$: No explicit time dependence

$$n_e m_e \frac{\mathrm{d}\mathbf{V}_e}{\mathrm{d}t} = 0 = -en_e \left(\mathbf{E} + \frac{\mathbf{V}_e}{c} \times \mathbf{B} \right) - \nabla P_e$$

- Equation of state: $\mathrm{d}(P_e/n_e^\gamma)/\mathrm{d}t = 0$
- Quasi-neutrality: $q_e n_e = q_i n_i$
- Ampere's law

$$\nabla \times \mathbf{B} = \frac{4\pi}{c} J + \cancel{\frac{\partial \mathbf{E}}{c \partial t}}$$

- Faraday's law

$$\frac{\partial \mathbf{B}}{\partial t} = -c(\nabla \times \mathbf{E})$$

Typical Update Cycle and the Challenge

- Ion acceleration: Assume at time step N, $\mathbf{x}_p^N$, $\mathbf{E}^N$, and $\mathbf{B}^N$ are known.

$$\mathbf{v}_p^{N+1/2} = \mathbf{v}_p^{N-1/2} + \frac{q_i \Delta t}{m_i}\left(\mathbf{E}^N + \frac{\mathbf{v}_p^N}{c} \times \mathbf{B}^N\right)$$

$$\mathbf{x}_p^{N+1} = \mathbf{x}_p^N + \mathbf{v}_p^{N+1/2}\Delta t$$

- Collect current at N + 1/2 and density at N + 1.
- Update **B** by a half-time step.

$$\mathbf{B}^{N+1/2} = \mathbf{B}^N - \frac{c\Delta t}{2}(\nabla \times \mathbf{E}^N)$$

- Calculate **E** using the quantities at N+1/2.

$$\mathbf{E}^{N+1/2} = -\frac{\mathbf{V}_i^{N+1/2}}{c} \times \mathbf{B}^{N+1/2} - \frac{\nabla P_e^{N+1/2}}{q_i n_i^{N+1/2}} - \frac{\mathbf{B}^{N+1/2} \times (\nabla \times \mathbf{B}^{N+1/2})}{4\pi q_i n_i^{N+1/2}}$$

$$= \mathbf{E}(\mathbf{B}^{N+1/2}, n_i^{N+1/2}, \mathbf{V}_i^{N+1/2})$$

- Update **B** by a half-time step.

$$\mathbf{B}^{N+1} = \mathbf{B}^{N+1/2} - \frac{c\Delta t}{2}(\nabla \times \mathbf{E}^{N+1/2})$$

- To complete the cycle, we need **E** at N+1, but $\mathbf{V}_i^{N+1}$ is not known.
- The evolution of hybrid codes has been developing a good algorithm that best calculates **E** at the time step when the velocity is not available.

Algorithm 1: Predictor-corrector Method

- Execute one or more of provisional updates to fine-tune **E** at the desired time step.
  – Pros: Implementation is straightforward.
  – Cons: This involves multiple provisional particle pushes which are the most time-consuming part in PIC-type code.

- Basic idea:
  1. Make a prediction of the fields at N+1
  2. Advance particles in the predicted field to get the ion source terms
  3. Use current and density to compute predicted fields at N+3/2
  4. Determine corrected field at N+1

Algorithm 2: Current Advanced Method

- Since the problem is not knowing $\mathbf{V}_i^{N+1}$, one can extrapolate $\mathbf{V}_i^{N+1}$ using previously saved values.
- The most popular approach along that line is to calculate the current at N+1 by doing an extra half-time step push.

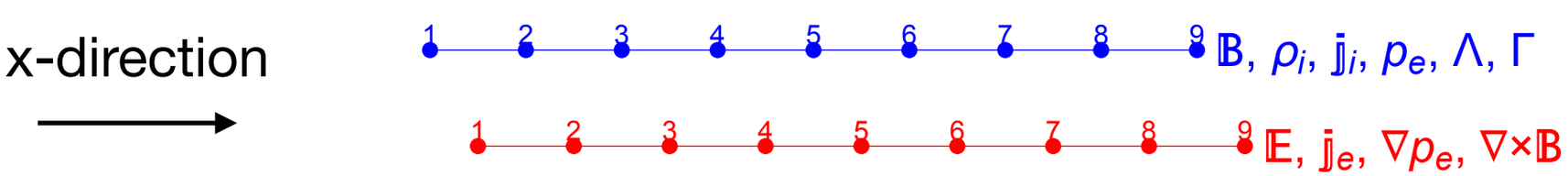
$$\mathbf{V}'^{N+1}_i = \mathbf{V}^{N+1/2}_i + \frac{\Delta t q_i}{2m_i}\left(\mathbf{E}^* + \frac{\mathbf{V}^{N+1/2}_i}{c} \times \mathbf{B}^{N+1}\right)$$

$$\mathbf{E}^* = \mathbf{E}(\mathbf{B}^{N+1}, n_i^{N+1}, \mathbf{V}_i^{N+1/2})$$

- Because the current is defined at grid points, this extra step is trivial. And yet, it results in good energy conservation.

Demo Code

- 1.5D: One-dimensional dependence and all three components of the field vectors.
- It assumes a homogeneous, collisionless, quasi-neutral plasma in a uniform background magnetic field.
- Both Predictor-corrector (Kunz et al., 2014) and CAM-CL (Matthew, 1994) algorithms are implemented.
- PIC technique and implementation are based on the full-dynamics code written by Liu (2007).
- Particles are advanced using the usual leap-frog algorithm with Boris push.
- **B** and **E** are saved at staggered grid points as follows:



- Other quantities are defined so as to the 2nd-order centered difference makes sense as much as possible.
- All vector components are placed at the same grid point (not entirely Yee lattice).
- Three-point Hamming filter is applied to source terms to reduce grid-scale fluctuations:

$$\Phi_j^{filtered} = \frac{1}{4}\left(\Phi_{j-1}^{source} + 2\Phi_j^{source} + \Phi_{j+1}^{source}\right)$$

Change of Variables

$$\begin{cases} \dfrac{q_0}{m_0 c}\mathbf{E} = \overrightarrow{\Xi} \\ \dfrac{q_0}{m_0 c}\mathbf{B} = \overrightarrow{\Omega} \\ \dfrac{4\pi q_0}{m_0}\rho = \Pi \\ \dfrac{4\pi q_0}{m_0 c}\mathbf{J} = \overrightarrow{\mathscr{J}} \\ \dfrac{4\pi q_0^2}{m_0^2 c^2}P_e = \mathscr{P}_e \end{cases}$$

$$\omega_s = \sqrt{\frac{4\pi q_s^2 n_s(0)}{m_s}}$$

$$\Omega_s = \frac{q_s B_0}{m_s c}$$

$$\begin{cases} \Pi_e = -\Pi_i = -\sum_{s\neq e}\Pi_s & \text{Charge neutrality} \\ \dfrac{\partial\overrightarrow{\Omega}}{\partial t} = -c\nabla\times\overrightarrow{\Xi} & \text{Faraday's law} \\ \overrightarrow{\mathscr{J}} = c\nabla\times\overrightarrow{\Omega} & \text{Ampere's law} \\ \overrightarrow{\Xi} = \dfrac{1}{\Pi_i}\left[\left(\overrightarrow{\mathscr{J}} - \overrightarrow{\mathscr{J}}_i\right)\times\overrightarrow{\Omega} - c\nabla\mathscr{P}_e\right] & \text{Generalized Ohm's law} \\ \dfrac{\mathrm{d}\mathbf{x}_p}{\mathrm{d}t} = \mathbf{v}_p & \text{Velocity} \\ \dfrac{\mathrm{d}\mathbf{v}_p}{\mathrm{d}t} = \dfrac{\Omega_s}{\Omega_0}\left(c\overrightarrow{\Xi} + \mathbf{v}_p\times\overrightarrow{\Omega}\right) & \text{Lorentz force} \\ \dfrac{\partial\overrightarrow{\mathscr{J}}_i}{\partial t} = \sum_{s\neq e}\Pi_s\dfrac{\Omega_s}{\Omega_0}\left(\overrightarrow{\Xi} + \dfrac{\mathbf{V}_s}{c}\times\overrightarrow{\Omega}\right) & \text{Current Advance} \\ \qquad = \Lambda_i\overrightarrow{\Xi} + \overrightarrow{\Gamma}_i\times\overrightarrow{\Omega} \end{cases}$$

Background B

Bulk velocity

- Electron pressure:

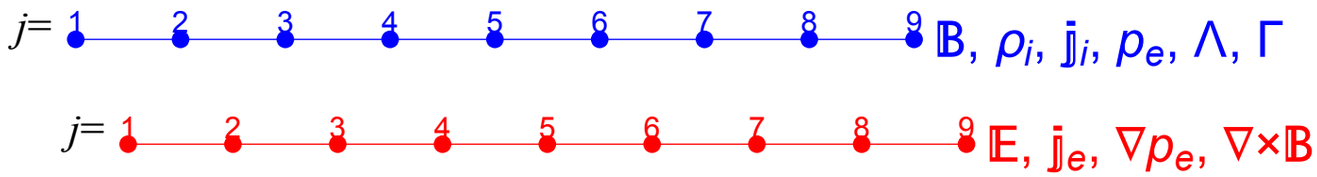
$$\mathscr{P}_e(t) = \Omega_0^2\frac{\beta_e}{2}\left(\frac{\Pi_e(t)}{\Pi_e(0)}\right)^\gamma = \Omega_0^2\frac{\beta_e}{2}\left(-\frac{\Omega_e\Pi_i(t)}{\Omega_0\omega_e^2}\right)^\gamma, \text{where } \omega_e^2 = \frac{\Omega_e}{\Omega_0}\Pi_e(0)$$

Change of Variables

- With new variables, a population of ions is described by their cyclotron frequency (Ω_s), plasma frequency (ω_s) (and temperatures, etc. depending on velocity distribution functions), instead of mass, charge, and the number density.
- The electric field (Ξ) and the magnetic field (Ω) have units of frequency, and are understood to be a sum of the static and perturbed part (i.e., $\mathbf{\Omega} = \mathbf{\Omega}_0 + \delta\mathbf{\Omega}$).
- Whereas, charge, current, and energy densities (or pressure) has units of frequency squared.
- The electric and magnetic field energy densities are $\Xi^2/2$ and $\Omega^2/2$, which then have units of frequency squared.
- Ω_0 denotes magnitude of the uniform background magnetic field. Time is in units of Ω_0^{-1}.
- The ratio Ω_s/Ω_0 describes the kind of ions. As an example, if $\Omega_p/\Omega_0 = 2\pi$, protons gyrate once per Ω_0^{-1}. For He^+, this ratio is then $2\pi/4$; and for He^{2+}, it is $2\pi/2$.

