PlasmaCalcs

A Consistent Interface for Plasma Calculations from Any Inputs

Samuel Evans[1], Save Koontaweepunya[1] (presenter)

Additional PlasmaCalcs Contributors:

Alexander Green[1], Juan Martínez-Sykora[2,3,4,5], Tess Goodman[1], Matthew Gregor[1]

[1] Boston University Center for Space Physics
[2] SETI Intitute
[3] Lockheed Martin Solar & Astrophysics Laboratory
[4] Rosseland Centre for Solar Physics, University of Oslo
[5] Institute of Theoretical Astrophysics, University of Oslo

Logo designed by Tess Goodman

The Problem…

- Plasma Physics is complicated!

Lots of data sources

- Simulators with different physics (kinetic, multifluid, MHD, …)
- Observations (not yet implemented in PlasmaCalcs, but it should certainly be doable!)
- Different grids (2D, 3D, rectilinear, curvilinear, …)
- Curious researchers asking "*what happens if I just change this parameter a bit?*"
- Any arbitrary Dataset (e.g. many artifically-chosen parameters)

Lots of quantities

$$\vec{E}, \vec{B}, \vec{J}, q, m,$$
$$n, \rho, \vec{u}, \vec{p}, T, \beta,$$
$$v_T, v_{alfven}, c_{sound},$$
$$\vec{u}_{drift}, \sigma_H, \vec{u}_{\mathrm{H}}, \sigma_P, \vec{u}_{\mathrm{P}},$$
$$\nu_{s,n}, \kappa_s, \lambda_{Debye}, \lambda_{mfp},$$
$$\ldots$$

Lots of operations

$$|\vec{a}|, \vec{a} \cdot \vec{b}, \vec{a} \times \vec{b}, \vec{a} \perp \vec{b},$$
$$\frac{\partial}{\partial t}, \frac{\partial}{\partial x}, \nabla f, \nabla \cdot \vec{f}, \nabla \times \vec{f},$$
$FFT(f)$, interpolate, convolve, stats (min, mean, stddev, …), arithmetic (+, -, ×, ÷, sin, cos, tan, log, e^x, …), …

The Solution: PlasmaCalcs

Consistent Interface
across all data sources

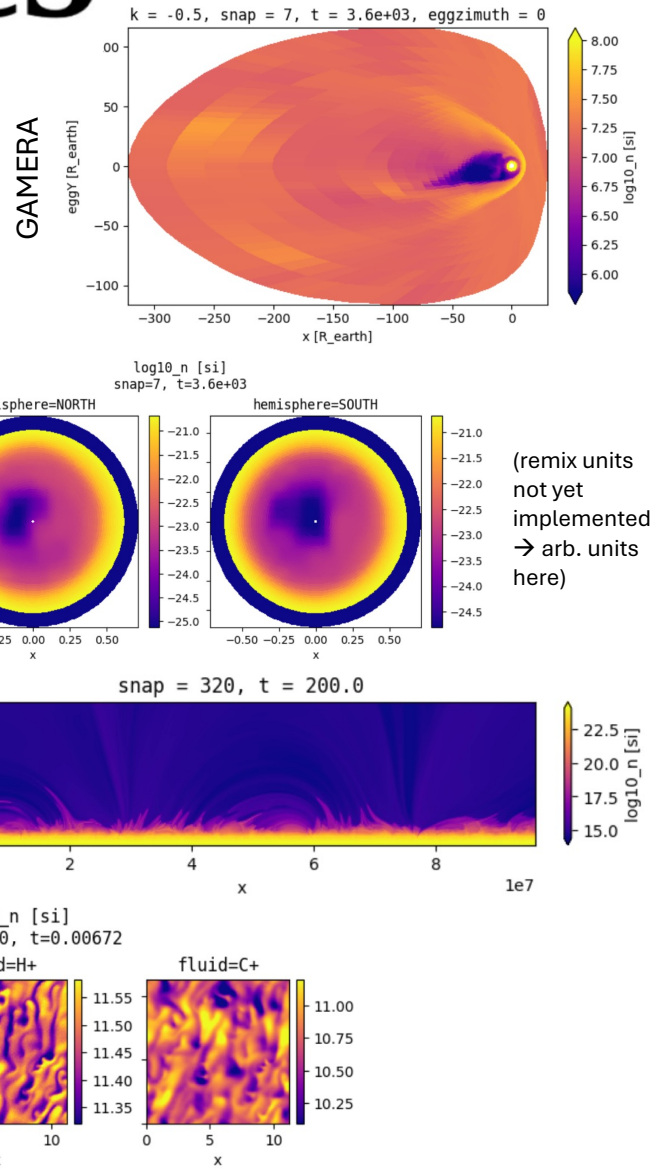
GAMERA
REMIX
Bifrost
EPPIC
…

```
cc=GameraCalculator(...)
cc=RemixCalculator(...)
cc=BifrostCalculator(...)
cc=EppicCalculator(...)
...
```

```
cc('log10_n')
```

