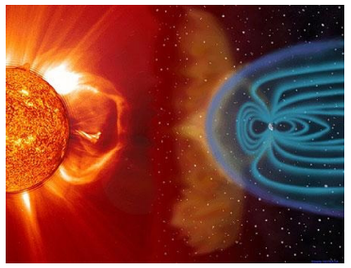
The Gkeyll simulation framework for plasma simulations (at all scales!)

Ammar Hakim†, Gregory Hammett†, Manaure Francisquez†,
Liang Wang†, Rupak Mukherjee†, Jason TenBarge†,
Jimmy JunoΔ, Noah Mandell∞, Petr Cagas§, Tess Bernardº,
and more valuable users + developers

† Princeton Plasma Physics Laboratory
∞ MIT
§ Virginia Tech
Δ University of Iowa
º General Atomics

The magnitude of scale separation in plasma physics

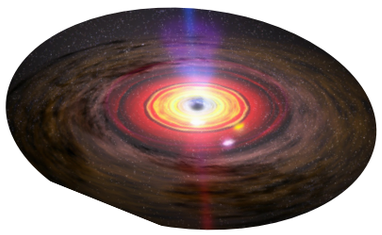


ℓ

ρ_e ρ_i λ_{mfp}

λ_D d_e d_i L

10^{-10} 10^{-6} 10^{-2} 1 au

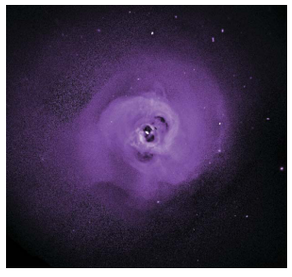


ℓ

ρ_e ρ_i λ_{mfp}

λ_D d_e d_i L

10^{-14} 10^{-11} 0.1 pc



ℓ

ρ_e ρ_i λ_{mfp}

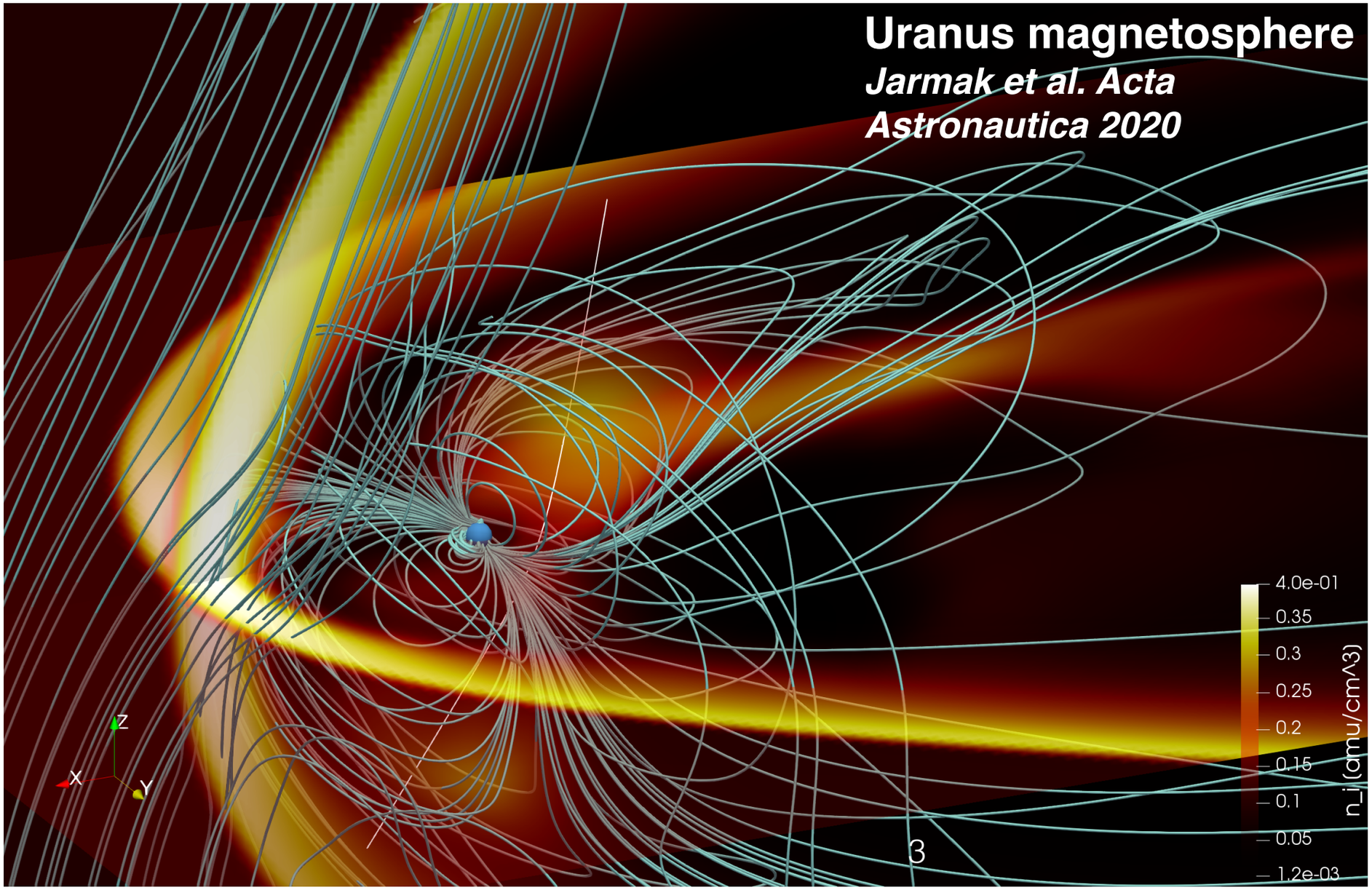
λ_D d_e d_i L

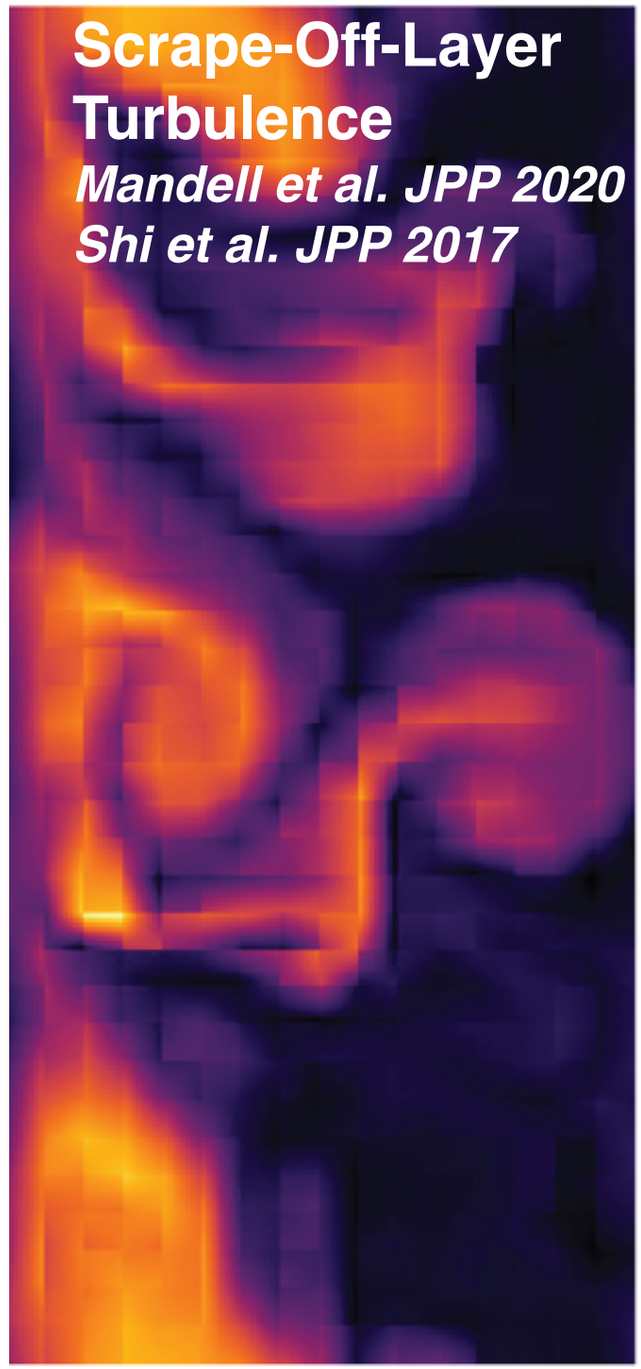
10^{-18} 10^{-14} $1\ 10^{2}$ kpc

So what is Gkeyll?

- Gkeyll is a general-purpose open-source simulation framework leveraging common infrastructure for a variety of equation systems of relevance in plasma physics
- Two pieces: **gkyl** (simulations) and **postgkyl** (post-processing)
  - Gkyl written in a combination of LuaJIT and C/C++
  - Postgkyl written in Python

https://github.com/ammarhakim/gkyl.git. https://gkyl.readthedocs.io/





The models within Gkeyll

So what are the options available?

Multi-fluid multi-moment models (Wave Propagation finite volume):