Magnetic Field Update

x-direction



- Magnetic field

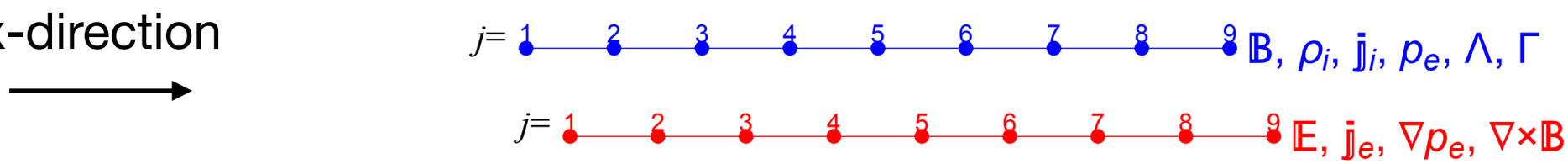
$$\frac{\partial \overrightarrow{\Omega}}{\partial t} = -c\nabla \times \overrightarrow{\Xi} \longrightarrow \begin{cases} \Omega_{x;j}^{t+\Delta t} - \Omega_{x;j}^{t} = 0; \\ \Omega_{y;j}^{t+\Delta t} - \Omega_{y;j}^{t} = \dfrac{c\Delta t}{\Delta x}\left(+\Xi_{z;j-0}^{t+\Delta t/2} - \Xi_{z;j-1}^{t+\Delta t/2}\right); \\ \Omega_{z;j}^{t+\Delta t} - \Omega_{z;j}^{t} = \dfrac{c\Delta t}{\Delta x}\left(-\Xi_{y;j-0}^{t+\Delta t/2} + \Xi_{y;j-1}^{t+\Delta t/2}\right). \end{cases}$$

```
//  BField.cc

void H1D::BField::_update(BField &B, EField const &E, Real cdtODx) noexcept
{
    for (long i = 0; i < E.size(); ++i) {
        B[i].x += 0;
        B[i].y += (+E[i-0].z -E[i-1].z) * cdtODx;
        B[i].z += (-E[i-0].y +E[i-1].y) * cdtODx;
    }
}
```

$\frac{c\Delta t}{\Delta x}$

Electric Field Calculation

x-direction



- Electron pressure
  - $\beta_e = n_e(0)T_e(0)/(B_0^2/8\pi)$
  - $\omega_e = \sqrt{(4\pi e^2 n_e(0)/m_e)}$; initial electron plasma frequency
  - $\Omega_e = -eB_0/(m_e c)$; initial electron cyclotron frequency

$$\mathscr{P}_e(t) = \Omega_0^2\frac{\beta_e}{2}\left(\frac{\Pi_e(t)}{\Pi_e(0)}\right)^\gamma = \Omega_0^2\frac{\beta_e}{2}\left(-\frac{\Omega_e\Pi_i(t)}{\Omega_0\omega_e^2}\right)^\gamma$$

```
//  EField.cc

void H1D::EField::_update_Pe(decltype(Pe) &Pe, Charge const &rho) noexcept
{
    namespace eFluid = Input::eFluid;
    //
    using Input::O0;
    Real const O02beO2 = (O0*O0)*eFluid::beta*0.5;
    Real const mOeOOOoe2 = -eFluid::Oc/(O0*(eFluid::op*eFluid::op));
    Real const gamma = Real{eFluid::closure/10}/(eFluid::closure%10);
    for (long i = -Pad; i < Pe.size() + Pad; ++i) {
        Pe[i] = std::pow(mOeOOOoe2*Real{rho[i]}, gamma) * O02beO2;
    }
}
```

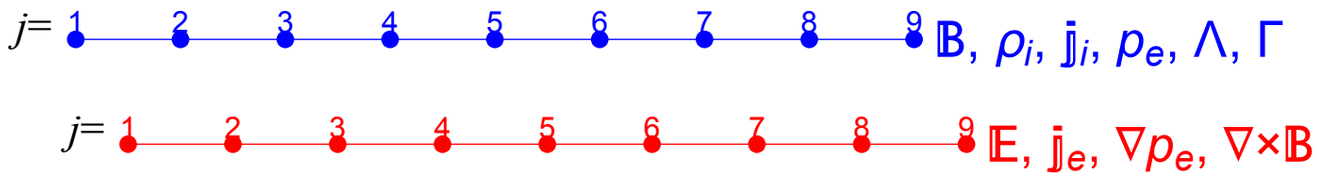
$\Omega_0^2\frac{\beta_e}{2}$ → `O02beO2`

$-\frac{\Omega_e}{\Omega_0\omega_e^2}$ → `mOeOOOoe2`

Π_i → `rho[i]`

Electric Field Calculation (cont'd)

x-direction



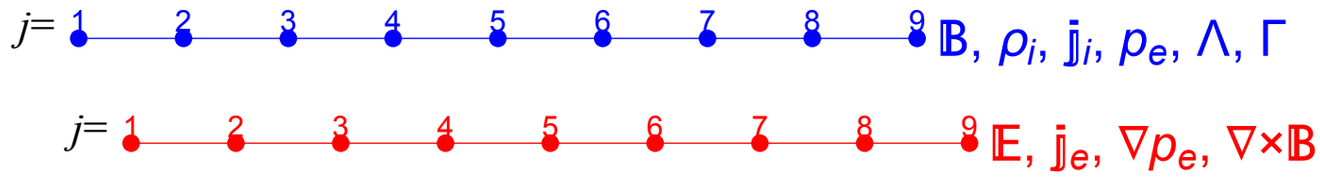
- Electron current: $\mathbf{J}_e = \mathbf{J} - \mathbf{J}_i = c\nabla x\mathbf{\Omega} - \mathbf{J}_i$

$$
\begin{cases}
\mathscr{J}_{e,x;j} = -\dfrac{\mathscr{J}_{i,x;j+1} + \mathscr{J}_{i,x;j+0}}{2} + 0; \\
\mathscr{J}_{e,y;j} = -\dfrac{\mathscr{J}_{i,y;j+1} + \mathscr{J}_{i,y;j+0}}{2} + \dfrac{c}{\Delta x}\left(-\Omega_{z;j+1} + \Omega_{z;j+0}\right); \\
\mathscr{J}_{e,z;j} = -\dfrac{\mathscr{J}_{i,z;j+1} + \mathscr{J}_{i,z;j+0}}{2} + \dfrac{c}{\Delta x}\left(+\Omega_{y;j+1} - \Omega_{y;j+0}\right).
\end{cases}
$$

```
//  EField.cc

void H1D::EField::_update_Je(decltype(Je) &Je, Current const &Ji, BField const &B) noexcept
{
    Real const cODx = Input::c/Input::Dx;
    for (long i = 0; i < B.size(); ++i) {
        // J total
        //
        Je[i].x = 0;
        Je[i].y = (-B[i+1].z +B[i+0].z)*cODx;
        Je[i].z = (+B[i+1].y -B[i+0].y)*cODx;

        // Je = J - Ji
        //
        Je[i] -= (Ji[i+1] + Ji[i+0])*0.5;
    }
}
```

Electric Field Calculation (cont'd)

x-direction



- Electric field

$$\overrightarrow{\Xi} = \frac{2}{\Pi_{i;j+1} + \Pi_{i;j+0}} \left[\overrightarrow{\mathscr{J}}_e \times \frac{\overrightarrow{\Omega}_{j+1} + \overrightarrow{\Omega}_{j+0}}{2} - c\nabla\mathscr{P}_e \right] \longleftarrow \begin{cases} (c\nabla\mathscr{P}_e)_{x;j} = \dfrac{c}{\Delta x}(\mathscr{P}_{e;j+1} - \mathscr{P}_{e;j+0}); \\ (c\nabla\mathscr{P}_e)_{y;j} = 0; \\ (c\nabla\mathscr{P}_e)_{z;j} = 0. \end{cases}$$

```
//  EField.cc

void H1D::EField::_update_E(EField &E, BField const &B, Charge const &rho) noexcept
{
    Real const cODx = Input::c/Input::Dx;
    for (long i = 0; i < E.size(); ++i) {
        Vector &Ei = E[i];

        // 1. Je x B term
        //
        Ei = cross(Je[i], (B[i+1] + B[i+0])*0.5);

        // 2. pressure gradient term
        //
        Ei.x -= Real{Pe[i+1] -Pe[i+0]}*cODx;
        Ei.y -= 0;
        Ei.z -= 0;

        // 3. divide by charge density
        //
        Ei /= Real{rho[i+1] + rho[i+0]}*0.5;
    }
}
```

Velocity Moments

- Solving field equations only requires quantities weighted by a velocity distribution function and summed over velocity space.

0th: $$n_s = \langle 1 \rangle_s = \int (1) f_s(t, \mathbf{x}, \mathbf{v}) \mathrm{d}^3\mathbf{v} \longrightarrow \text{Charge density} = q_s \langle 1 \rangle_s$$

1st: $$n_s \mathbf{V}_s = \langle \mathbf{v} \rangle_s = \int (\mathbf{v}) f_s(t, \mathbf{x}, \mathbf{v}) \mathrm{d}^3\mathbf{v} \longrightarrow \text{Current density} = q_s \langle \mathbf{v} \rangle_s$$

2nd: $$\langle \mathbf{vv} \rangle_s = \int (\mathbf{vv}) f_s(t, \mathbf{x}, \mathbf{v}) \mathrm{d}^3\mathbf{v}$$

$$\longrightarrow \quad \text{Kinetic energy density} = m_s \sum_{j=x,y,z} \langle \frac{v_j^2}{2} \rangle_s$$

$$\longrightarrow \quad \text{Pressure Tensor} = m_s \left(\langle \mathbf{vv} \rangle_s - n_s \mathbf{V}_s \mathbf{V}_s \right)$$

- How can velocity moments be obtained? More generally, what does the operation $\langle ... \rangle$ mean in a PIC code?

Velocity Moments (cont'd)

- Imagine equally-spaced simulation grid points.
- Initially, the number density $n_s(0)$ is spatially uniform and the particle velocity distribution is represented by $N_c = 10$ marker (simulation) particles.

t = 0

$j-1$	j Cell	$j+1$
10	10	10

- Over time, some particles will leave the j cell; Likewise, some also enter the j cell.

t > 0

$j-1$	j Cell	$j+1$
20	5	10

$n_s(t) = 2 \times n_s(0)$ $\quad$ $n_s(t) = 1 \times n_s(0)$

$n_s(t) = 0.5 \times n_s(0)$

- In the figure above, the number of particles in the jth cell decreased by a factor of 2; that means $n_s(t)$ there is also half of the initial value.
- Formally,

Shape function

$$\frac{n_{s;j}(t)}{n_s(0)} = \sum_{p=1}^{N_{s,p}} \frac{S(x_{s,p}(t) - X_j)}{N_{c,s}}$$

Velocity Moments (cont'd)

$$n_s = \langle 1 \rangle_s = \int (1) f_s \mathrm{d}^3\mathbf{v}$$

$$\frac{\int (1) f_s(t, \mathbf{x}, \mathbf{v}) \mathrm{d}^3\mathbf{v}}{n_s(0)} = \frac{\sum_{p=1}^{N_{s,p}} (1)\, S(x_{s,p}(t) - X_j)}{N_{c,s}}$$

Generalization ↓

$$\langle \ldots \rangle_s = \int (\ldots) f_s(t, \mathbf{x}, \mathbf{v}) \mathrm{d}^3\mathbf{v} = n_s(0) \frac{\sum_{p=1}^{N_{s,p}} (\ldots)\, S(x_{s,p}(t) - X_j)}{N_{c,s}}$$

$$\omega_s^2 = \frac{4\pi q_s^2 n_s(0)}{m_s} = \frac{\Omega_s}{\Omega_0} \Pi_s(0)$$

$$\Pi_{i;j}(t) = \sum_{s \neq e} \frac{\Omega_0}{\Omega_s} \omega_s^2 \langle 1 \rangle'_s$$

$$\overrightarrow{\mathscr{J}}_{i;j}(t) = \sum_{s \neq e} \frac{\Omega_0}{\Omega_s} \frac{\omega_s^2}{c} \langle \mathbf{v} \rangle'_s$$

$$\mathscr{E}_{i;j}(t) = \sum_{s \neq e} \frac{\Omega_0^2}{\Omega_s^2} \frac{\omega_s^2}{c^2} \langle v^2/2 \rangle'_s$$

where

$$\langle \ldots \rangle'_s = \frac{\langle \ldots \rangle_s}{n_s(0)} = \frac{\sum_{p=1}^{N_{s,p}} (\ldots)\, S(x_{s,p}(t) - X_j)}{N_{c,s}}$$

Velocity Moments (cont'd)

```
//  Species.cc

void H1D::Species::_collect_all(GridQ<Scalar> &n, GridQ<Vector> &nV, GridQ<Tensor> &nvv) const
{
    n.fill(Scalar{0});
    nV.fill(Vector{0});
    nvv.fill(Tensor{0});
    Tensor tmp{0};
    ::Shape sx;
    for (Particle const &ptl : bucket) {
        sx(ptl.pos_x); // position is normalized by grid size
        n.deposit(sx, 1);
        nV.deposit(sx, ptl.vel);
        tmp.hi() = tmp.lo() = ptl.vel;
        tmp.lo() *= ptl.vel; // diagonal part; {vx*vx, vy*vy, vz*vz}
        tmp.hi() *= {ptl.vel.y, ptl.vel.z, ptl.vel.x}; // off-diag part; {vx*vy, vy*vz, vz*vx}
        nvv.deposit(sx, tmp);
    }
    //
    n /= Scalar{Nc};
    nV /= Vector{Nc};
    nvv /= Tensor{Nc};
}
```

$\langle 1 \rangle'_s$ →

$\langle \mathbf{v} \rangle'_s$ →

$\langle \mathbf{vv} \rangle'_s$ →

Shape Function →

(a) ←

(b) ←

(c) ←

$$\frac{\sum_{p=1}^{N_{s,p}} (\dots)\, S(x_{s,p}(t) - X_j)}{N_{c,s}}$$

$$\begin{cases} (a) \rightarrow \sum_{p=1}^{N_{s,p}} (1)\, S(x_{s,p}(t) - X_j) \\ (b) \rightarrow \sum_{p=1}^{N_{s,p}} (\mathbf{v}_p)\, S(x_{s,p}(t) - X_j) \\ (c) \rightarrow \sum_{p=1}^{N_{s,p}} (\{v_x^2, v_y^2, v_z^2, v_x v_y, v_y v_z, v_z v_x\})\, S(x_{s,p}(t) - X_j) \end{cases}$$