(or some other quantity!)

Lots of operations available!
Lots of quantities available!



(remix units not yet implemented → arb. units here)

PlasmaCalcs

Dimensions and coordinates are attached directly to the data (stored as xarray)!

```
import PlasmaCalcs as pc
rc = pc.mage.RemixCalculator(...)
rc('log10_n')
```

makes:

xarray.DataArray 'log10_n' (**snap**: 14, **hemisphere**: 2, **theta**: 360, **radius**: 44)

-24.48 -24.48 -24.48 -24.48 -24.48 ... -25.85 -25.85 -25.85 -25.85

▼ Coordinates:

theta	(theta)	float64	0.0 0.01745 0.03491 ... 6.248 6.266
radius	(radius)	float64	0.01785 0.03569 ... 0.6943 0.7071
x	(theta, radius)	float64	0.01785 0.03569 ... 0.6942 0.707
y	(theta, radius)	float64	-2.101e-19 -4.337e-19 ... -0.01234
snap	(snap)	object	0 1 2 3 4 5 6 7 8 9 10 11 12 13
t	(snap)	float64	-7.2e+03 3.013e-12 ... 7.2e+03
hemisphere	(hemisphere)	object	NORTH SOUTH

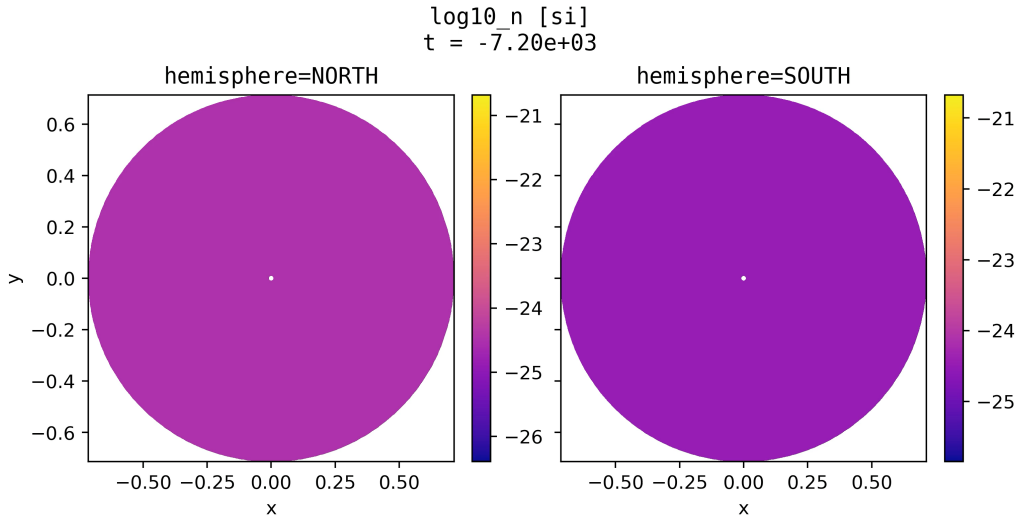
▶ Indexes: (4)