$$\frac{\partial(m_s n_s)}{\partial t} + \frac{\partial(m_s n_s u_{s,i})}{\partial x_i} = 0$$

$$\frac{\partial(m_s n_s u_{s,i})}{\partial t} + \frac{\partial \mathcal{P}_{s,ij}}{\partial x_j} = n_s q_s \left(E_i + \epsilon_{ijk} u_{s,j} B_k\right)$$

5-moment (Euler)

$$\frac{\partial \mathcal{E}_s}{\partial t} + \frac{\partial}{\partial x_i}(p_s + \mathcal{E}_s) u_{s,i} = n_s q_s u_{s,i} E_i$$

10-moment

$$\frac{\partial \mathcal{P}_{s,ij}}{\partial t} + \frac{\partial \mathcal{Q}_{s,ijk}}{\partial x_k} = n_s q_s u_{[s,i} E_{j]} + \frac{q_s}{m_s} \epsilon_{[ikl} \mathcal{P}_{s,kj]} B_l$$

Maxwell's induction equations

$$\epsilon_0 \mu_0 \frac{\partial \mathbf{E}}{\partial t} - \nabla \times \mathbf{B} = -\mu_0 \mathbf{j}$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \times \mathbf{E} = 0$$

Vlasov-Maxwell model (Runge-Kutta + discontinuous Galerkin):

$$\frac{\partial f_s}{\partial t} + \nabla \cdot \mathbf{v} f_s + \nabla_{\mathbf{v}} \cdot \frac{q_s}{m_s} \left[\mathbf{E} + \mathbf{E}_{\text{ext}} + \mathbf{v} \times (\mathbf{B} + \mathbf{B}_{\text{ext}})\right] = C[f_s] + S_s$$

Gyrokinetic model (Runge-Kutta + discontinuous Galerkin):

$$\frac{\partial(\mathcal{J} f_s)}{\partial t} + \nabla \cdot \mathcal{J} \dot{\mathbf{R}} f_s + \frac{\partial}{\partial v_\parallel}\left(\dot{v}_\parallel^H - \frac{q_s}{m_s}\frac{\partial A_\parallel}{\partial t}\right)\mathcal{J} f_s = \mathcal{J} C[f_s] + \mathcal{J} S_s$$