Velocity Moments (cont'd)

$$\Pi_i = \sum_{s \neq e} \frac{\Omega_0}{\Omega_s} \omega_s^2 \langle 1 \rangle'_s$$

```
//  Charge.cc

H1D::Charge &H1D::Charge::operator+=(Species const &sp) noexcept
{
    ::accumulate(this->dead_begin(), sp.moment<0>().dead_begin(), sp.moment<0>().dead_end(), sp.charge_density_conversion_factor());
    return *this;
}
```

Π_i $\langle 1 \rangle'_s$ $\frac{\Omega_0}{\Omega_s} \omega_s^2$

$$\vec{\mathscr{F}}_i = \sum_{s \neq e} \frac{\Omega_0}{\Omega_s} \frac{\omega_s^2}{c} \langle \mathbf{v}_p \rangle'_s$$

```
//  Current.cc

H1D::Current &H1D::Current::operator+=(Species const &sp) noexcept
{
    ::accumulate(this->dead_begin(), sp.moment<1>().dead_begin(), sp.moment<1>().dead_end(), sp.current_density_conversion_factor());
    return *this;
}
```

$\vec{\mathscr{F}}_i$ $\langle \mathbf{v}_p \rangle'_s$ $\frac{\Omega_0}{\Omega_s} \frac{\omega_s^2}{c}$

Current Advance Algorithm

$$
\begin{cases}
\dfrac{\partial \overrightarrow{\mathscr{J}}_i}{\partial t} = \Lambda_i \overrightarrow{\Xi} + \overrightarrow{\Gamma}_i \times \overrightarrow{\Omega}, \\
\text{where } \Lambda_i = \displaystyle\sum_{s \neq e} \omega_s^2 \langle 1 \rangle_s' = \sum_{s \neq e} \frac{\Omega_s}{\Omega_0} \left(\frac{\Omega_0}{\Omega_s} \omega_s^2 \langle 1 \rangle_s' \right), \\
\text{and } \overrightarrow{\Gamma}_i = \displaystyle\sum_{s \neq e} \frac{\omega_s^2}{c} \langle \mathbf{v}_p \rangle_s' = \sum_{s \neq e} \frac{\Omega_s}{\Omega_0} \left(\frac{\Omega_0}{\Omega_s} \frac{\omega_s^2}{c} \langle \mathbf{v}_p \rangle_s' \right).
\end{cases}
$$

```
//  Charge.cc

H1D::Lambda &H1D::Lambda::operator+=(Species const &sp) noexcept
{
    ::accumulate(this->dead_begin(), sp.moment<0>().dead_begin(), sp.moment<0>().dead_end(),
                                                                  sp.charge_density_conversion_factor()*sp.Oc/Input::O0);
    return *this;
}
```

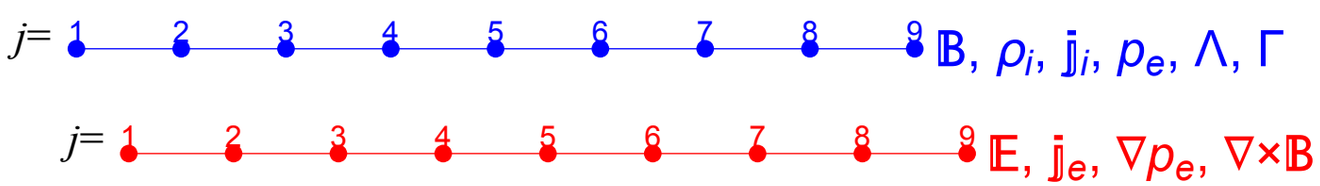
Λ_i $\langle 1 \rangle_s'$ $\dfrac{\Omega_0}{\Omega_s}\omega_s^2$ $\dfrac{\Omega_s}{\Omega_0}$

```
//  Current.cc

H1D::Gamma &H1D::Gamma::operator+=(Species const &sp) noexcept
{
    ::accumulate(this->dead_begin(), sp.moment<1>().dead_begin(), sp.moment<1>().dead_end(),
                                                                  sp.current_density_conversion_factor()*sp.Oc/Input::O0);
    return *this;
}
```

$\overrightarrow{\Gamma}_i$ $\langle \mathbf{v}_p \rangle_s'$ $\dfrac{\Omega_0}{\Omega_s}\dfrac{\omega_s^2}{c}$ $\dfrac{\Omega_s}{\Omega_0}$

Current Advance Algorithm (cont'd)

x-direction



$$\overrightarrow{\mathcal{J}}^{t+\Delta t}_{i;j} - \overrightarrow{\mathcal{J}}^{t}_{i;j} = \Delta t \left(\Lambda_{i;j} \frac{\overrightarrow{\Xi}_{j-0} + \overrightarrow{\Xi}_{j-1}}{2} + \overrightarrow{\Gamma}_{i;j} \times \overrightarrow{\Omega}_{j} \right)$$

```
//  Current.cc

void H1D::Current::_advance(Current &J, Lambda const &L, Gamma const &G, BField const &B,
EField const &E, Real const dt) noexcept
{
    for (long i = 0; i < J.size(); ++i) {
        Vector const &Ei = (E[i-0] + E[i-1])*0.5;
        J[i] += (Ei*Real{L[i]} + cross(G[i], B[i])) *= dt;
    }
}
```

Update Cycle: Kunz's PC

1. Predict **B** and **E** at N+1/2

$$\mathbf{B}'^{N+1} = \mathbf{B}^N - c\Delta t(\nabla \times \mathbf{E}^N)$$
$$\mathbf{E}'^{N+1} = \mathbf{E}(\mathbf{B}'^{N+1}, \rho_i^N, \mathbf{J}_i^N)$$
$$\mathbf{B}'^{N+1/2} = (\mathbf{B}'^{N+1} + \mathbf{B}^N)/2$$
$$\mathbf{E}'^{N+1/2} = (\mathbf{E}'^{N+1} + \mathbf{E}^N)/2$$

```
//  Domain_PC.cc
//
// 1. Faraday's law; predict 1
//
bfield_1 = bfield_0;
bfield_1.update(efield_0, dt), delegate->pass(domain, bfield_1);
//
// 2. Ohm's law; predict 1
//
efield_1.update(bfield_1, charge, current), delegate->pass(domain, efield_1);
//
// 3. Average fields
//
(bfield_1 += bfield_0) *= Vector{.5};
(efield_1 += efield_0) *= Vector{.5};
```

2. Provisional particle push

$$\mathbf{x}'^{N+1/2}_p = \mathbf{x}^N_p + \mathbf{v}^N_p \Delta t/2$$
$$\mathbf{v}'^{N+1}_p = \mathbf{v}^N_p + \frac{q_i}{m_i c}\Delta t(c\mathbf{E}'^{N+1/2} + \mathbf{v}^{N+1/2}_p \times \mathbf{B}'^{N+1/2})$$
$$\mathbf{x}'^{N+1}_p = \mathbf{x}'^{N+1/2}_p + \mathbf{v}'^{N+1}_p \Delta t/2$$

```
//  Domain_PC.cc
//
// 4 & 5. Particle push and deposit charge and
//                current densities; predict
//
charge.reset();
current.reset();
for (Species const &sp : species) {
    predictor = sp;
    predictor.update_pos(dt/2.0, 0.5);
    predictor.update_vel(bfield_1, efield_1, dt);
    predictor.update_pos(dt/2.0, 0.5),
              delegate->pass(domain, predictor);
```

3. Collect provisional charge and current densities: ρ'^{N+1}_i, $\mathbf{J}'^{N+1}_i$

```
    predictor.collect_part();
    charge += predictor;
    current += predictor;
}
delegate->gather(domain, current),
          delegate->pass(domain, current);
delegate->gather(domain, charge),
          delegate->pass(domain, charge);
```

Update Cycle: Kunz's PC (cont'd)

4. Refine **B** and **E** at N+1/2

$$\mathbf{B}'^{N+1} = \mathbf{B}^N - c\Delta t(\nabla \times \mathbf{E}'^{N+1/2})$$

$$\mathbf{E}'^{N+1} = \mathbf{E}(\mathbf{B}'^{N+1}, \rho_i'^N, \mathbf{J}_i'^N)$$

$$\mathbf{B}'^{N+1/2} = (\mathbf{B}'^{N+1} + \mathbf{B}^N)/2$$

$$\mathbf{E}'^{N+1/2} = (\mathbf{E}'^{N+1} + \mathbf{E}^N)/2$$

```
//  Domain_PC.cc
//
// 6. Faraday's law; predict 2
//
bfield_1 = bfield_0;
bfield_1.update(efield_1, dt), delegate->pass(domain, bfield_1);
//
// 7. Ohm's law; predict 2
//
efield_1.update(bfield_1, charge, current), delegate->pass(domain, efield_1);
//
// 8. Average fields
//
(bfield_1 += bfield_0) *= Vector{.5};
(efield_1 += efield_0) *= Vector{.5};
```

5. Particle push

$$\mathbf{x}_p^{N+1/2} = \mathbf{x}_p^N + \mathbf{v}_p^N \Delta t/2$$

$$\mathbf{v}_p^{N+1} = \mathbf{v}_p^N + \frac{q_i}{m_i c}\Delta t(c\mathbf{E}'^{N+1/2} + \mathbf{v}_p^{N+1/2} \times \mathbf{B}'^{N+1/2})$$

$$\mathbf{x}_p^{N+1} = \mathbf{x}_p^{N+1/2} + \mathbf{v}_p^{N+1} \Delta t/2$$

```
//  Domain_PC.cc
//
// 9 & 10. Particle push and deposit charge and
                                    current densities; correct
//
charge.reset();
current.reset();
for (Species &sp : species) {
    sp.update_pos(dt/2.0, 0.5);
    sp.update_vel(bfield_1, efield_1, dt);
    sp.update_pos(dt/2.0, 0.5),
                  delegate->pass(domain, sp);
```

6. Collect corrected charge and current densities: ρ_i^{N+1}, $\mathbf{J}_i^{N+1}$

```
    sp.collect_part();
    charge += sp;
    current += sp;
}
delegate->gather(domain, current),
          delegate->pass(domain, current);
delegate->gather(domain, charge),
          delegate->pass(domain, charge);
```

Update Cycle: Kunz's PC (cont'd)

7. Calculate corrected **B** and **E** at N+1

$$\mathbf{B}^{N+1} = \mathbf{B}^{N} - c\Delta t(\nabla \times \mathbf{E}'^{N+1/2})$$
$$\mathbf{E}^{N+1} = \mathbf{E}(\mathbf{B}^{N+1}, \rho_i^N, \mathbf{J}_i^N)$$

```
//  Domain_PC.cc
//
// 11. Faraday's law; correct
//
bfield_0.update(efield_1, dt), delegate->pass(domain, bfield_0);
//
// 12. Ohm's law; correct
//
efield_0.update(bfield_0, charge, current), delegate->pass(domain, efield_0);
```

Update Cycle: Matthews' CAM-CL

1. Advance particle velocity from N-1/2 to N+1/2 and collect moments

$$\mathbf{v}_p^{N+1/2} = \mathbf{v}_p^{N-1/2} + \frac{q_i}{m_i c}\Delta t(c\mathbf{E}^N + \mathbf{v}_p^N \times \mathbf{B}^N)$$

$$\rho_i^N = \rho(\mathbf{x}_p^N)$$

$$\mathbf{J}_i^- = \mathbf{J}(\mathbf{x}_p^N, \mathbf{v}_p^{N+1/2})$$

2. Advance particle position from N to N+1 and collect moments

$$\mathbf{x}_p^{N+1} = \mathbf{x}_p^N + \mathbf{v}_p^{N+1/2}\Delta t$$

$$\rho_i^{N+1} = \rho(\mathbf{x}_p^{N+1})$$

$$\mathbf{J}_i^+ = \mathbf{J}(\mathbf{x}_p^{N+1}, \mathbf{v}_p^{N+1/2})$$

$$\Lambda_i^{N+1} = \Lambda(\mathbf{x}_p^{N+1})$$

$$\mathbf{\Gamma}_i^+ = \mathbf{\Gamma}(\mathbf{x}_p^{N+1}, \mathbf{v}_p^{N+1/2})$$