▼ Attributes:

units : si

```
rc('log10_n').pc.subplots().ani()
```

makes:*



This is 4D data!
It is easy to plot using subplots()

**renders in-line in Jupyter notebook. To save to file, use .save() instead.*

PlasmaCalcs

Slicing is easy! (And, for efficiency, applies before loading*)

* Fully implemented for GAMERA, Bifrost, EPPIC;
Not fully implemented in for REMIX yet

```
import PlasmaCalcs as pc
rc = pc.mage.RemixCalculator(...)
arr = rc('log10_n', snap=8,          # pick just one snapshot
    theta=slice(40, None, 10),       # skip 40 angles & downsample by 10x
    radius=slice(0.2, 0.7),          # fractional indexing! 20% to 70%
    hemisphere='NORTH')              # just look at northern hemisphere
```

xarray.DataArray 'log10_n' (**theta**: 32, **radius**: 22)

-23.62 -23.58 -23.55 -23.5 -23.42 ... -21.64 -21.43 -21.24 -21.1

▼ Coordinates:

theta	(theta)	float64	0.6981 0.8727 1.047 ... 5.934 6.109
radius	(radius)	float64	0.1775 0.1951 ... 0.5102 0.5255
x	(theta, radius)	float64	0.136 0.1494 ... 0.5024 0.5175
y	(theta, radius)	float64	0.1141 0.1254 ... -0.08859 -0.09124
snap	()	object	8
t	()	float64	4.2e+03
hemisphere	()	object	NORTH

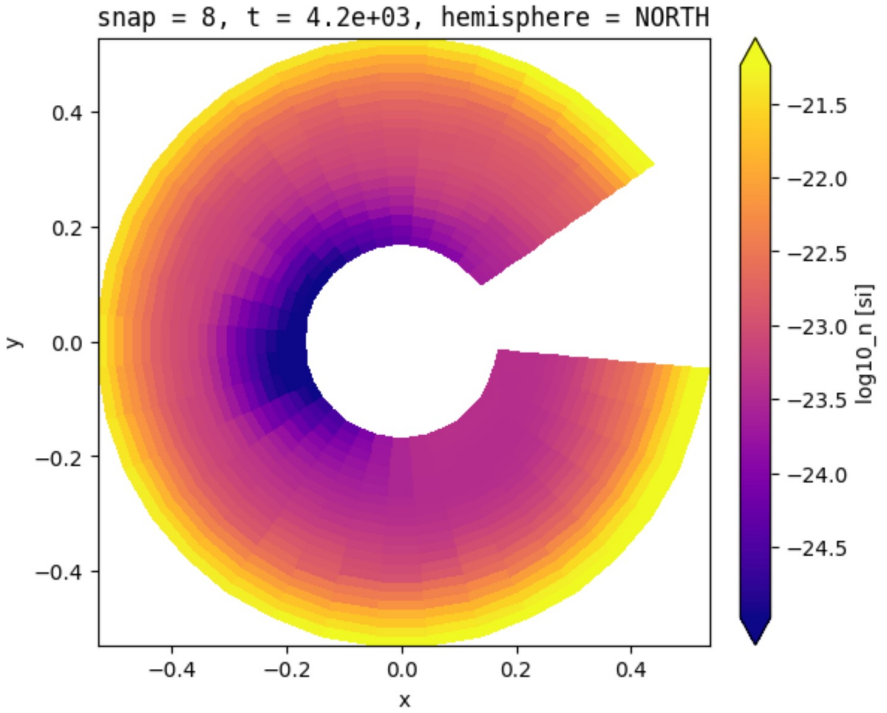
▶ Indexes: (2)

▼ Attributes:

units : si

```
arr.pc.image()
```

makes:



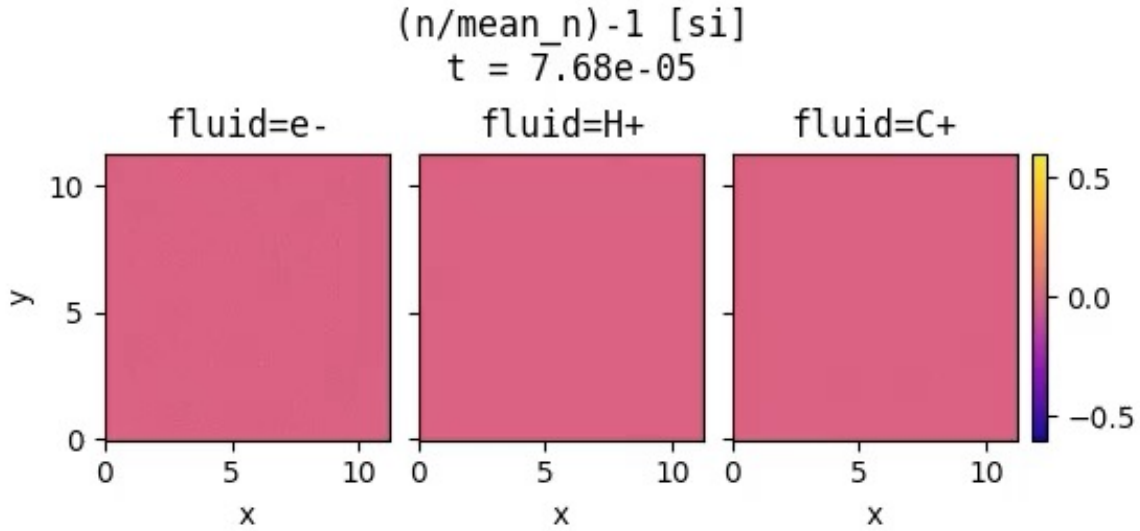
PlasmaCalcs

It is easy to combine lots of operations and to customize behavior!

```
import PlasmaCalcs as pc
ec = pc.EppicCalculator(...)
```

```
ec('(n/mean_n)-1').pc.subplots(vmin=-0.6, vmax=0.6)
```

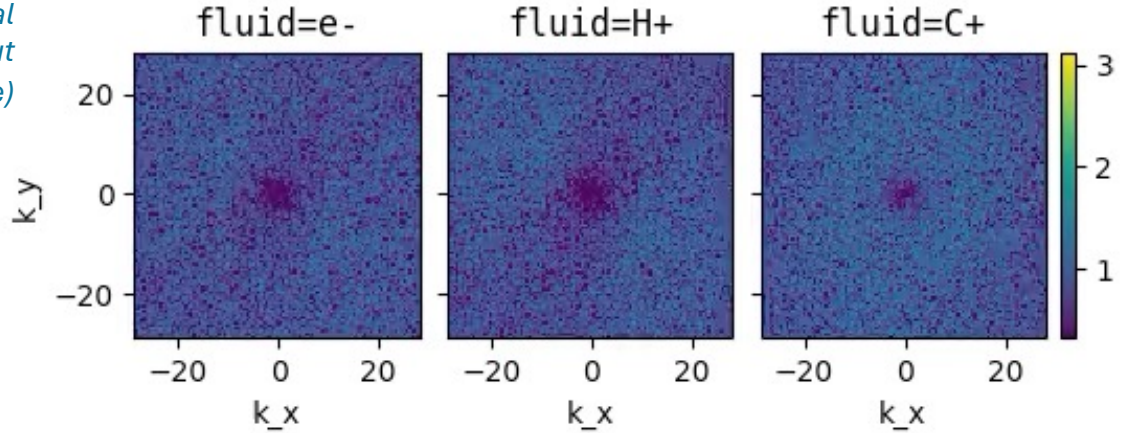
$$= \frac{n}{\langle n \rangle} - 1$$




```
arr=ec('log10_abs_radfft_((n/mean_n)-1)', fft_keep=0.2)
```

$$= \log_{10} \left| \mathrm{FFT} \left(\frac{n}{\langle n \rangle} - 1 \right) \right|$$