$$-\nabla \cdot \epsilon_\perp \nabla_\perp \phi = \sum_s q_s n_s$$

$$-\nabla_\perp^2 A_\parallel = \mu_0 \sum_s q_s u_{\parallel s}$$

https://github.com/ammarhakim/gkyl.git. https://gkyl.readthedocs.io/

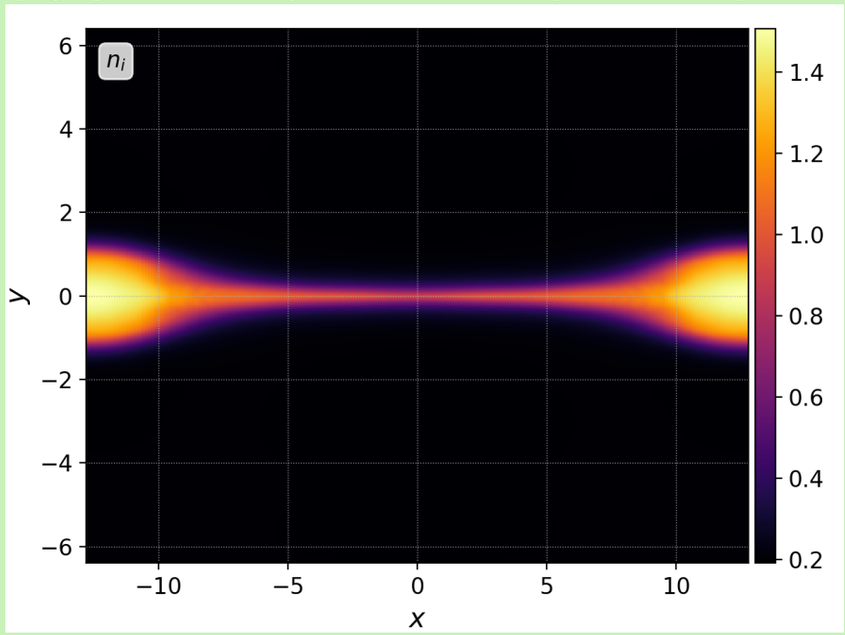
Flexible, common infrastructure for post-processing too!

Gkeyll's post-processing tool, **postgkyl**, can view and analyze fluid & kinetic data:

Petr Cagas, VIRGINIA TECH

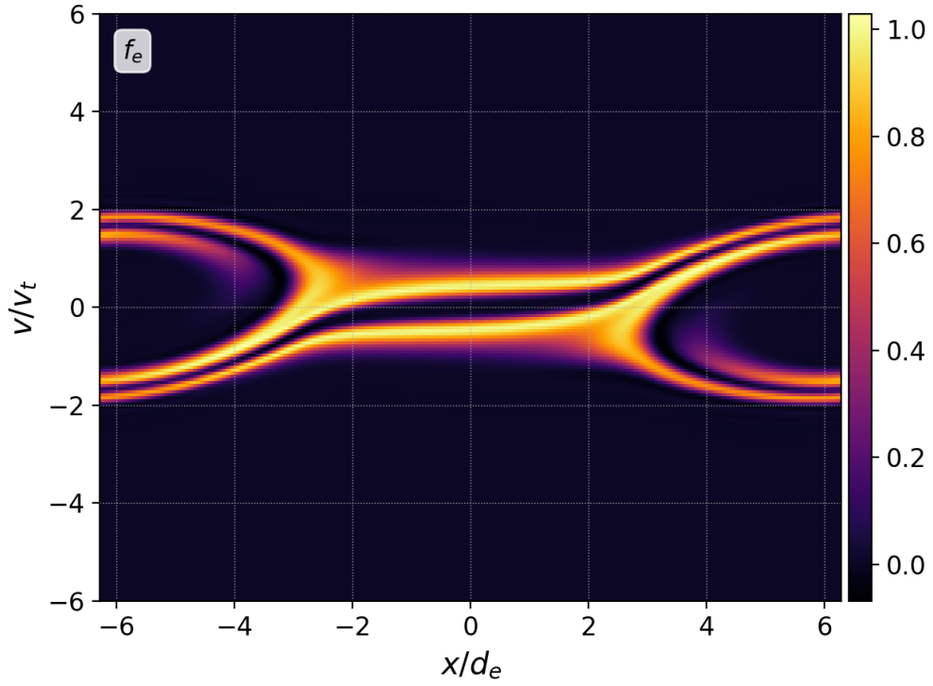
5-moment GEM reconnection

pgkyl mom5-gem_ion_4.bp sel -c0 pl

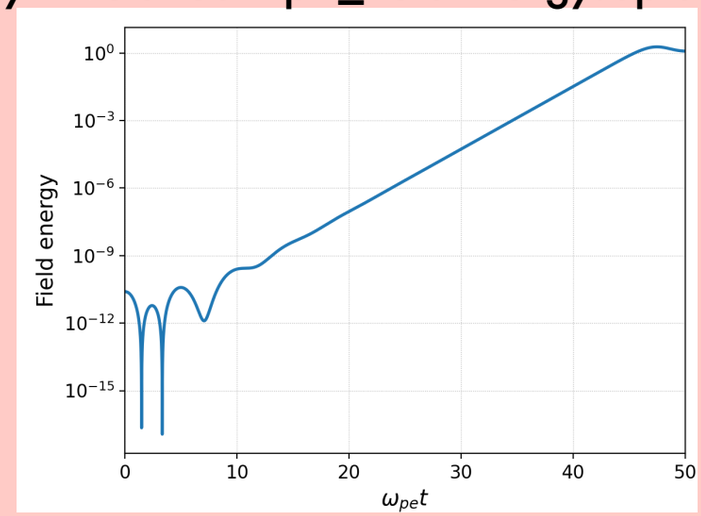


Vlasov-Maxwell two-stream instability

pgkyl two-stream-p2_elc_50.bp interp plot



pgkyl two-stream-p2_fieldEnergy.bp sel -c0 plot



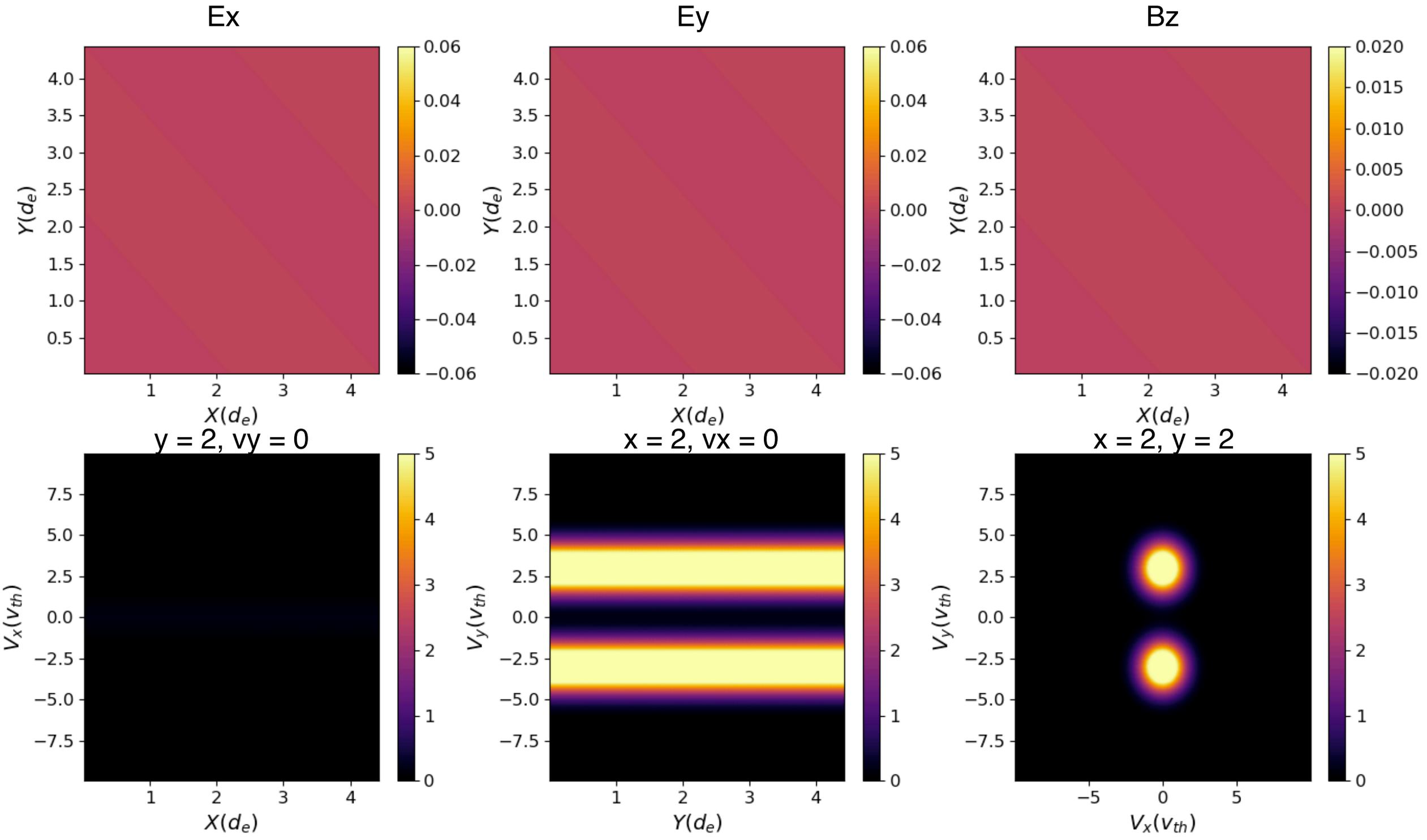
$\gamma = 0.1069$

pgkyl two-stream-p2_fieldEnergy.bp sel -c0 **growth**

https://github.com/ammarhakim/postgkyl.git. https://gkyl.readthedocs.io/

An example Gkeyll (Vlasov-Maxwell) simulation

- Instability calculation: hybrid two-stream & Weibel instability



Gkeyll 1.0: Making the user do all the work

- Because Gkeyll allows for multiple equations systems that are all using the same infrastructure (looping, I/O, parallel decomposition) input files can be complex

