

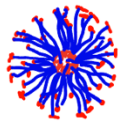
# PLASMACALCS

Getting Started Guide

(for Potential Developers of a New Hookup/Calculator)

# PLASMACALCS — Object Oriented

- Provides a consistent interface for plasma calculations (in Python) from any inputs!
- Uses object-oriented programming to accomplish this.
- Think: what *is* this object, what is it meant to handle? Examples:
  - “this class deals with directly loading the data. It should handle any quirks related to loading data from my simulation files.”
  - “this attribute should always tell me the main dimensions. It will usually be ('x', 'y', 'z'), but for different hookups it could a list of any strings.”
- Classes inherit from multiple parents to combine their behaviors!
  - Your Calculator will inherit from DirectLoader, MainDimensionsHaver, and possibly many others.

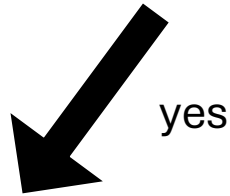


# PLASMACALCS — Dimensions

- Uses well-labeled multidimensional arrays, via xarray.
- Has “main dimensions” (the inherent dimensions of your data), and “special dimensions” (treated specially by PlasmaCalcs):
  - component — vector components (e.g. `Component('y', 1)`)
  - snap — snapshot (Snap objects usually have a name, number, and t)
  - fluid — species (e.g., `Fluid('e-', q=-1)`, `Fluid('H_I', m=1, q=0)`)
  - jfluid — species, for quantities with fluid-fluid interaction (like `nusj`)
- Each special dimension in a PlasmaCalculator has a sense of its “current value(s)” and “all possible values”.
  - e.g., `cc.snap` is current snap (or snaps); `cc.snaps` is all possible snaps.
  - e.g., `cc.snap=slice(3, 7)`; `cc('B')` to get array of B at snaps 3, 4, 5, 6.

# Potential Off-Ramp

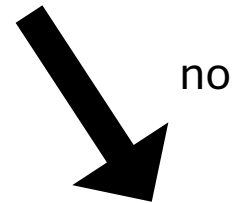
Is the data you analyze usually small enough to load all the base variables (e.g.,  $\rho$ ,  $u$ ,  $T$ ,  $B$ ) into memory simultaneously?



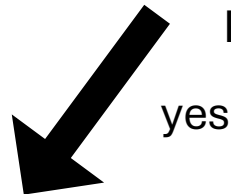
You can probably get away with just using a `FromDatasetCalculator` (or related class, e.g. `VectorlessFromDatasetCalculator` if all your values are scalars.)

You just need to make a properly-labeled `xarray.Dataset` with all your data, then use it to initialize one of those calculators!

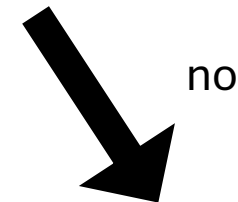
For a walkthrough, follow the example, `getting-started_from_dataset.ipynb`, in `PlasmaCalcs/examples`. (Est. time: **10 mins**)



Are you willing to analyze only small chunks at a time? Do you want a shortcut for trying out `PlasmaCalcs` with some of your data for now?



You will want to build a new hookup in `PlasmaCalcs`. (Est. time: **hours or days**, depending on how “quirky” your data is, and how familiar you are with working in a new codebase.) Proceed to next slide...



# What's in a hookup?

The main interface for loading your data & computing relevant quantities.

**YourCalculator**

Handles initializing calculator (e.g., tell it which directory contains your data). Can manage other things, but ideally most details will be inherited.

...inherits from...

**YourBasesLoader**

**Connect “base” quantities to the PlasmaCalcs code architecture.**

E.g., when user asks to get calculator(“B”), go here for instructions.

Bases can include m, q, n, u, T, E, B, nusj

**YourDirectLoader**

**Handle loading values “directly”** (e.g., from files). Probably numpy; convert to xarray elsewhere.

Recommended, but semi-optional. (E.g., maybe you already have another code which handles this?)

# What's in a hookup?

The main interface for loading your data & computing relevant quantities.

**YourCalculator**

Handles initializing calculator (e.g., tell it which directory contains your data). Can manage other things, but ideally most details will be inherited.

```
class YourCalculator(YourBasesLoader, YourDirectLoader, ..., PlasmaCalculator):
    '''calculator for loading your data!'''
    def __init__(self, ...):
        # <-- write some code to initialize the calculator!
        # e.g. maybe you want to input "dirname" and do:
        self.dirname = dirname
        # if you have any "snapshots" then also include:
        self.init_snaps()

    def init_snaps(self):
        '''initialize self.snaps and self.snap'''
        self.snaps = ... # <-- SnapList of all possible snaps

    maindims = ('x', 'y', 'z') # or something else

    def get_maindims_coords(self):
        '''return {'x': xcoords, 'y': ycoords, 'z': zcoords}.'''
        # <-- write your own code to do this!
```

\*rough idea, not a complete working example.