keeps only the central 20% of the FFT output (here, the rest is just noise)




```
arr.pc.subplots(cmap='vidiris', share_vlims='all')
```

shares auto-inferred vmin, vmax for all plots

PlasmaCalcs

"u" (velocity) has vector components (x, y, z) → "component" dimension
and can be "u" for single fluid (MHD) or multiple species (as shown here) → "fluid" dimension

```
import PlasmaCalcs as pc
ec = pc.EppicCalculator(...)
arr = ec('u', component=['x', 'y'])
```

xarray.DataArray 'u' (**snap**: 100, **fluid**: 3, **component**: 2, **x**: 512, **y**: 512)

-5.718e+03 -9.171e+03 -244.2 ... -2.567e+03 -2.554e+03 -2.555e+03

▼ Coordinates:

x	(x)	float64	0.0 0.022 0.044 ... 11.22 11.24
y	(y)	float64	0.0 0.022 0.044 ... 11.22 11.24
snap	(snap)	object	5120 10240 15360 ... 506880 512000
t	(snap)	float64	7.68e-05 0.0001536 ... 0.00768
fluid	(fluid)	object	e- H+ C+
component	(component)	object	x y

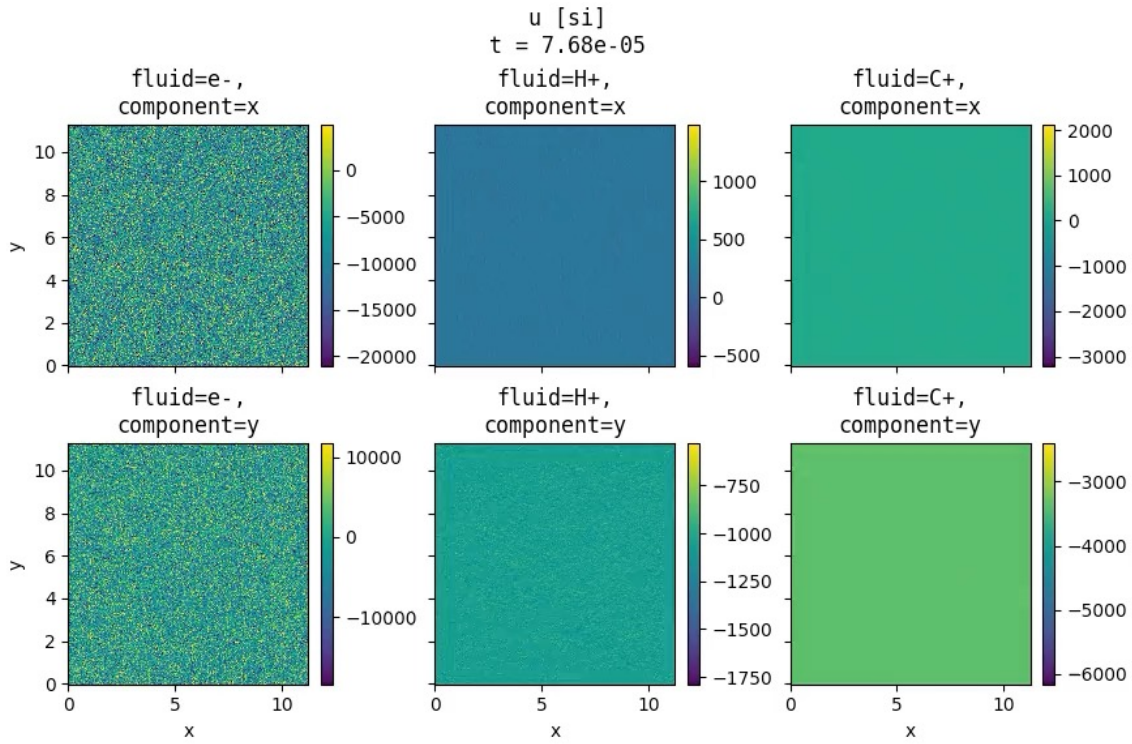
▶ Indexes: (5)