$$\frac{\partial f}{\partial t} = \mathcal{L}(f, \dots) \quad \rightarrow$$

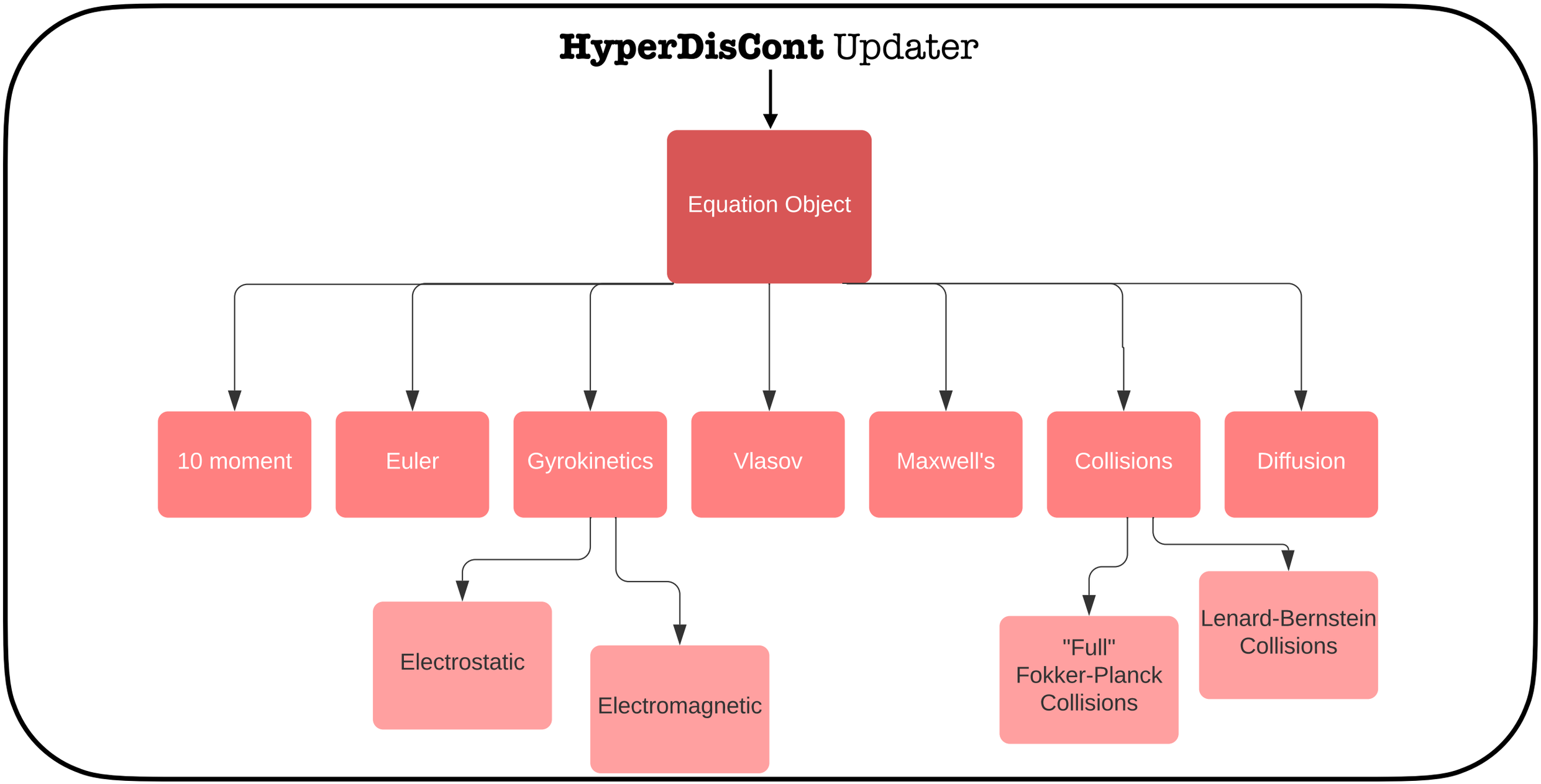
$$f^{n+1} = f^n + \Delta t \mathcal{L}(f^n, \dots)$$

```
216  -----------------------------------------------------------
217  -- Updaters --
218  -----------------------------------------------------------
219  VlasovCalc = EqVlasov {
220     onGrid = phaseGrid,
221     phaseBasis = phaseBasis,
222     confBasis = confBasis,
223     charge = chargeElc,
224     mass = massElc,
225     hasElectricField = true,
226     hasMagneticField = true,
227  }
228  -- Specify the directions for zero-flux directions in velocity space.
229  local zfd = { }
230  for d = 1, vdim do zfd[d] = cdim+d end
231
232  MaxwellCalc = EqMaxwell {
233     lightSpeed = 1.0,
234     elcErrorSpeedFactor = 0.0,
235     mgnErrorSpeedFactor = 0.0,
236     basis = confBasis,
237  }
238
239  VlasovSlvr = Updater.HyperDisCont {
240     onGrid = phaseGrid,
241     basis = phaseBasis,
242     cfl = cflNum,
243     equation = VlasovCalc,
244     zeroFluxDirections = zfd,
245  }
246
247  MaxwellSlvr = Updater.HyperDisCont {
248     onGrid = confGrid,
249     basis = confBasis,
250     cfl = cflNum,
251     equation = MaxwellCalc,
252  }
253
```

```
313  -----------------------------------------------------------
314  -- Helper functions for main simulation --
315  -----------------------------------------------------------
316  -- Function to combine and accumulate forward Euler time-step.
317  local function combine(outIdx, a, aIdx, ...)
318     local args = {...} -- Package up rest of args as table.
319     local nFlds = #args/2
320     outIdx:combine(a, aIdx)
321     for i = 1, nFlds do -- Accumulate rest of the fields.
322        outIdx:accumulate(args[2*i-1], args[2*i])
323     end
324  end
325
326  -- Function to take a single forward-euler time-step.
327  local function forwardEuler(tCurr, dt, fIn, emIn, fOut, emOut)
328     -- Clear CFL before taking time-step.
329     cflRateByCell:clear(0.0)
330     cflRateByCellMaxwell:clear(0.0)
331     -- Clear totalEmField and accumulate charge/mass electromagnetic field.
332     totalEmField:clear(0.0)
333     local qbym = chargeElc/massElc
334     totalEmField:accumulate(qbym, emIn)
335
336     -- Compute momentum from distribution function.
337     momDensityCalc:advance(0.0, {fIn}, {momDens})
338
339     -- Compute RHS of Vlasov equation.
340     VlasovSlvr:setDtAndCflRate(dtGlobal[0], cflRateByCell)
341     VlasovSlvr:advance(tCurr, {fIn, totalEmField}, {fOut})
342
343     -- Compute RHS of Maxwell's equations.
344     MaxwellSlvr:setDtAndCflRate(dtGlobalMaxwell[0], cflRateByCellMaxwell)
345     MaxwellSlvr:advance(tCurr, {emIn}, {emOut})
346
347     -- Accumulate current onto RHS of electric field.
348     -- Third input is the component offset.
349     -- The offset is zero for accumulating the current onto the electric field.
350     emOut:accumulateOffset(-chargeElc/epsilon0, momDens, 0)
351
352     -- Get the size of the time-step (only do this for the first RK step)
353     local dtSuggested = tEnd - tCurr + 1e-20
354     if dt == nil then
355        dtSuggested = math.min(cflNum/cflRateByCell:reduce('max')[1], dtSuggested)
356        dtSuggested = math.min(cflNum/cflRateByCellMaxwell:reduce('max')[1], dtSuggested)
357        dtGlobal[0] = dtSuggested
358     else
359        dtSuggested = dt
360     end
361
362     -- Increment solution f^{n+1} = f^n + dtSuggested*fRHS.
363     combine(fOut, dtSuggested, fOut, 1.0, fIn)
364     combine(emOut, dtSuggested, emOut, 1.0, emIn)
365     -- Synchronize ghost cells across MPI processes.
366     fOut:sync()
367     emOut:sync()
368
369     return dtSuggested
370  end
371
372  -- Function to advance solution using SSP-RK3 scheme.
373  local function rk3(tCurr)
374     -- RK stage 1
375     local dt = forwardEuler(tCurr, nil, distf, em, distf1, em1)
376
377     -- RK stage 2
378     forwardEuler(tCurr+dt, dt, distf1, em1, distfNew, emNew)
379     distf1:combine(3.0/4.0, distf, 1.0/4.0, distfNew)
380     em1:combine(3.0/4.0, em, 1.0/4.0, emNew)
381
382     -- RK stage 3
383     forwardEuler(tCurr+dt/2, dt, distf1, em1, distfNew, emNew)
384     distf1:combine(1.0/3.0, distf, 2.0/3.0, distfNew)
385     distf:copy(distf1)
386     em1:combine(1.0/3.0, em, 2.0/3.0, emNew)
387     em:copy(em1)
388
389     -- return size of the time-step so we know what time we have evolved to
390     return dt
391  end
```

Useful (but burdensome) flexibility

- Gkeyll does not know *a priori* how complex your equation system is.
  - Maybe you want to run with three species or include collisions or ionization
  - What form does the system of equations take? How is the system coupled?