3. Calculate charge and current at N+1/2

$$\rho_i^{N+1/2} = (\rho_i^N + \rho_i^{N+1})/2$$

$$\mathbf{J}_i^{N+1/2} = (\mathbf{J}_i^- + \mathbf{J}_i^+)/2$$

```
//  Domain_CAMCL.cc
// 1 & 2. update velocities and positions by full step and
                              collect charge and current densities
//
current_0.reset();
current_1.reset();
charge_0.reset();
charge_1.reset();
lambda.reset();
gamma.reset();
for (Species &sp : species) {
     sp.update_vel(bfield, efield, dt); // v^N-1/2 -> v^N+1/2
     sp.collect_part();
     current_0 += sp; // J^-
     charge_0 += sp; // rho^N


     sp.update_pos(dt, 1),
                    delegate->pass(domain, sp); // x^N -> x^N+1
     sp.collect_part();
     current_1 += sp; // J^+
     charge_1 += sp; // rho^N+1
     lambda += sp; // L^N+1
     gamma += sp; // G^+
}
//
// 3. gather, smooth and average charge and current densities
//
delegate->gather(domain, current_0),
           delegate->pass(domain, current_0);
delegate->gather(domain, charge_0),
           delegate->pass(domain, charge_0);
delegate->gather(domain, current_1),
           delegate->pass(domain, current_1);
delegate->gather(domain, charge_1),
           delegate->pass(domain, charge_1);
delegate->gather(domain, lambda);
delegate->gather(domain, gamma);



(charge_0 += charge_1) *= Scalar{.5}; // rho^N+1/2
(current_0 += current_1) *= Vector{.5}; // J^N+1/2
```

Update Cycle: Matthews' CAM-CL (cont'd)

4. Calculate **B** at N+1 using subcycling algorithm

$$\mathbf{B}^{N+1} = \mathbf{B}^{N} - c\int_{t^N}^{t^{N+1}} \nabla \times \mathbf{E}(\mathbf{B}(t), \rho_i^{N+1/2}, \mathbf{J}_i^{N+1/2})\mathrm{d}t$$

```
//  Domain_CAMCL.cc
//
// 4. subcycle magnetic field by full step
//
subcycle(domain, charge_0, current_0, dt);
```

5. Advance current by half step

$$\mathbf{E}^{*} = \mathbf{E}(\mathbf{B}^{N+1}, \rho_i^{N+1}, \mathbf{J}_i^{N+1/2})$$

$$\mathbf{J}_i^{N+1} = \mathbf{J}^{+} + \frac{\Delta t}{2}(\Lambda_i^{N+1}\mathbf{E}^{*} + \mathbf{\Gamma}_i^{+} \times \mathbf{B}^{N+1})$$

```
//
// 5. calculate electric field* and
                          advance current density
//
efield.update(bfield, charge_1, current_0),
                          delegate->pass(domain, efield);
current_1.advance(lambda, gamma, bfield, efield, dt/2.0),
                          delegate->pass(domain, current_1);
```

6. Calculate **E** at N+1

$$\mathbf{E}^{N+1} = \mathbf{E}(\mathbf{B}^{N+1}, \rho_i^{N+1}, \mathbf{J}_i^{N+1})$$

```
//
// 6. calculate electric field
//
efield.update(bfield, charge_1, current_1),
                          delegate->pass(domain, efield);
```

Update Cycle: Matthews' CAM-CL (cont'd)

– Magnetic field subcycling algorithm (*m* is the number of sub-cycles)

Define $\mathbf{E}_j = \mathbf{E}(\mathbf{B}_j, \rho_i^{N+1/2}, \mathbf{J}_i^{N+1/2})$, where $\mathbf{B}_j = \mathbf{B}^{N+j/m}$ and $\delta t = \Delta t/m$.

$$\mathbf{B}_1 = \mathbf{B}_0 - c\delta t \nabla \times \mathbf{E}_0, \quad \text{// prologue}$$

$$\mathbf{B}_{j+1} = \mathbf{B}_{j-1} - 2c\delta t \nabla \times \mathbf{E}_j, \text{ where } j = 1, \ldots, m-1, \quad \text{// loop}$$

$$\mathbf{B}_m^* = \mathbf{B}_{m-1} - c\delta t \nabla \times \mathbf{E}_m, \text{ and finally} \quad \text{// epilogue}$$

$$\mathbf{B}^{N+1} = (\mathbf{B}_m + \mathbf{B}_m^*)/2. \quad \text{// average}$$

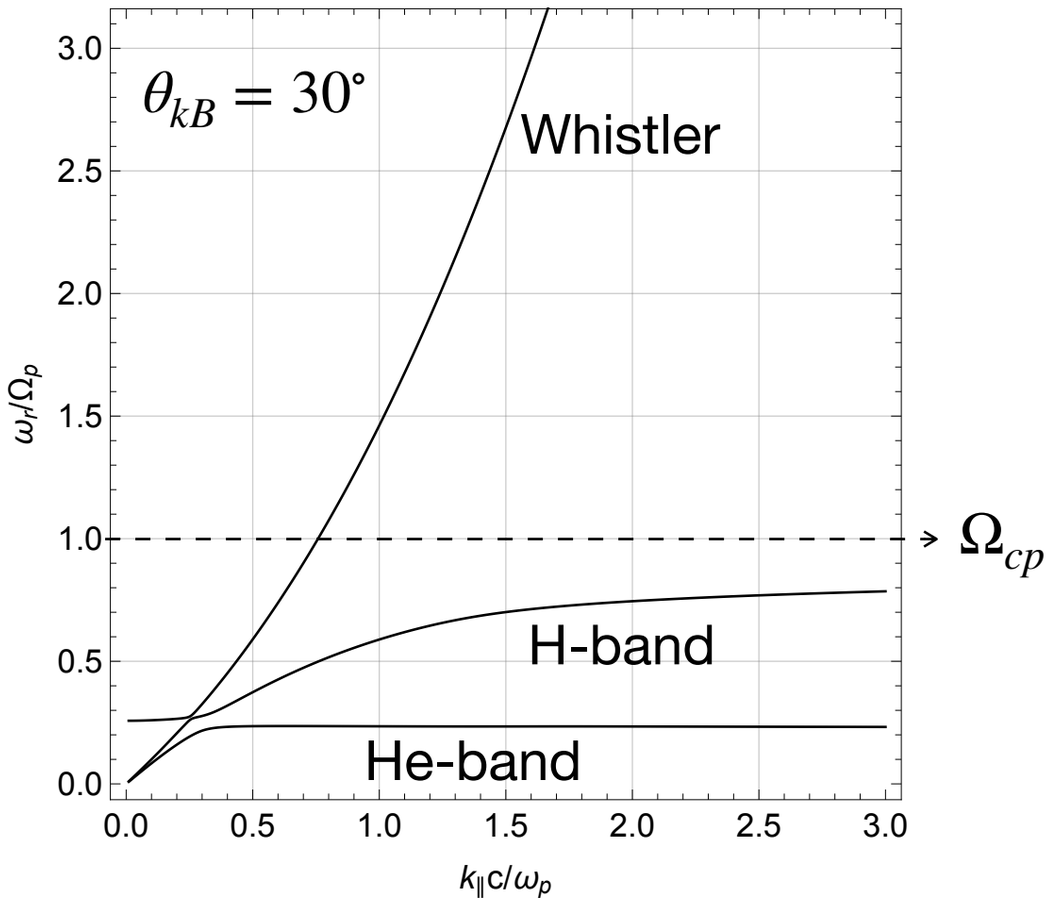
```
//  Domain_CAMCL.cc

void H1D::Domain_CAMCL::subcycle(Domain const &domain, Charge const &charge,
Current const &current, Real const _Dt)
{
    Real const dt = _Dt/m, dt_x_2 = dt*2.0;
    //
    // prologue
    //
    bfield_1 = bfield_0;
    efield.update(bfield_0, charge, current), delegate->pass(domain, efield);
    bfield_1.update(efield, dt), delegate->pass(domain, bfield_1);
    //
    // loop
    //
    for (long i = 1; i < m; ++i) {
        efield.update(bfield_1, charge, current), delegate->pass(domain, efield);
        bfield_0.update(efield, dt_x_2), delegate->pass(domain, bfield_0);
        bfield_0.swap(bfield_1);
    }
    //
    // epilogue
    //
    efield.update(bfield_1, charge, current), delegate->pass(domain, efield);
    bfield_0.update(efield, dt), delegate->pass(domain, bfield_0);
    //
    // average
    //
    (bfield_0 += bfield_1) *= Vector{.5};
}
```

Example 1: Low-frequency electromagnetic fluctuations in a relatively cool H+-He+ plasma

Input parameters under “`examples/1_cold_dispersion_theta-{0,30,60}`”

Low-frequency dispersion relation

- Input parameters under “examples/1_cold_dispersion_theta-{0,30,60}”
- $\omega_{pe}/|\Omega_{ce}| = 5$ and $\beta_e = 0.01$
- 99% protons with $\tilde{\beta}_p \equiv 8\pi n_e T_p / B_0^2 = 0.01 \rightarrow \beta_p = \frac{n_p}{n_e}\tilde{\beta}_p = 0.0099$
- 1% singly-ionized helium with $\tilde{\beta}_{\mathrm{He^+}} = 0.01 \rightarrow \beta_{\mathrm{He^+}} = \frac{n_{\mathrm{He^+}}}{n_e}\tilde{\beta}_{\mathrm{He^+}} = 0.0001$
- Wave normal angles θ_{kB} = 0, 30, and 60°
- Normalization:
  velocity by $v_{A0} = B_0/\sqrt{4\pi n_e m_p}$
  frequency by $\Omega_{cp} = eB_0/m_p c$
  time by Ω_{cp}^{-1}
  length by $\lambda_{p0} = v_{A0}/\Omega_{cp}$



Low-frequency dispersion relation

```
//  Inputs.h
/// light speed
///
constexpr Real c = 214.243;
```

$$= \frac{\omega_{pe}}{|\Omega_{ce}|}\sqrt{\frac{m_p}{m_e}}v_A = 5\sqrt{1836}v_A$$

```
/// angle in degrees between the x-axis and the uniform magnetic field direction.
///
constexpr Real theta = 30;
```

$\theta_{kB} = 0,\ 30,\ \text{or}\ 60^\circ$

```
/// simulation grid size
///
constexpr Real Dx = 0.2;
```

$k_{\max} = \pi/\Delta x \approx 15.7\lambda_p^{-1}$

```
/// number of grid points
///
constexpr unsigned Nx = 1440;
```