▼ Attributes:

units : si

This is 5D data!
It is **also** easy to plot using subplots()

```
arr.pc.subplots().ani()
```



PlasmaCalcs

The underlying grid does not need to be rectilinear!

```
import PlasmaCalcs as pc
gc = pc.mage.GameraCalculator(...)
gc.coords_units = 'R_earth'
gc('n')
```

xarray.DataArray 'n' (**snap**: 14, **i**: 48, **j**: 48, **k**: 64)

2e+05 2e+05 2e+05 2e+05 ... 1.456e+07 1.456e+07 1.456e+07 1.456e+07

▼ Coordinates:

i	(i)	int64	0 1 2 3 4 5 6 ... 42 43 44 45 46 47
j	(j)	int64	0 1 2 3 4 5 6 ... 42 43 44 45 46 47
k	(k)	int64	0 1 2 3 4 5 6 ... 58 59 60 61 62 63
x	(i, j)	float32	2.199 2.188 2.166 ... -305.5 -305.9
y	(i, j, k)	float32	0.07921 0.07845 ... 1.768 1.785
z	(i, j, k)	float32	0.003891 0.01164 ... -0.08769
azimuth	(k)	float32	0.04909 0.1473 ... -0.1473 -0.04909
rcyl	(i, j)	float32	0.0793 0.237 0.3928 ... 5.369 1.787
snap	(snap)	object	0 1 2 3 4 5 6 7 8 9 10 11 12 13
t	(snap)	float64	-7.2e+03 0.07049 ... 7.2e+03

The grid is 3D along i, j, k

x, y, z coordinates vary across the grid
(e.g., x varies across i and j)

This grid happens to be azimuthally symmetric, so we add extra coordinates:

$$azimuth = \tan^{-1}\left(\frac{z}{y}\right)$$

$$r_{cyl} = \sqrt{y^2 + z^2}$$

▶ Indexes: (4)

▼ Attributes:

units :	si
coords_units :	R_earth

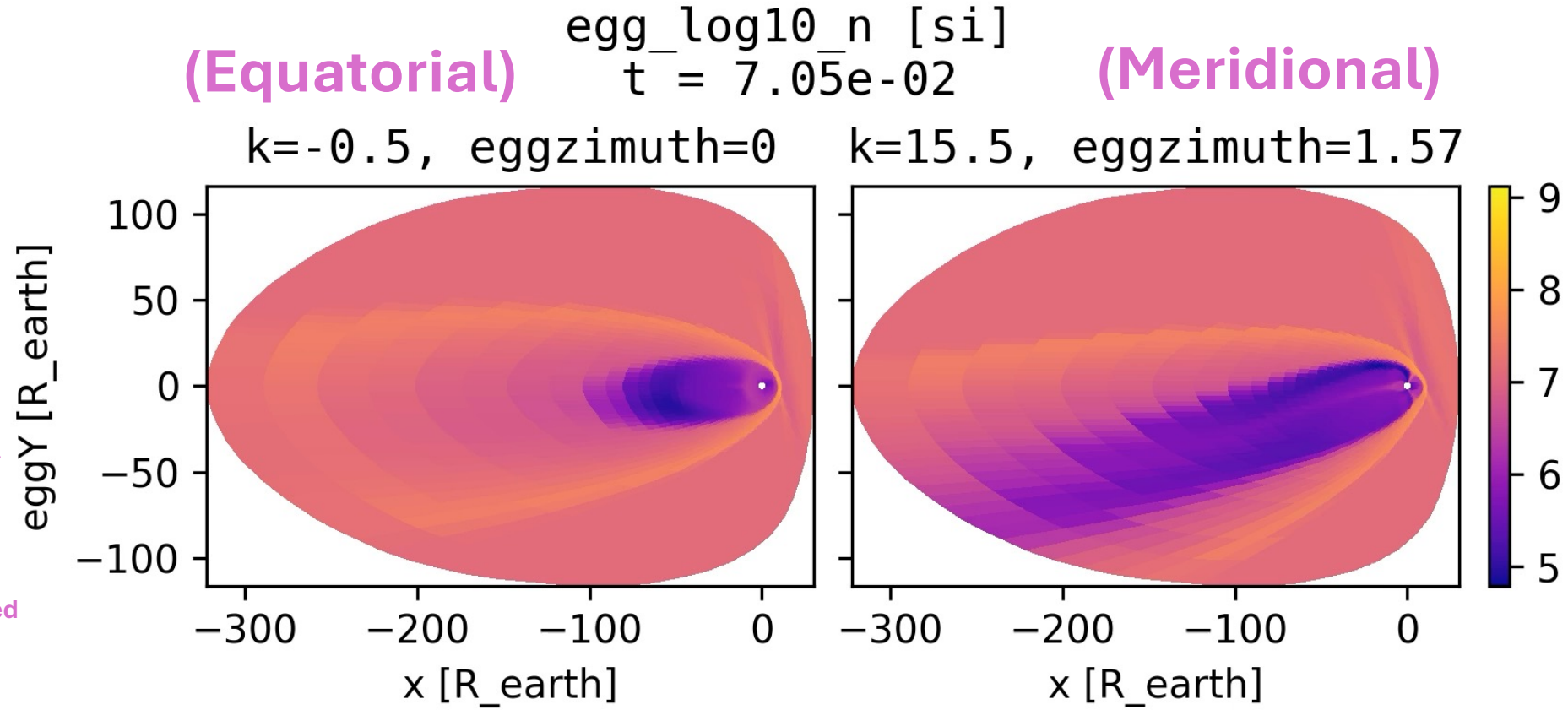
PlasmaCalcs

The underlying grid does not need to be rectilinear!

```
import PlasmaCalcs as pc
gc = pc.mage.GameraCalculator(...)
gc.coords_units = 'R_earth'
```

```
arr = gc('egg_log10_n', eggzimuth=[0, np.pi/2])
arr.pc.subplots(share_vlims='all').ani()
```

special syntax for GAMERA in particular, to get "egg slice" at a given azimuth.



In equatorial slice, eggY = y
In meridional slice, eggY = z

In general,

$\mathbf{eggY} = \pm r_{cyl} = \pm\sqrt{y^2 + z^2}$

with sign = +1 in "top half"
-1 in "bottom half"

GAMERA-specific details are implemented in the gamera subpackage, so they only affect GAMERA-related parts of the code!

PlasmaCalcs

- Internal implementation looks like physical equations!

```python
@known_var(deps=['abs_q', 'mod_B', 'm'], aliases=['cyclof', 'omega_c'])
def get_gyrof(self):
    '''(unsigned) gyrofrequency. gyrof == |sgyrof| == |q| |B| / m == |charge| * |B| / mass.'''
    return self('mod_B') * (self('abs_q') / self('m'))
```

m and q might be constants or vary with fluid.
B might be constant or vary in space or time.
This code handles it all!
(xarray is very helpful!!!)

- Has custom help systems to learn which options exist!

```
gc.help('gyro')
```

```
Showing help for related KNOWN_VARS: ['sgyrof', 'gyrof', 'cyclof', 'omega_c', 'timescale_gyrof']

'sgyrof':
    signed gyrofrequency. sgyrof == q |B| / m == charge * |B| / mass. Negative when charge < 0.

'gyrof':
    (unsigned) gyrofrequency. gyrof == |sgyrof| == |q| |B| / m == |charge| * |B| / mass.

'cyclof':
    (unsigned) gyrofrequency. gyrof == |sgyrof| == |q| |B| / m == |charge| * |B| / mass.

'omega_c':
    (unsigned) gyrofrequency. gyrof == |sgyrof| == |q| |B| / m == |charge| * |B| / mass.

'timescale_gyrof':
    timescale for cyclotron motion. 2 * pi / gyrof.  (Hz, not rad/s)
    gyrof = |q| |B| / m.
```