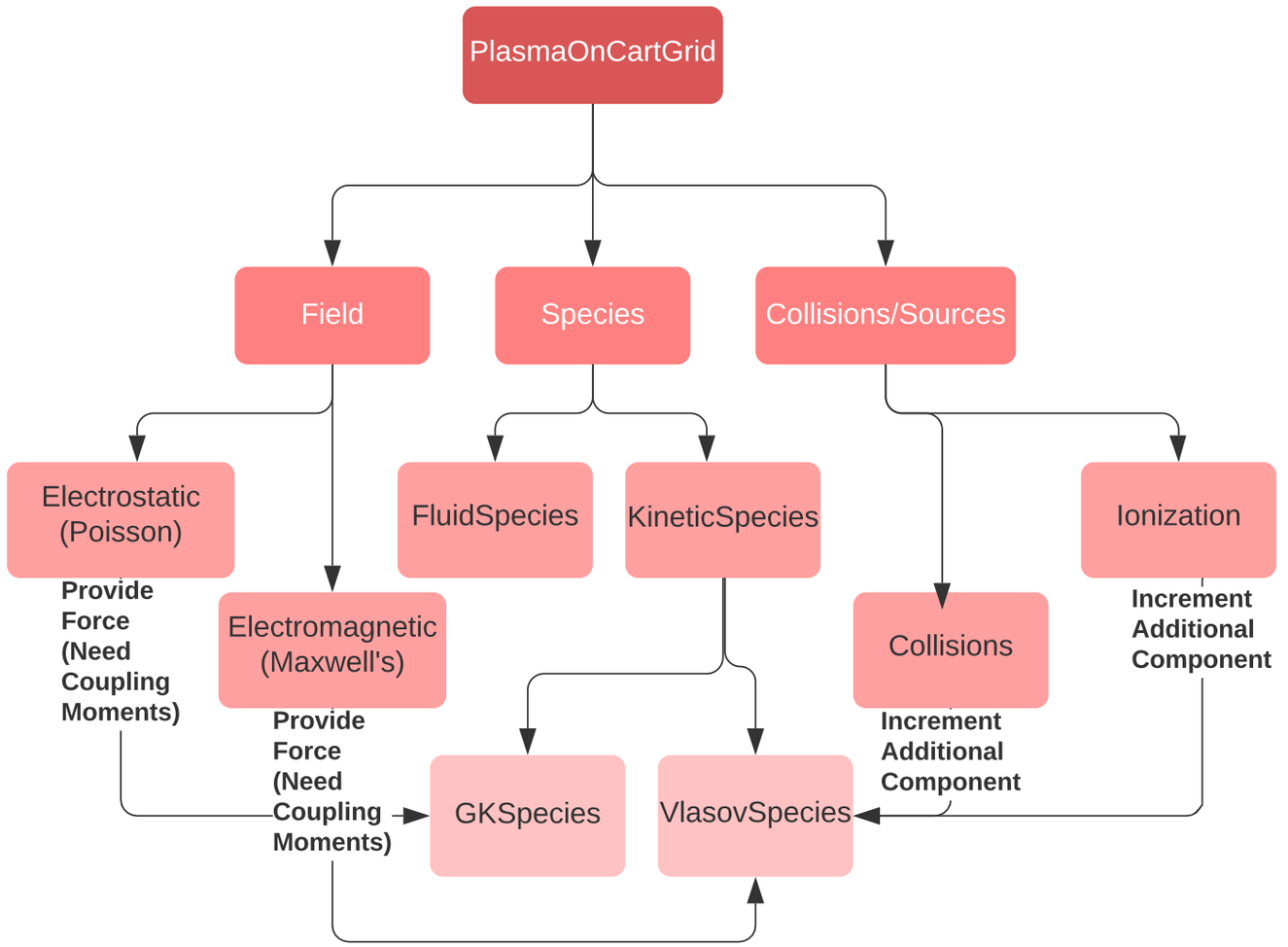
Gkeyll 2.0: The App System

```
local function maxwellian2D(n, vx, vy, ux, uy, vth)
   local v2 = (vx - ux)^2 + (vy - uy)^2
   return n/(2*math.pi*vth^2)*math.exp(-v2/(2*vth^2))
end

vlasovApp = Plasma.App {
   ----------------------------------------------------------------------------
   -- Common
   ----------------------------------------------------------------------------
   logToFile = true,
   tEnd = tEnd,              -- End time
   nFrame = nFrame,          -- Number of output frames
   lower = {0.0,0.0},        -- Lower boundary of configuration space
   upper = {Lx,Ly},          -- Upper boundary of configuration space
   cells = {Nx,Ny},          -- Configuration space cells
   basis = "serendipity",    -- One of "serendipity", "maximal-order", or "tensor"
   polyOrder = polyOrder,    -- Polynomial order
   timeStepper = "rk3",      -- One of "rk2", "rk3", or "rk3s4"
   -- MPI decomposition for configuration space
   decompCuts = {Ncx,Ncy}, -- Cuts in each configuration direction
   useShared = false,        -- If using shared memory
   -- Boundary conditions for configuration space
   periodicDirs = {1,2},     -- periodic directions (both x and y)
   ----------------------------------------------------------------------------
   -- Electrons
   ----------------------------------------------------------------------------
   elc = Plasma.Species {
      charge = chargeElc, mass = massElc,
      -- Velocity space grid
      lower = {-Vmax, -Vmax},
      upper = {Vmax, Vmax},
      cells = {Nvx, Nvy},
      -- Initial conditions
      init = function (t, xn)
         local x, y, vx, vy = xn[1], xn[2], xn[3], xn[4]
         local fv = maxwellian2D(nElc10, vx, vy, uxElc10, uyElc10, vthElc10) +
            maxwellian2D(nElc20, vx, vy, uxElc20, uyElc20, vthElc20)
         return (1.0+perturb_n*math.cos(kx*x+ky*y))*fv
         end,
      evolve = true,
      diagnosticMoments = { "M1i" },
   },
   ----------------------------------------------------------------------------
   -- Field solver
   ----------------------------------------------------------------------------
   field = Plasma.Field {
      epsilon0 = epsilon0, mu0 = mu0,
      init = function (t, xn)
         local x, y = xn[1], xn[2]
         local E_x  = -perturb_n*math.sin(kx*x+ky*y)/(kx+ky*alpha)
         local E_y  = alpha*E_x
         local B_z  = kx*E_y-ky*E_x
         return E_x, E_y, 0.0, 0.0, 0.0, B_z
      end,
      evolve = true,
   },
}
----------------------------------------------------------------------------
-- Run application
----------------------------------------------------------------------------
vlasovApp:run()
```

```
2   local Plasma = require("App.PlasmaOnCartGrid").VlasovMaxwell()
```