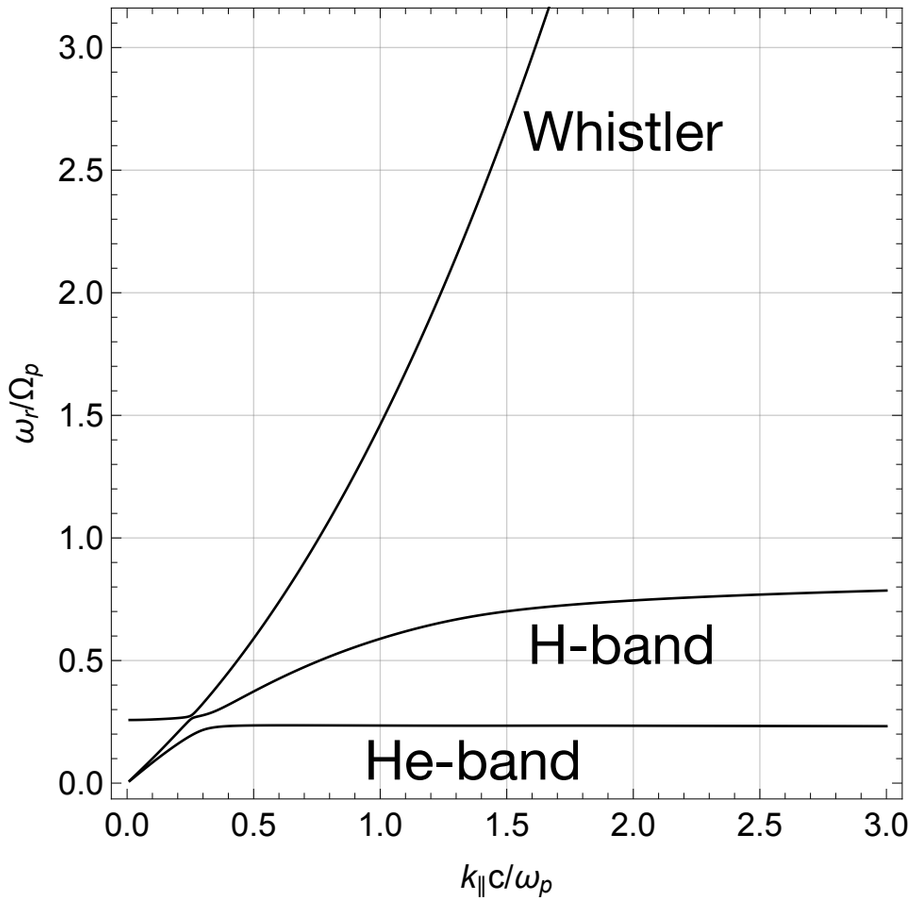
$k_{\min} = 2\pi/(N_x\Delta x) \approx 0.022\lambda_p^{-1}$

```
namespace eFluid {
    /// electron cyclotron frequency
    ///
    constexpr Real Oc = -1836;

    /// electron plasma frequency
    ///
    constexpr Real op = 9180;

    /// electron beta
    ///
    constexpr Real beta = 0.01;
}
```

$$= -\frac{m_p}{m_e}\Omega_{cp} \approx -1836\Omega_{cp}$$

$$= \frac{\omega_{pe}}{\Omega_{ce}}\Omega_{ce} = 5\times 1836\Omega_{cp}$$

$\theta_{kB} = 30^\circ$



Low-frequency dispersion relation

```
//  Inputs.h

namespace eFluid {
    /// electron cyclotron frequency
    ///
    constexpr Real Oc = -1836;

    /// electron plasma frequency
    ///
    constexpr Real op = 9180;

    /// electron beta
    ///
    constexpr Real beta = 0.01;
}

namespace iKinetic {
    /// number of ion species
    ///
    constexpr unsigned Ns = 2;

    /// number of simulation particles per cell for individual populations
    ///
    constexpr std::array<unsigned, Ns> Ncs = {500, 500};

    /// ion cyclotron frequencies for individual populations
    ///
    constexpr std::array<Real, Ns> Ocs = {1, .25};

    /// ion plasma frequencies for individual populations
    ///
    constexpr std::array<Real, Ns> ops = {213.169, 10.7122};

    /// parallel (w.r.t the background magnetic field direction)
    /// ion betas for individual populations
    ///
    constexpr std::array<Real, Ns> betas = {0.0099, 0.0001};

    /// ion temperature anisotropies (T_perp/T_para) for individual populations
    ///
    constexpr std::array<Real, Ns> T2OT1s = {1, 1};
}
```

$$= \frac{\omega_{pe}}{\Omega_{ce}}\Omega_{ce} = 5 \times 1836\Omega_{cp}$$

H+ and He+

$$\omega_{p0} \equiv \sqrt{\frac{4\pi e^2 n_e}{m_p}} = \frac{c}{v_{A0}}\Omega_{cp}$$

$$\omega_{pp} = \omega_{p0}\sqrt{0.99} \qquad \omega_{p\mathrm{He}^+} = \omega_{p0}\sqrt{0.01}/\sqrt{4}$$

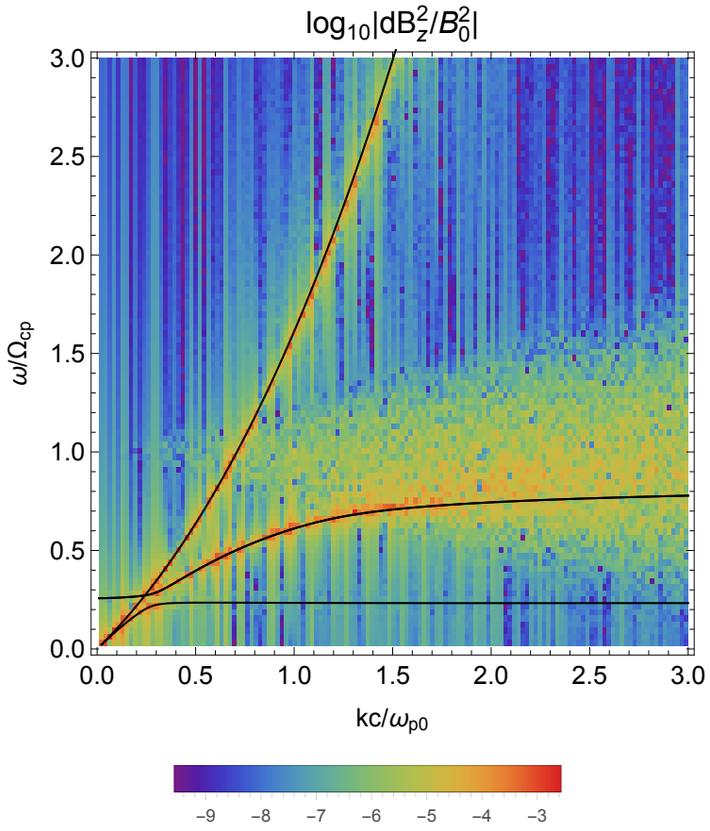
$$\{0.99\tilde{\beta}_p,\ 0.01\tilde{\beta}_{\mathrm{He}^+}\}$$

$$\tilde{\beta}_i \equiv 8\pi n_e T_i / B_0^2 = 0.01$$

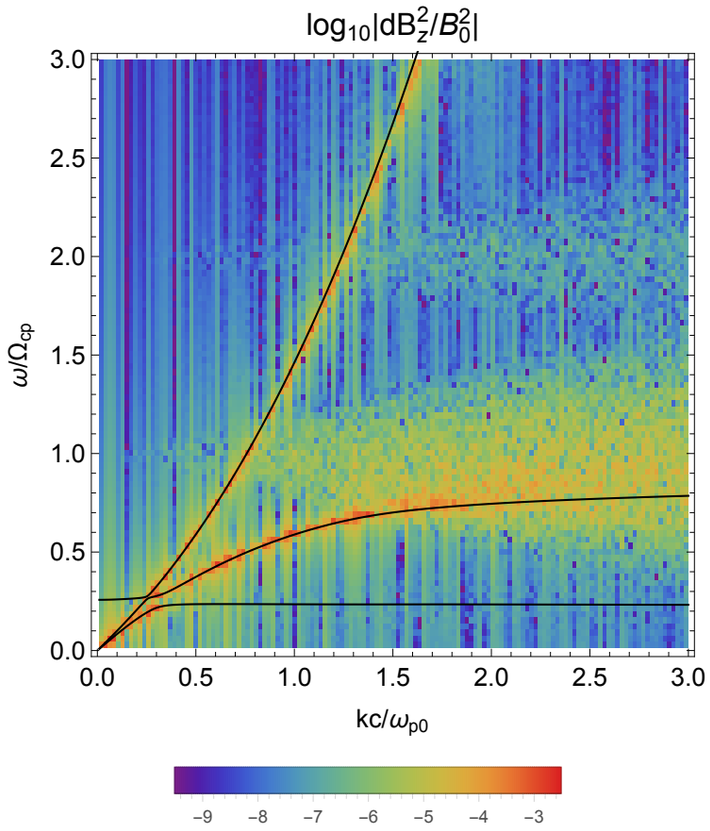
$$T_{\perp i}/T_{\parallel i}$$

Low-frequency dispersion relation

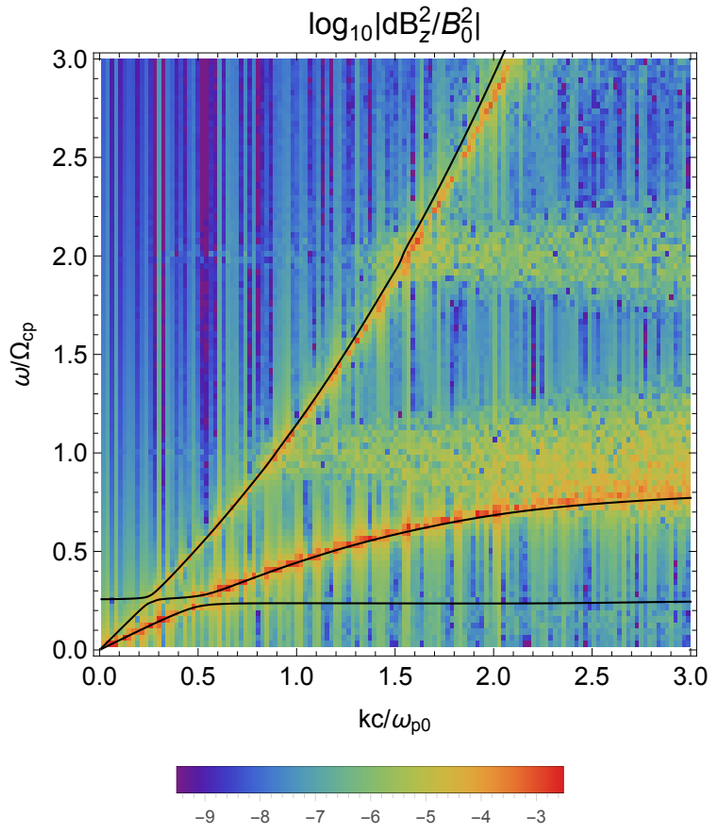
$\theta_{kB} = 0°$



$\theta_{kB} = 30°$



$\theta_{kB} = 60°$



Example 2: Temperature anisotropy-driven H+-band & He+-band EMIC waves relevant to Earth's Magnetosphere

Input parameters under “`examples/2_Fu_et_al_2016_EMIC_baseline_run`”

Fu et al. (2016) EMIC Study

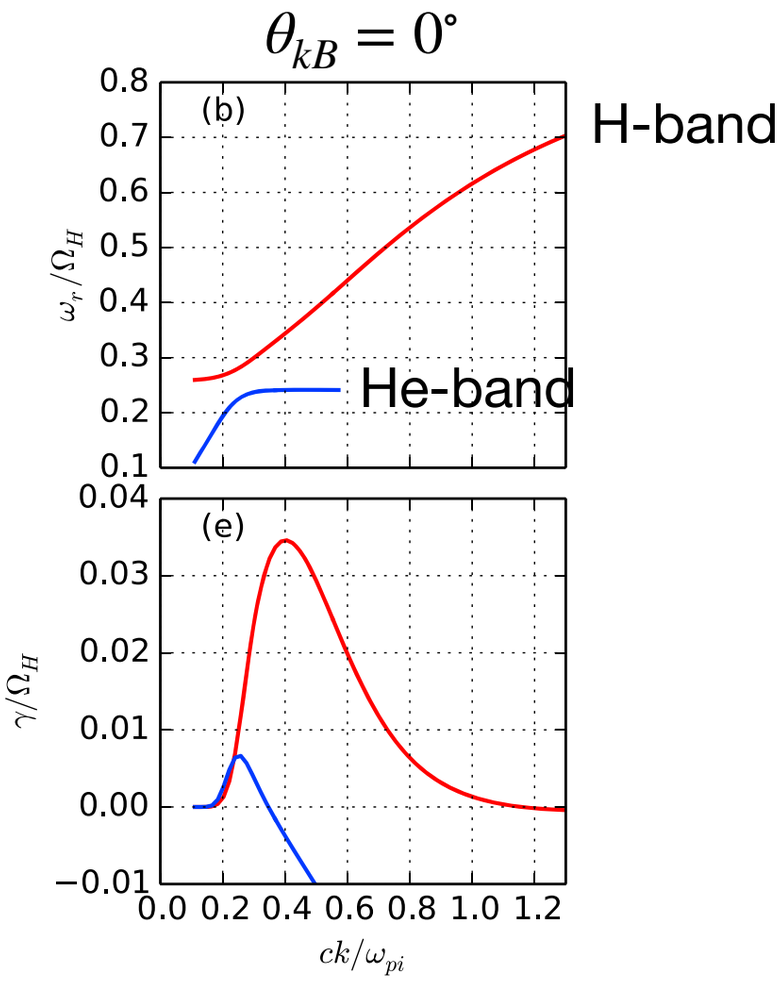
Predicting electromagnetic ion cyclotron wave amplitude from unstable ring current plasma conditions

Xiangrong Fu[1], Misa M. Cowee[2], Vania K. Jordanova[2], S. Peter Gary[3], Geoffrey D. Reeves[1,2], and Dan Winske[2]

[1]New Mexico Consortium, Los Alamos, New Mexico, USA, [2]Los Alamos National Laboratory, Los Alamos, New Mexico, USA, [3]Space Science Institute, Boulder, Colorado, USA

Table 1. Initial Plasma Parameters for EMIC Waves

Parameter	Values
ω_{pe}/ω_{ce}	5
$\tilde{\beta}_h = n_e T_h/(B_0^2/8\pi)$	0.3, 1.0, **3.0**, 10.0
n_h/n_e	**0.05**, 0.1, 0.2, 0.4
$T_{\perp h}/T_{\parallel h}$	2, **3**, 4, 5
n_{He}/n_e	0, **0.01**, 0.03, 0.05



- Fu, X., M. M. Cowee, V. K. Jordanova, S. P. Gary, G. D. Reeves, and D. Winske (2016), Predicting electromagnetic ion cyclotron wave amplitude from unstable ring current plasma conditions, J. Geophys. Res. Space Physics, 121, doi:10.1002/2016JA023303.