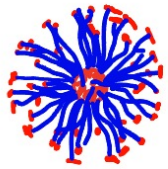
```
gc.tree('gyrof')
```

```
▼ QuantTree([depth=0, height=2, size=6], obj=MatchedVar('gyrof', deps=['abs_q', 'mod_B', 'm']))
    ▼ QuantTree([depth=1, height=1, size=2], obj=MatchedPattern('abs_q', 'get_abs', deps=[0], groups=['q']))
        ▼ QuantTree([depth=2, height=0, size=1], obj=MatchedVar('q'))
    ▼ QuantTree([depth=1, height=1, size=2], obj=MatchedPattern('mod_B', 'get_mod', deps=[1],
    ignores_dims=['component'], groups=['mod', 'B', None]))
        ▼ QuantTree([depth=2, height=0, size=1], obj=MatchedVar('B', dims=['snap', 'component']))
    ▼ QuantTree([depth=1, height=0, size=1], obj=MatchedVar('m'))
```

```
gc.help_call_options('fft')
```

```
Showing self.help_call_options(): docs for keys from self.kw_call_options()
Only showing keys containing search='fft'
--------------------------------------------------------------------------------
'fft_dims': the dims over which to possibly apply fft (FFTLoader methods).
    will only apply fft along these dims for an array if they actually appear in the array.
    None --> use self.maindims. (this is the default.)
    See also: self.fft_dims_for(array).

'fft_half': alias to self.fft_slices.half

'fft_keep': alias to self.fft_slices.keep

'fft_slices': the dict of indexers to apply to all fft results from self, by default.
    keys can be a pre-fft or post-fft dimension name,
        e.g. 'x' or 'freq_x' both lead to slicing of the result's 'freq_x' dimension.
        (note if rad=True it would be 'k_x' in the result, and 'x' would apply but not 'freq_x'.)
        all other keys (not a pre-fft or post-fft dimension name) are ignored.
    values can be slice, int, iterable, or non-integer value between -1 and 1.
        fractional values are interpreted as a fraction of the length of the corresponding dimension,
        as per interprets_fractional_indexing. Negative fractions refer to distance from the end.
        e.g., dict(x=slice(-0.3, None, 0.01), y=0.8), where x and y correspond to length 1000,
            would be equivalent to dict(x=slice(-300, None, 10), y=800).
    Can also have special keys (which apply to all fft dims without a specifically-related key):
        keep: fraction of each dimension to keep,
            e.g. keep=0.4 with length=1000 would result in slice(300, 700),
            since that keeps 400 out of 1000 points, and is centered at 1000/2.
        half: dimension(s) along which to keep only the right half of the result.
        step: slice step. Can also be fractional to use fraction of dimension length.

        For more help on special keys, see help(self._fft_slices_cls) or help(FFTSlices)

'fft_step': alias to self.fft_slices.step
```

(fft_keep option was used on a previous slide!)



PlasmaCalcs

...is most helpful if you spend more time writing code than running code!

- Great at **quickly exploring & visualizing data**, computing derived quantities, navigating many possible options and settings, inspecting intermediate values, and **comparing across different data sources.**
- *Not* centered around "push a button, output standardized plots."
  - But, if you build a pipeline like that *using* PlasmaCalcs, you get compatibility across many kinds of inputs for free!
- Aims for accuracy, crashes if anything is ambiguous.
- Highly modularized and object-oriented. Shared behaviors are inherited, unique quirks are handled directly where relevant. **Built to grow & support many more kinds of inputs in the future!**

PlasmaCalcs

How to get started or get involved?

gitlab.com/Sevans7/plasmacalcs

- Find PlasmaCalcs on PyPI, Zenodo, or GitLab
  - Installing is easy! Just: pip install PlasmaCalcs
  - Readthedocs pages available online
  - Various example Jupyter notebooks available online, linked in README

- Reach out to us after the talk! Happy to discuss more potential use-cases and possible collaborations

- Join the PlasmaCalcs Slack! (Ask for link)

Thank you for your time!!!