- Note: fundamental data structure in Lua is the table (more flexible Python dictionary)



Extending the input file easily

```
vlasovApp = Plasma.App {
   ----------------------------------------------------------------------------
   -- Common
   ----------------------------------------------------------------------------
   logToFile = true,
   tEnd = tEnd,            -- End time
   nFrame = nFrame,        -- Number of output frames
   lower = {0.0,0.0},      -- Lower boundary of configuration space
   upper = {Lx,Ly},        -- Upper boundary of configuration space
   cells = {Nx,Ny},        -- Configuration space cells
   basis = "serendipity",  -- One of "serendipity", "maximal-order", or "tensor"
   polyOrder = polyOrder,  -- Polynomial order
   timeStepper = "rk3",    -- One of "rk2", "rk3", or "rk3s4"
   -- MPI decomposition for configuration space
   decompCuts = {Ncx,Ncy}, -- Cuts in each configuration direction
   useShared = false,      -- If using shared memory
   -- Boundary conditions for configuration space
   periodicDirs = {1,2},   -- periodic directions (both x and y)
   ----------------------------------------------------------------------------
   -- Electrons
   ----------------------------------------------------------------------------
   elc = Plasma.Species {
      charge = chargeElc, mass = massElc,
      -- Velocity space grid
      lower = {-Vmax, -Vmax},
      upper = {Vmax, Vmax},
      cells = {Nvx, Nvy},
      -- Initial conditions
      init = function (t, xn)
         local x, y, vx, vy = xn[1], xn[2], xn[3], xn[4]
         local fv = maxwellian2D(nElc10, vx, vy, uxElc10, uyElc10, vthElc10) +
            maxwellian2D(nElc20, vx, vy, uxElc20, uyElc20, vthElc20)
         return (1.0+perturb_n*math.cos(kx*x+ky*y))*fv
         end,
      evolve = true,
      diagnosticMoments = {"M0", "M1i", "M2ij", "M3i" },
      diagnosticIntegratedMoments = { "intM0", "intM1i", "intM2Flow", "intM2Thermal" },
      coll = Plasma.LBOCollisions {
         collideWith = {'elc', 'ion'},
         frequencies = {nuElc, nuElcIon},
      },
   },
   ----------------------------------------------------------------------------
   -- Ions
   ----------------------------------------------------------------------------
   ion = Plasma.Species {
      charge = chargeIon, mass = massIon,
      -- Velocity space grid
      lower = {-VmaxIon, -VmaxIon},
      upper = {VmaxIon, VmaxIon},
      cells = {Nvx, Nvy},
      -- Initial conditions
      init = function (t, xn)
         local x, y, vx, vy = xn[1], xn[2], xn[3], xn[4]
         local fv = maxwellian2D(nIon, vx, vy, 0.0, 0.0, vthIon)
         return fv
         end,
      evolve = true,
      diagnosticMoments = {"M0", "M1i", "M2ij", "M3i" },
      diagnosticIntegratedMoments = { "intM0", "intM1i", "intM2Flow", "intM2Thermal" },
      coll = Plasma.LBOCollisions {
         collideWith = {'ion', 'elc'},
         frequencies = {nuIon, nuIonElc},
      },
   },
   ----------------------------------------------------------------------------
   -- Field solver
   ----------------------------------------------------------------------------
   field = Plasma.Field {
      epsilon0 = epsilon0, mu0 = mu0,
      init = function (t, xn)
         local x, y = xn[1], xn[2]
         local E_x  = -perturb_n*math.sin(kx*x+ky*y)/(kx+ky*alpha)
         local E_y  = alpha*E_x
         local B_z  = kx*E_y-ky*E_x
         return E_x, E_y, 0.0, 0.0, 0.0, B_z
      end,
      evolve = true,
   },
}
```

- We've added
  - Second species (ions)
  - Self-collisions
  - Cross-species collisions
  - New diagnostics (density, pressure tensor, and heat flux vector, integrated density, integrated momentum, and integrated energy)

Extending the input file easily

```
vlasovApp = Plasma.App {
   -----------------------------------------------------------------------------
   -- Common
   -----------------------------------------------------------------------------
   logToFile = true,
   tEnd = tEnd,            -- End time
   nFrame = nFrame,        -- Number of output frames
   lower = {0.0,0.0},      -- Lower boundary of configuration space
   upper = {Lx,Ly},        -- Upper boundary of configuration space
   cells = {Nx,Ny},        -- Configuration space cells
   basis = "serendipity",  -- One of "serendipity", "maximal-order", or "tensor"
   polyOrder = polyOrder,  -- Polynomial order
   timeStepper = "rk3",    -- One of "rk2", "rk3", or "rk3s4"
   -- MPI decomposition for configuration space
   decompCuts = {Ncx,Ncy}, -- Cuts in each configuration direction
   useShared = false,      -- If using shared memory
   -- Boundary conditions for configuration space
   periodicDirs = {1,2},   -- periodic directions (both x and y)
   -----------------------------------------------------------------------------
   -- Electrons
   -----------------------------------------------------------------------------
   elc = Plasma.Species {
      charge = chargeElc, mass = massElc,
      -- Velocity space grid
      lower = {-Vmax, -Vmax},
      upper = {Vmax, Vmax},
      cells = {Nvx, Nvy},
      -- Initial conditions
      init = function (t, xn)
         local x, y, vx, vy = xn[1], xn[2], xn[3], xn[4]
         local fv = maxwellian2D(nElc10, vx, vy, uxElc10, uyElc10, vthElc10) +
            maxwellian2D(nElc20, vx, vy, uxElc20, uyElc20, vthElc20)
         return (1.0+perturb_n*math.cos(kx*x+ky*y))*fv
         end,
      evolve = true,
      diagnosticMoments = {"M0", "M1i", "M2ij", "M3i" },
      diagnosticIntegratedMoments = { "intM0", "intM1i", "intM2Flow", "intM2Thermal" },
      coll = Plasma.LBOCollisions {
         collideWith = {'elc', 'ion'},
         frequencies = {nuElc, nuElcIon},
      },
   },
   -----------------------------------------------------------------------------
   -- Ions
   -----------------------------------------------------------------------------
   ion = Plasma.Species {
      charge = chargeIon, mass = massIon,
      -- Velocity space grid
      lower = {-VmaxIon, -VmaxIon},
      upper = {VmaxIon, VmaxIon},
      cells = {Nvx, Nvy},
      -- Initial conditions
      init = function (t, xn)
         local x, y, vx, vy = xn[1], xn[2], xn[3], xn[4]
         local fv = maxwellian2D(nIon, vx, vy, 0.0, 0.0, vthIon)
         return fv
         end,
      evolve = true,
      diagnosticMoments = {"M0", "M1i", "M2ij", "M3i" },
      diagnosticIntegratedMoments = { "intM0", "intM1i", "intM2Flow", "intM2Thermal" },
      coll = Plasma.LBOCollisions {
         collideWith = {'ion', 'elc'},
         frequencies = {nuIon, nuIonElc},
      },
   },
   -----------------------------------------------------------------------------
   -- Field solver
   -----------------------------------------------------------------------------
   field = Plasma.Field {
      epsilon0 = epsilon0, mu0 = mu0,
      init = function (t, xn)
         local x, y = xn[1], xn[2]
         local E_x  = -perturb_n*math.sin(kx*x+ky*y)/(kx+ky*alpha)
         local E_y  = alpha*E_x
         local B_z  = kx*E_y-ky*E_x
         return E_x, E_y, 0.0, 0.0, 0.0, B_z
      end,
      evolve = true,
   },
}
```