Fu et al. (2016) EMIC Study

Table 1. Initial Plasma Parameters for EMIC Waves

Parameter	Values
ω_{pe}/ω_{ce}	5
$\tilde{\beta}_h = n_e T_h/(B_0^2/8\pi)$	0.3, 1.0, 3.0, 10.0
n_h/n_e	0.05, 0.1, 0.2, 0.4
$T_{\perp h}/T_{\parallel h}$	2, 3, 4, 5
n_{He}/n_e	0, 0.01, 0.03, 0.05

```
namespace iKinetic {
    /// number of ion species
    ///
    constexpr unsigned Ns = 3;

    /// number of simulation particles per cell for individual populations
    ///
    constexpr std::array<unsigned, Ns> Ncs = {1000, 500, 500};

    /// ion cyclotron frequencies for individual populations
    ///
    constexpr std::array<Real, Ns> Ocs = {1, 1, .25};

    /// ion plasma frequencies for individual populations
    ///
    constexpr std::array<Real, Ns> ops = {47.9062, 207.716, 10.7122};

    /// ion betas for individual populations
    ///
    constexpr std::array<Real, Ns> betas = {0.15, 0.0094, 0.0001};

    /// ion temperature anisotropies (T_perp/T_para) for individual populations
    ///
    constexpr std::array<Real, Ns> T2OT1s = {3, 1, 1};
}
```

{5% hot H+, 94% cool H+, 1% He+}

Fu et al. (2016) EMIC Study

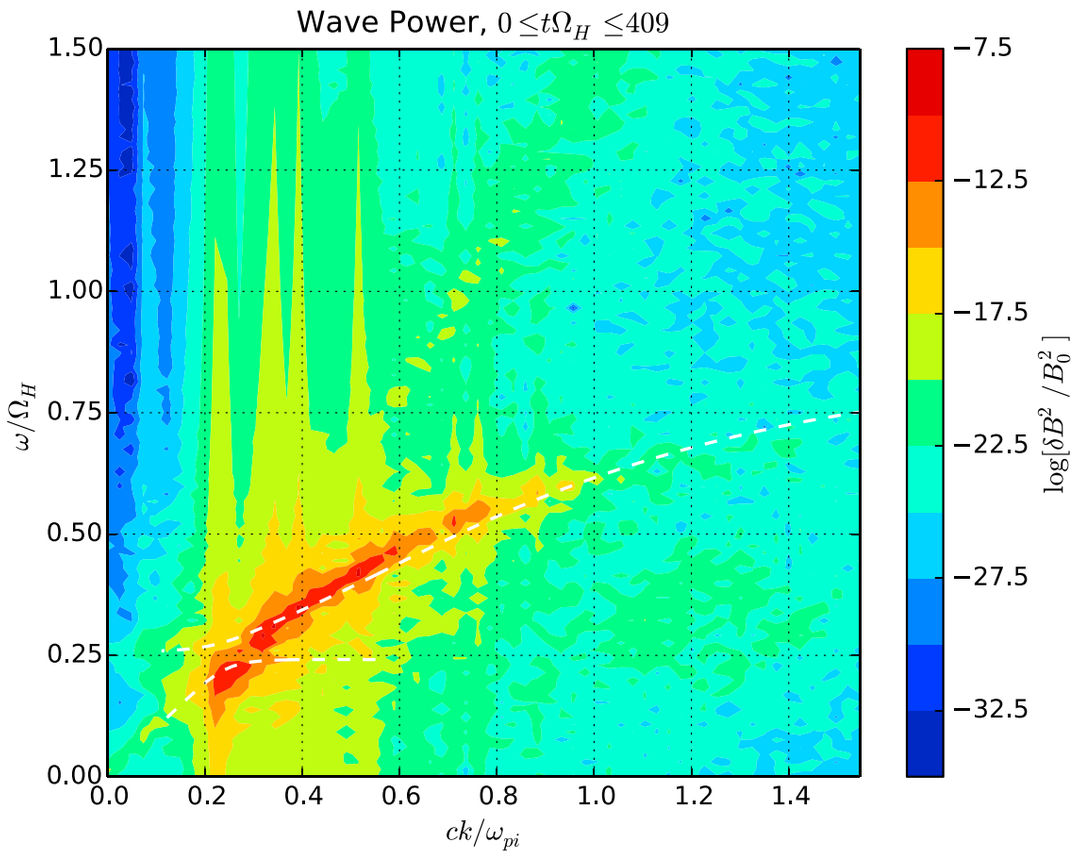
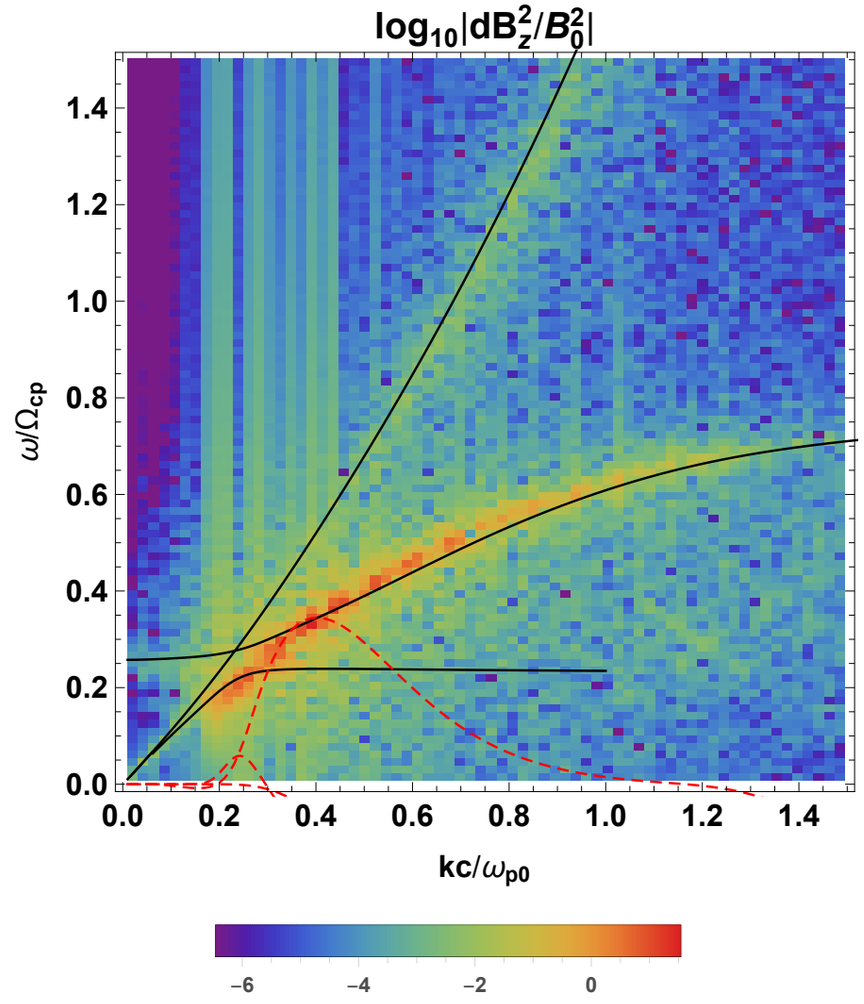


Figure 5 of Fu et al. (2016)

$$0 < t\Omega_{cp} < 400$$



Fu et al. (2016) EMIC Study

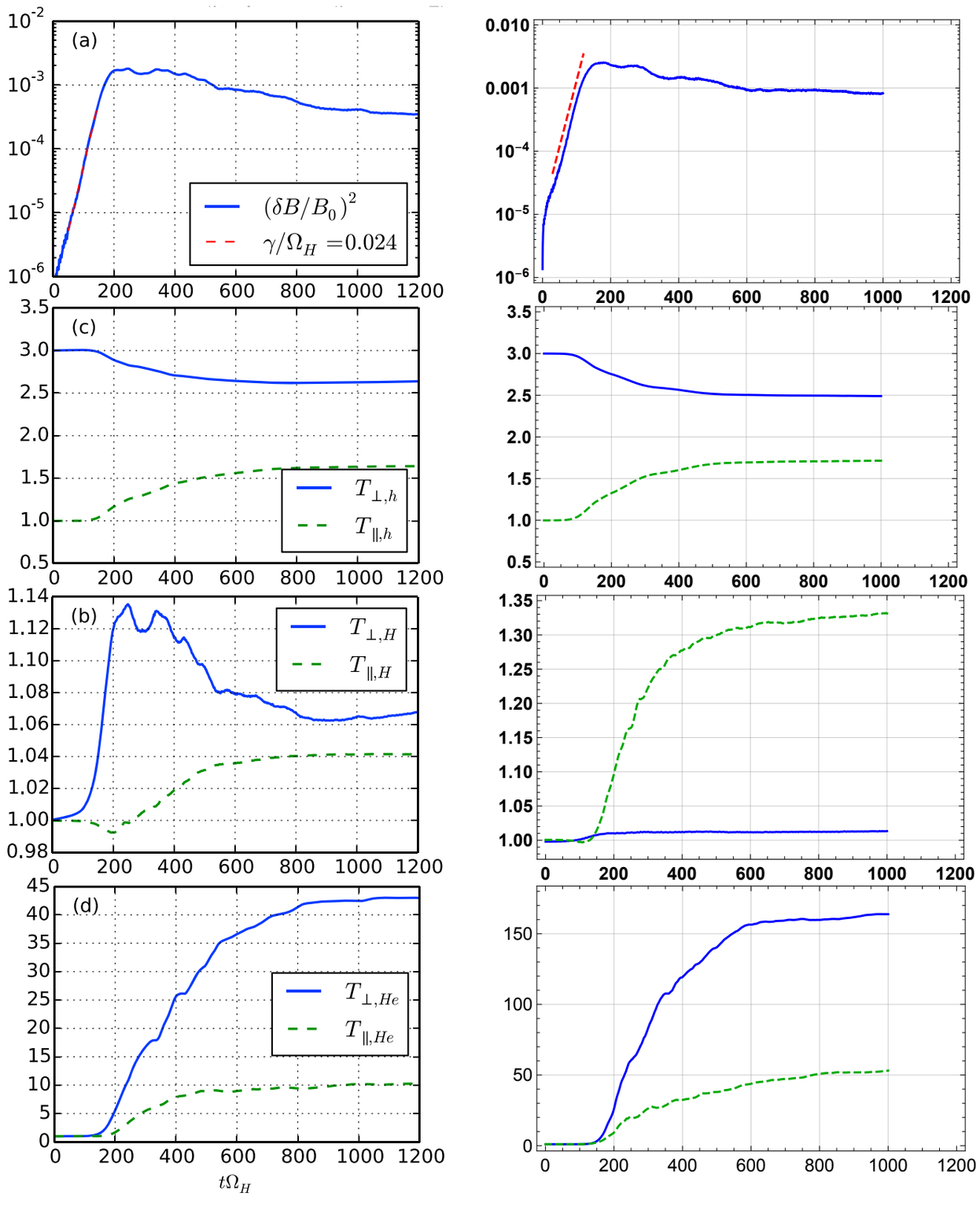


Figure 4 of Fu et al. (2016)

