Notes workshop yt

Introduce main concepts

Dataset

- metadata
- unit system
- config & options
- loadable + derivable fields

? Think about it as the method section about the simulation

Ex: `ds = yt.load("output_00080")`

Data Container

- basic metadata
  sphere center, radius, ...
- actual data ⚠ data are immutable!

? Think about it as the analysis section

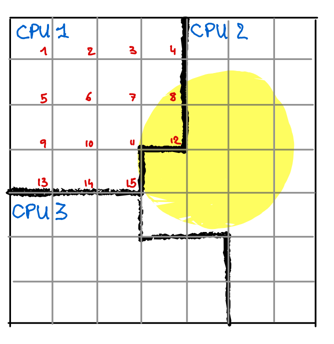
Ex:
```
sp = ds.sphere(center, radius)
ad = ds.all_data()
grid = ds.covering_grid(left, right, level)

sp["gas", "density"]  # 1D array
grid["gas", "temperature"]  # 3D array
```

Index

(this is technical, but useful to understand how yt works)

- mapping from (x, y, z, level) to (file, position-in-file)
- built internally from hilbert keys (since yt 4.4) + amr files

One example



Ex:
```
ds = yt.load("output_00080")
# Will trigger point 1 only
sp = ds.sphere(center, radius)
# Will trigger points 2-5
sp["gas", "density"]
# will trigger points 2'-3'
sp["star", "particle_position"]
```

What happened internally?

region

fast 1. may intersect with CPUs 1 & 2 (checked from Hilbert curve)

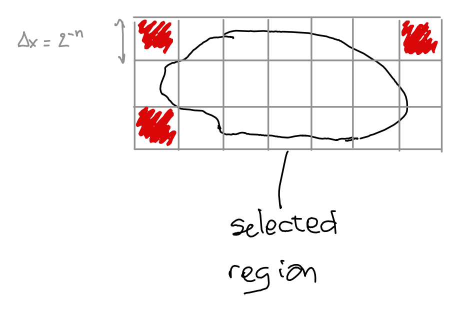
for cells

- slow 2. read corresponding AMR files
- fast 3. find whether intersection actually happens
- 4. return position of cells
  - fast CPU 1 : cells 8 & 12
  - CPU 2 : cells [...]
- slow 5. read corresponding hydro/rt/.... files at selected positions

for particles

- slow 2'. Read corresponding part files
- meh 3'. Filter out particles outside of selection

Note: yt's Hilbert key intersection is somewhat similar to the bounding box option in e.g. amr2cube BUT
- it returns fewer false positives,
- works with arbitrary 3D geometries.



1. Compute bounding box of selection
2. Split smallest axis into at least three chunks with size $\Delta x = 2^{-n}$
3. Fill bound. box with cubes of size Δx
4. Reject cubes fully outside selection (■)
5. For all others, find range of Hilbert keys at level n in cube
6. Build union of CPUs which have Hilbert key ranges overlapping with any range of step 5.

Getting the raw data

Once you have
- a dataset
- a data container

you can get the raw data in it.

To list the available fields : `ds.field_list`
and derived ones : `ds.derived_field_list`

cell based	index :	amr structure
	hydro, rt :	hydro and rt data
	deposit :	qties interpolated from particles
	ramses :	raw qties as read from disk
particle based	io :	all particles (DM, stars, ..)
	nbody :	collisionless but likely incorrect
	star, DM, tracer :	defined if you use particle families

Ex: `ad["gas", "density"]`, `ad["DM", "particle_mass"]`

And plotting it

Basic plots can be achieved with

for cells (yt.SlicePlot , yt.ProjectionPlot, yt.OffAxisProjectionPlot

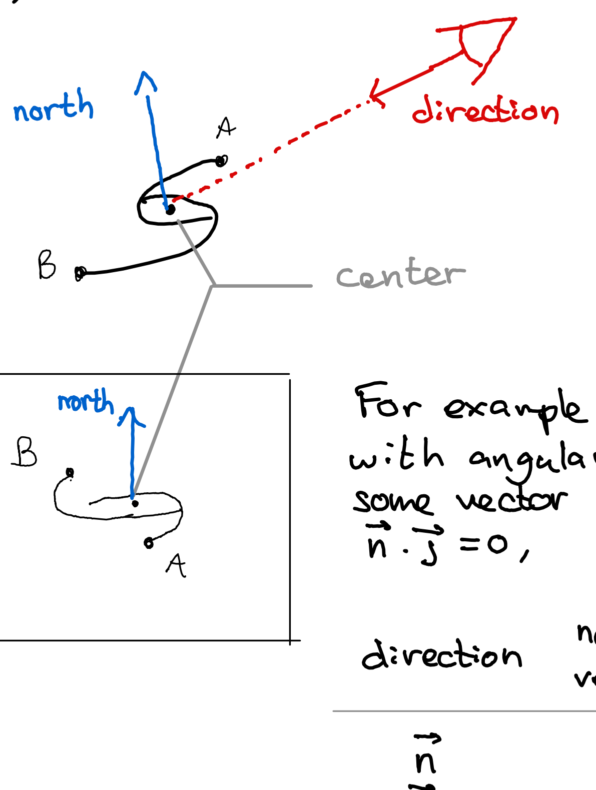
for parts (yt.ParticleProjectionPlot

yt.PhasePlot

⚠ All the above take an optional argument "data_source" to provide the data container you want to use. If omitted, yt will use `ds.all_data()`

? For off axis projections, you need to specify a camera angle and (optionally) a north vector; without the latter, the plot is defined up to a rotation. north_vector does not affect the camera's direction. Think about it as : direction controls how to make the map, north_vector orients it.