- We've added
  - Second species (ions)
  - Self-collisions
  - Cross-species collisions
  - New diagnostics (density, pressure tensor, and heat flux vector, integrated density, integrated momentum, and integrated energy)

Gkeyll 2.0 (LuaJIT)

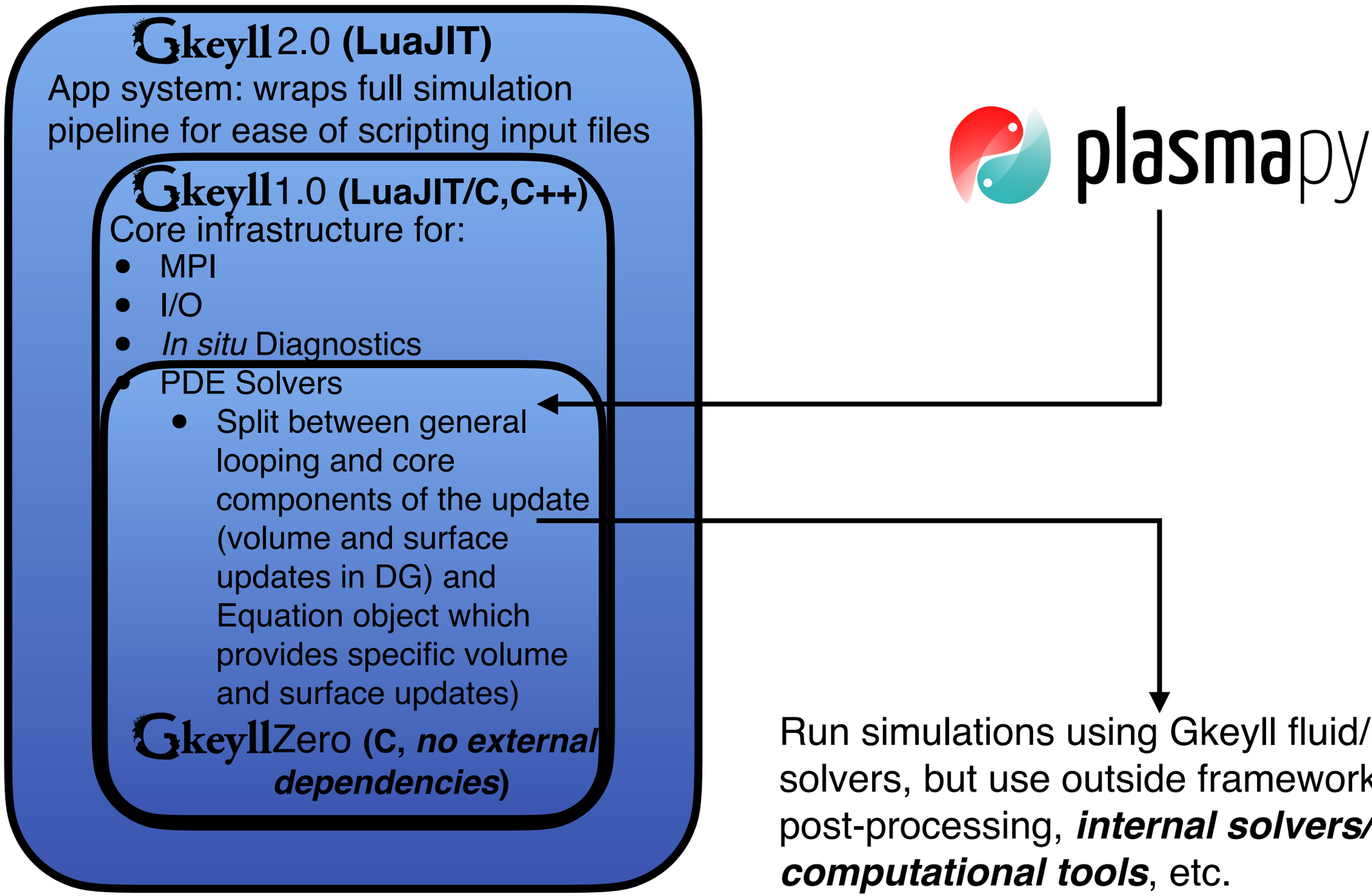
App system: wraps full simulation pipeline for ease of scripting input files

Gkeyll 1.0 (LuaJIT/C,C++)

Core infrastructure for:

- MPI
- I/O
- *In situ* Diagnostics
- PDE Solvers
  - Split between general looping and core components of the update (volume and surface updates in DG) and Equation object which provides specific volume and surface updates)

GkeyllZero layer underneath it all



Live GkeyllZero Demo

Interested in learning more about

Gkeyll

https://gkyl.readthedocs.io/en/latest/gkyl/presentations.html.
https://gkyl.readthedocs.io/en/latest/gkyl/pubs.html
Jimmy Juno YouTube Channel