Example 3: Firehose Instability

Input parameters under “`examples/3_FHI`*”*

Firehose Instability

- $\omega_{pp}/\Omega_{cp} = 400$ and $\beta_e = 1$
- 100% protons with $\beta_{\|p} = 8\pi n_p T_{\|p}/B_0^2 = 3$ and $T_{\perp p}/T_{\|p} = 1/3$
- Wave normal angle $\theta_{kB} = 0^\circ$
- Normalization:
  velocity by $v_A = B_0/\sqrt{4\pi n_p m_p}$

  frequency by $\Omega_{cp} = eB_0/m_p c$

  time by Ω_{cp}^{-1}

  length by $\lambda_p = v_A/\Omega_{cp}$

Extra Example: O+ harmonic Bernstein modes

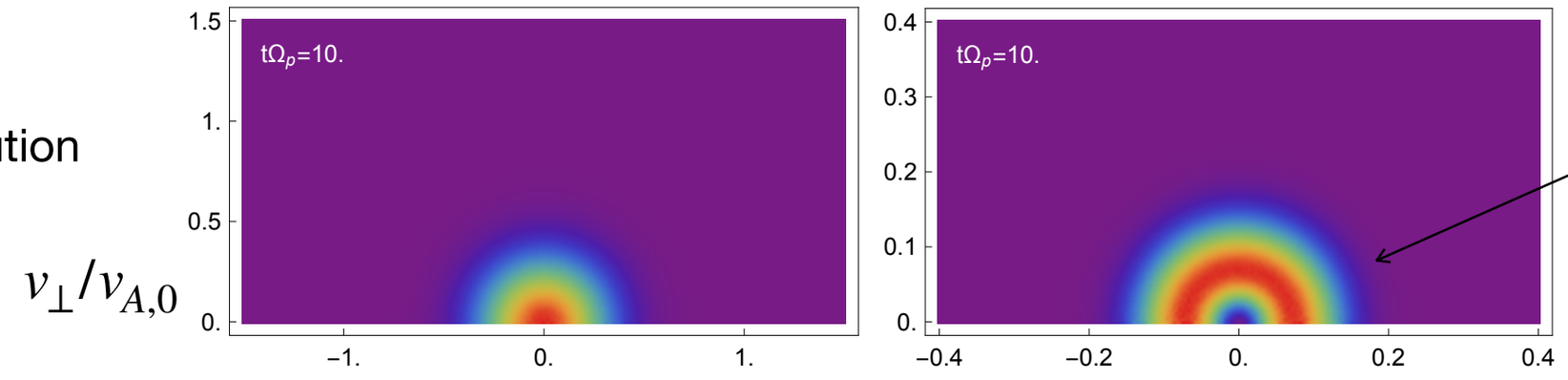
No corresponding example

$$\omega_{pe}/\Omega_{ce} = 1.2$$

50% H+

50% O+

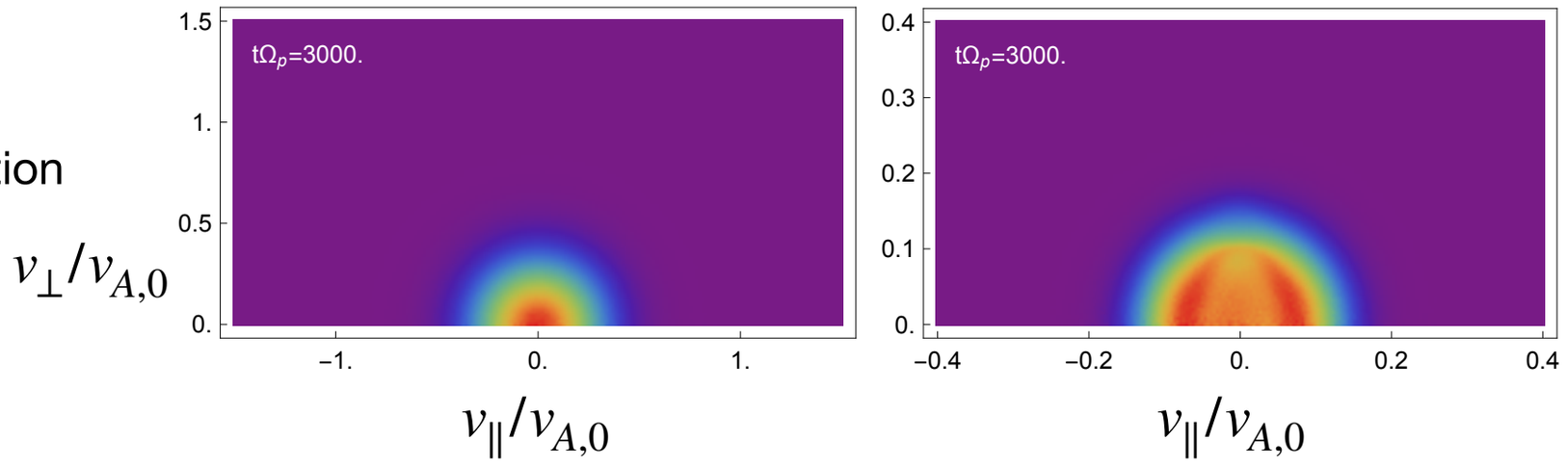
Initial Distribution



$v_\perp/v_{A,0}$

Subtracted Maxwellian

Final Distribution



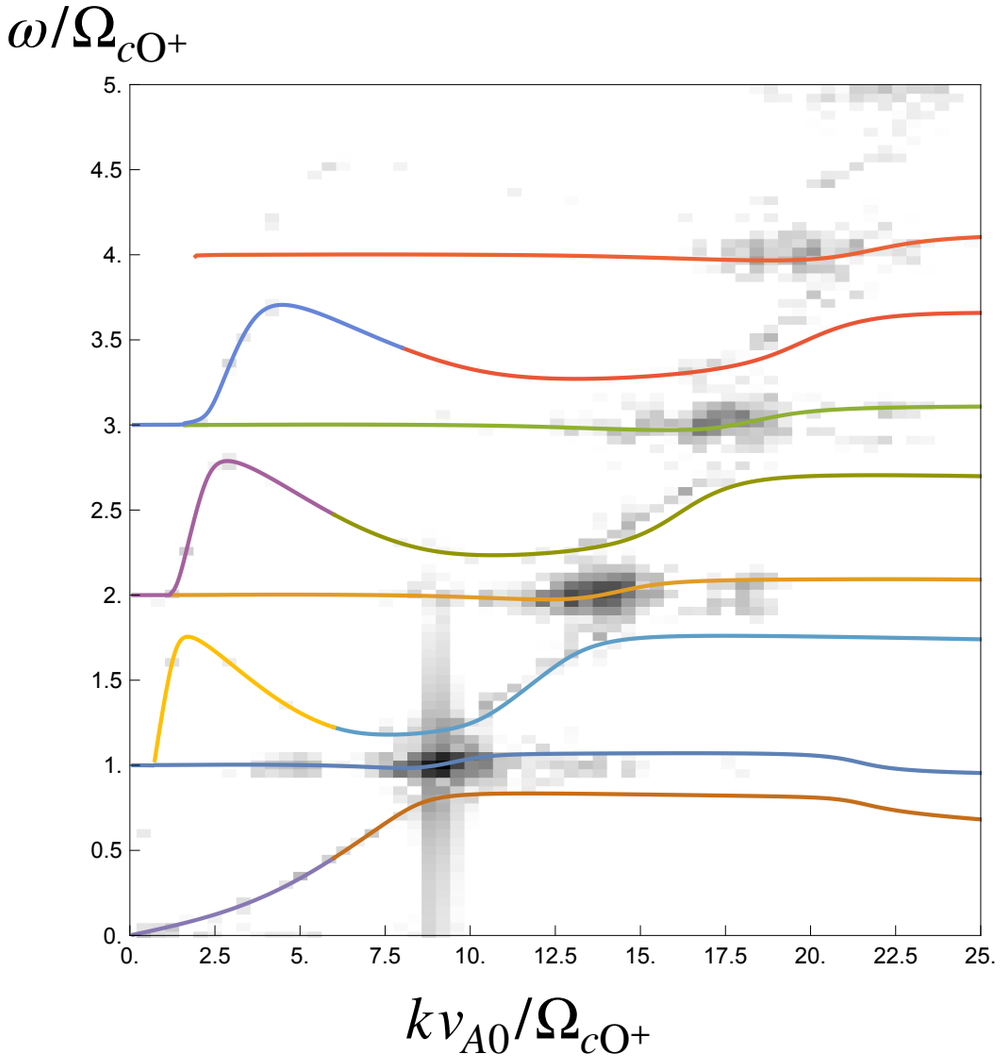
$v_\perp/v_{A,0}$

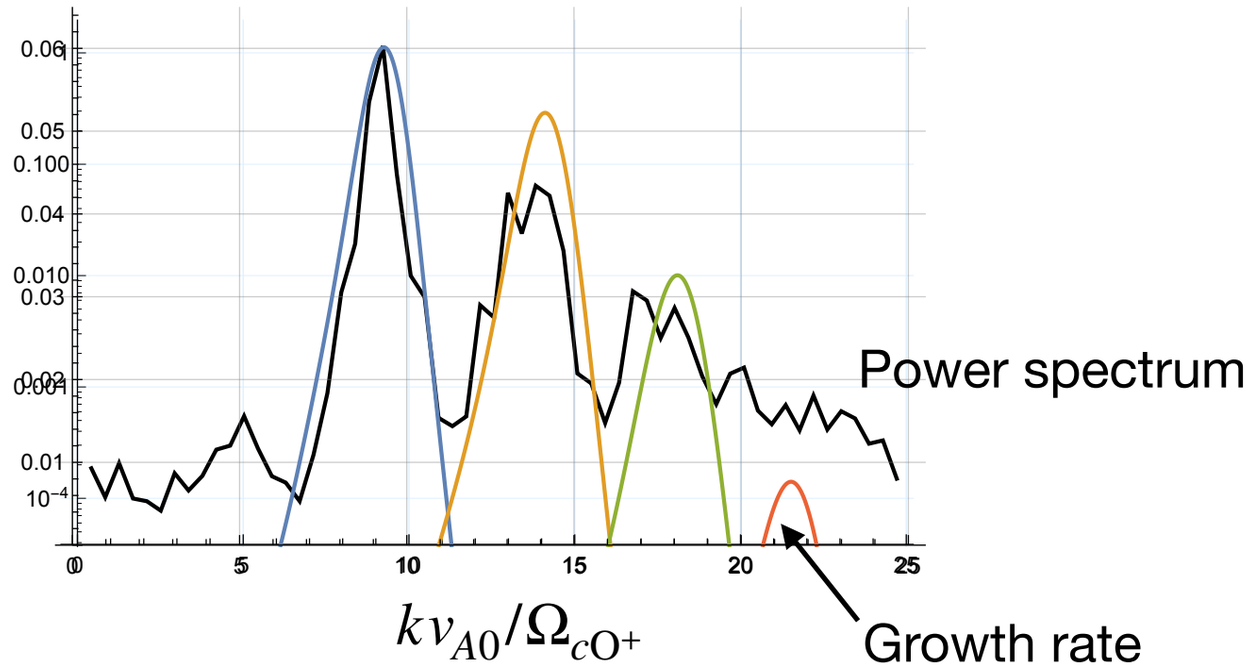
$v_\parallel/v_{A,0}$

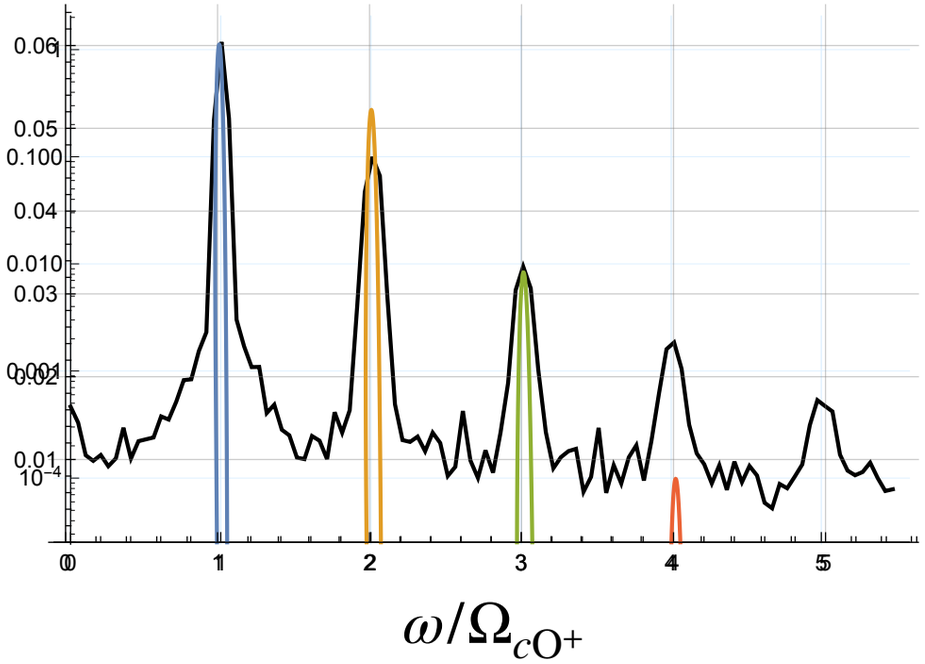
$v_\parallel/v_{A,0}$

$$v_{A,0} = B_0/\sqrt{4\pi n_e m_{\mathrm{O}^+}}$$

$\theta_{kB} = 88.2^{\circ}$



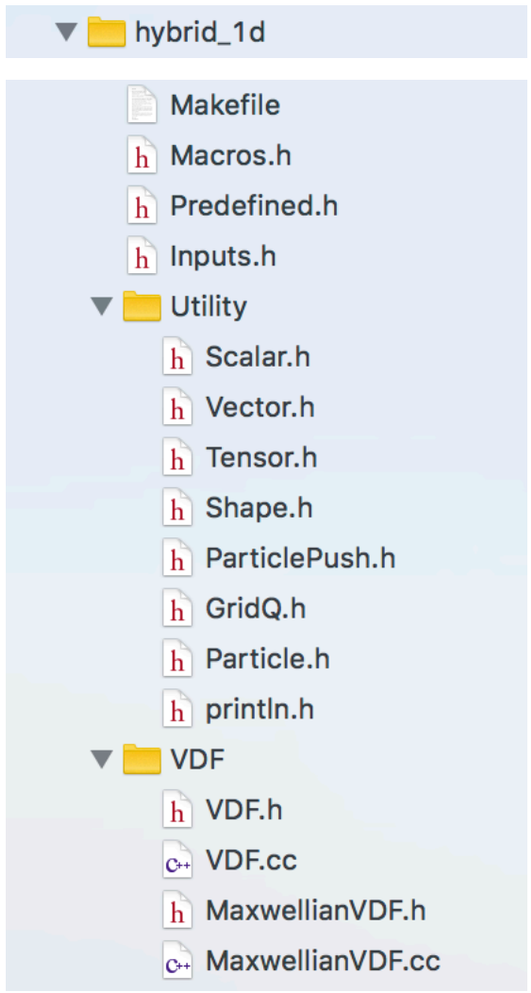


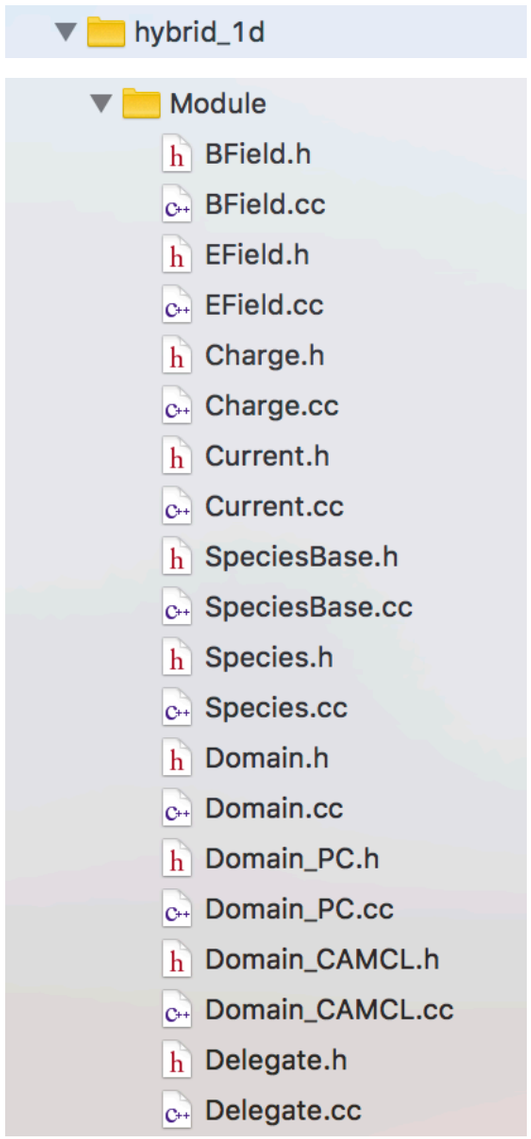


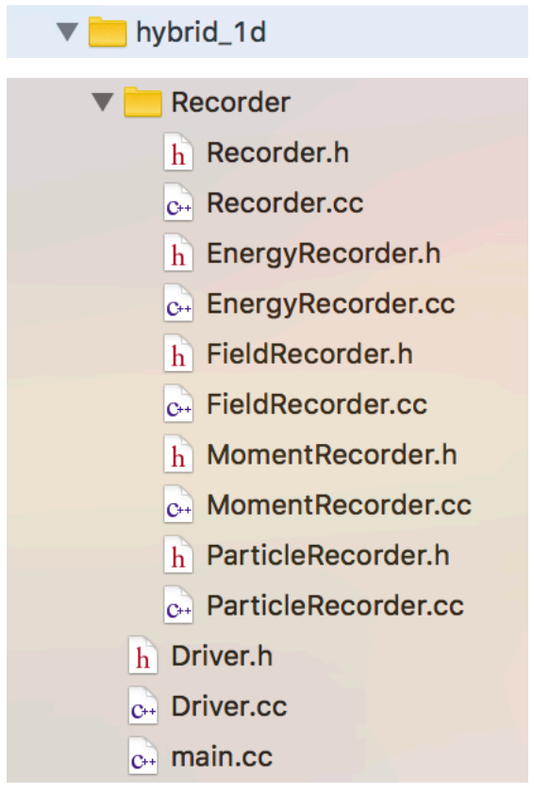
References

- Winske, D., L. Yin, N. Omidi, H. Karimabadi, and K. Quest (2003), Hybrid codes: Past, present and future—A tutorial, Lect. Notes Phys., 615, 136–265.
- Kunz, Matthew W.; Stone, James M.; Bai, Xue-Ning (2014), Pegasus: A new hybrid-kinetic particle-in-cell code for astrophysical plasma dynamics, Journal of Computational Physics, Volume 259, p. 154-174.
- Matthews, A. P. (1994). Current Advance Method and Cyclic Leapfrog for 2D Multispecies Hybrid Plasma Simulations. Journal of Computational Physics, 112, 102–116. https://doi.org/10.1006/jcph.1994.1084
- Liu, K. (2007), Particle-in-cell simulations of particle energization in the auroral region, PhD thesis, Cornell Univ., New York.
- Lipatov, Alexander S. (2002), *The Hybrid Multiscale Simulation Technology*, Springer-Verlag Berlin Heidelberg.
- Birdsall, C. K., and A. B. Langdon (1985), *Plasma Physics via Computer Simulation*, McGraw-Hill, New York.

Source Tree


hybrid_1d
Makefile
Macros.h
Predefined.h
Inputs.h
Utility
Scalar.h
Vector.h
Tensor.h
Shape.h
ParticlePush.h
GridQ.h
Particle.h
printIn.h
VDF
VDF.h
VDF.cc
MaxwellianVDF.h
MaxwellianVDF.cc


hybrid_1d
Module
BField.h
BField.cc
EField.h
EField.cc
Charge.h
Charge.cc
Current.h
Current.cc
SpeciesBase.h
SpeciesBase.cc
Species.h
Species.cc
Domain.h
Domain.cc
Domain_PC.h
Domain_PC.cc
Domain_CAMCL.h
Domain_CAMCL.cc
Delegate.h
Delegate.cc


hybrid_1d
Recorder
Recorder.h
Recorder.cc
EnergyRecorder.h
EnergyRecorder.cc
FieldRecorder.h
FieldRecorder.cc
MomentRecorder.h
MomentRecorder.cc
ParticleRecorder.h
ParticleRecorder.cc
Driver.h
Driver.cc
main.cc

Input Parameters: Inputs.h

```
//
// MARK:- Housekeeping
//

/// parallelize particle update
///
constexpr bool enable_concurrency = false;

/// electric field extrapolation method
///
constexpr _Algorithm algorithm = PC;

/// particle and interpolation order
///
constexpr _ShapeOrder shape_order = CIC;

/// number of subscyles for magnetic field update; applied only for CAM-CL algorithm
///
constexpr unsigned Nsubcycles = 4;

/// number of smoothings
///
constexpr unsigned Nsmooths = 2;
```

```
//  Predefined.h
```

```
enum _Algorithm : long {
        PC, //!< Using predictor-corrector by Kunz et al. (2014).
        CAMCL //!< Using CAM-CL by Matthew (1994).
    };
```

```
enum _ShapeOrder : long {