# What's in a hookup?

\*rough idea, not a complete working example.

## YourBasesLoader

Connect “base” quantities to the PlasmaCalcs code architecture.

E.g., when user asks to get calculator(“B”), go here for instructions.

Bases can include m, q, n, u, T, E, B, nusj

```
class YourBasesLoader(AllBasesLoader, SimpleDerivedLoader):
    '''base quantities based on Your data.'''
    @known_var(load_across_dims=['snap', 'component'])
    def get_B(self):
        '''magnetic field (directly from Your data)'''
        B_internal_name = 'Bfield' # gets passed to DirectLoader
        return self.load_maintains_var_across_dims(
            B_internal_name, u='b_field',
            dims=['snap', 'component'])

    @known_var(load_across_dims=['snap', 'fluid'])
    def get_n(self):
        '''number density of each fluid'''
        n_internal_name = 'ndens' # gets passed to DirectLoader
        return self.load_maintains_var_across_dims(
            n_internal_name, u='number_density',
            dims=['snap', 'fluid'])

    @known_var(load_across_dims=['fluid'])
    def get_m(self):
        '''mass, of a "single particle". For protons, ~= +1 amu'''
        return xr.DataArray(self.fluid.m * self.u('M'))
```

# What's in a hookup?

```
class YourDirectLoader(DirectLoader):
    '''manages loading directly from Your data files.'''
    snapdir = property(lambda self: os.path.join(self.dirname, 'outputs'),
                        doc='''directory with snapshot data. The example here assumes
                            snapshots are inside a folder at "self.dirname/outputs".''' )

    def _var_for_load_fromfile(self, varname_internal):
        '''return var, suitably adjusted to pass into load_fromfile().
        Here, just do: if self._loading_component, append component.
        E.g. 'Bfield' --> 'Bfieldz' if loading 'z' component.
        ...

        result = varname_internal
        if getattr(self, '_loading_component', False):
            result = result + str(self.component)

    def load_fromfile(self, fromfile_var, snap=None):
        '''return numpy array of fromfile_var, loaded directly from file.
        snap: None, str, int, or Snap
        ... the snapshot number to load. if None, use self.snap.
        ...

        # assuming only 1 snapshot per file:
        snap = self._as_single_snap(snap)
        # getting filename (example... you will need to adjust)
        filename = os.path.join(self.snapdir, f'{str(snap)}.h5')
        with h5py.File(filename, 'r') as file:
            result = file[fromfile_var][:]
        return result
```

\*rough idea, not a  
complete working  
example.

## YourDirectLoader

**Handle loading values “directly”**  
(e.g., from files). Probably numpy;  
convert to xarray elsewhere.

Recommended, but semi-optional.  
(E.g., maybe you already have  
another code which handles this?)



# What's in a hookup?

- Anything else you want!
- It is common to make more parent classes for YourCalculator, for loading quantities and making helper functions very specific to your particular hookup.
- To get started, make a new folder in PlasmaCalcs/hookups.
- Consider looking at the other hookups codes for more examples!

# Sidenote: Working with MHD data?

- Can use muram and bifrost hookups as examples.
- Consider using MhdBasesLoader and MhdCalculator!
  - can deal with EOS lookup tables (like in Bifrost)
  - can do radiative calculations like VDEMs, spectra, and  $G(T)$
  - simplifies “multifluid analysis based on single-fluid MHD”. (E.g., “assuming photospheric abundances and Saha equation, compute densities of each element’s neutrals and ions”.)

# PLASMA CALCS — Design Principles

- **Physics code should look like physical equations** as much as possible. (Try to encapsulate any coding details elsewhere.)
- **Extra dependencies are fine, but should be optional** — only require *your\_special\_package* inside *your* parts of the code.
- **Never give wrong answers.** If there is any ambiguity, or something might be wrong, crash! (Corollary: if no crash, can trust result.)
- **Write documentation and comments! More is always better.** If you actually do too much, someone will let you know. That's not likely though.
  - *For known quantities: what is it, physically? and, what is the formula?*
  - *For functions: what does it return? and, what is each of the inputs?*

These slides are available in .pptx format at  
<https://gitlab.com/Sevans7/plasmacalcs-assets>  
inside of the “examples\_assets” folder.