Ex:
```
yt.OffAxisProjectionPlot(
    ds,
    direction,
    ("gas", "density"),
    data_source = sp,
    center = center,
    width = (20, "kpc"),
    north_vector = north,
)
```



For example, for a galaxy with angular mom. $\vec{j}$ and some vector $\vec{n}$ s.that $\vec{n} \cdot \vec{j} = 0$,

direction	north vector	resulting view
$\vec{n}$	$\vec{j}$	edge-on
$\vec{j}$	$\vec{n}$	face-on
$\vec{j}+\vec{n}$	$\vec{j}-\vec{n}$	tilted 45°

Complete example

```
ds = yt.load("output_00080")
# Assuming 'center' is given
sp = ds.sphere(center, (100, "kpc"))
# Define bulk velocity so that "radial_velocity"
# is properly defined
sp.set_field_parameter("bulk_velocity", vel)


p = yt.ProjectionPlot(
    ds,
    "x"
    [
        ("gas", "density"),
        ("gas", "temperature"),
        ("gas", "radial_velocity"),
    ],
    data_source = sp,
    center = sp.center,
    width = (100, "kpc"),
    weight_field = ("gas", "density"),
)
p.set_cmap(("gas", "radial_velocity"), "BuOr")
p.set_log(("gas", "radial_velocity"), False)
p.set_zlim(("gas", "radial_velocity"), -100, 100)
p.set_unit(("gas", "radial_velocity"), "km/s")

p.save("projection/")
```

You should now have density-weighted projs of your object!

A deeper dive in ... data containers

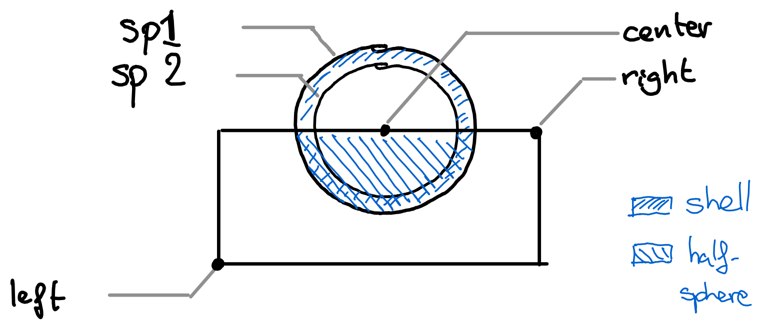
Simple ones : ds.region(left, right)
ds.sphere(center, radius)
ds.disk (center, normal, radius, height)
ds.all_data()
ds.ellipsoid(...)
ds.ray(...)

Interpolations : ds.covering_grid(...)
ds.smooth_covering_grid()

Making it better : containers can be combined!

Ex :
```
sp1 = ds.sphere(center, (10, "kpc"))
sp2 = ds.sphere(center, (9, "kpc"))
lower = ds.region(left, right)
shell = sp1 - sp2
half_sphere = sp1 & lower
```

Boolean or is also supported.



What if I want only regions containing, e.g., cold gas?
cut_region to the rescue!

Ex :
```
sp = ds.sphere(...)
cold = sp.cut_region("obj['gas', 'temperature'] < 1e4")
# or equivalently
cold = sp.include_below(("gas", "temperature"), 1e4)
```

Ex :
```
sp.cut_region([
    "obj['gas', 'temperature'].to('K') < 1e4",
    "obj['gas', 'density'].to('mp/cm**3') > 10",
]) # select the cold and dense gas
```

⚠ Cut regions end up as mere boolean masks on the cells... so you can't filter based on particles.
(although you should be able to get particles in selected cells : cold["star", "particle_mass"] should only include stars in cold gas).
To filter particles ... use a particle filter!

Mesh-to-particle, particle-to-mesh

Let's assume you have particles and want to "depose" them on the mesh:

```
ds.add_deposited_particle_field(
    particle_type, field_name, "cic" or "nearest"
)
```

This will create a field like
("deposit", "<part type>_<field_name>_<method>")
that is defined for every cell.

Conversely, you can sample the grid at particle location (super useful with tracers!)

```
ds.add_mesh_sampling_particle_field(
    field_name, particle_type
)
```

Ex :
```
ds.add_mesh_... (("gas", "density"),
    "gas_tracer")
ad = ds.all_data()
ad["gas_tracer", "mesh_gas_density"]
# ↑ this is the value of the gas density,
# evaluated at each gas tracer's position
```