        CIC = 1, //!< First order; cloud-in-cell scheme.
        TSC = 2  //!< Second order; Triangular-shaped density cloud sheme.
    };
```

Input Parameters: Inputs.h (cont'd)

```
//
// MARK: Global parameters
//

/// light speed
///
constexpr Real c = 400;

/// magnitude of uniform background magnetic field
///
constexpr Real O0 = 1;

/// angle in degrees between the x-axis and the uniform magnetic field direction.
///
constexpr Real theta = 0;

/// simulation grid size
///
constexpr Real Dx = 0.4;

/// number of grid points
///
constexpr unsigned Nx = 960;

/// time step size
///
constexpr Real dt = 0.04;

/// number of time steps for inner loop
/// total time step Nt = inner_Nt * outer_Nt
/// simulation time t = dt*Nt
///
constexpr unsigned inner_Nt = 5;

/// number of time steps for outer loop
/// total time step Nt = inner_Nt * outer_Nt
/// simulation time t = dt*Nt
///
constexpr unsigned outer_Nt = 1000;
```

Outer loop

Inner loop

```
//  driver.cc

void H1D::Driver::operator()() const
{
    long step_count{};
    for (long i_step = 1; i_step <= Input::outer_Nt; ++i_step) {

        // advance simulation
        //
        domain->advance_by(Input::inner_Nt);

        // update step count
        //
        long const steps = step_count += Input::inner_Nt;

        // record data
        //
        for (auto &pair : recorders) {
            if (pair.second) {
                pair.second->record(*domain, steps);
            }
        }
    }
}
```

Input Parameters: Inputs.h (cont'd)

```
//
// MARK: Fluid Electrons
//
namespace eFluid {
    /// electron cyclotron frequency
    ///
    constexpr Real Oc = -1836;

    /// electron plasma frequency
    ///
    constexpr Real op = 17139.4;

    /// electron beta
    ///
    constexpr Real beta = 0.0;

    /// specific heat ratio, gamma
    ///
    constexpr _Closure closure = isothermal;
}
```

```
//  Predefined.h
```

```
enum _Closure : long {
        isothermal = 11, //!< gamma = 1/1
        adiabatic = 53 //!< gamma = 5/3
    };
```

Input Parameters: Inputs.h (cont'd)

```
//
// MARK: Kinetic Ions
//
namespace iKinetic {
    /// number of ion species
    ///
    constexpr unsigned Ns = 2;

    /// number of simulation particles per cell for individual populations
    ///
    constexpr std::array<unsigned, Ns> Ncs = {100, 100};

    /// ion cyclotron frequencies for individual populations
    ///
    constexpr std::array<Real, Ns> Ocs = {1, 1};

    /// ion plasma frequencies for individual populations
    ///
    constexpr std::array<Real, Ns> ops = {282.843, 282.843};

    /// ion betas for individual populations
    ///
    constexpr std::array<Real, Ns> betas = {.5, .5};

    /// ion temperature anisotropies (T_perp/T_para) for individual populations
    ///
    constexpr std::array<Real, Ns> T2OT1s = {3, 3};
}
```

Input Parameters: Inputs.h (cont'd)

```
//
// MARK: Data Recording
//

/// a top-level directory to which outputs will be saved
///
constexpr char working_directory[] = ".";

/// frequency of field and particle energy density recordings; in units of inner_Nt
/// `0' means `not interested'
///
constexpr unsigned energy_recording_frequency = 1;

/// frequency of electric and magnetic field recordings
///
constexpr unsigned field_recording_frequency = 3;

/// frequency of kinetic ion moment recordings
///
constexpr unsigned moment_recording_frequency = 5;

/// frequency of simulation particle recordings
///
constexpr unsigned particle_recording_frequency = 10000;
```