ASTP 720 - Homework 2 - Multiwavelength Galactic Structure

Due September 10, 2020

The Structure of Galaxies

The mass of galaxies is dominated by dark matter, which explains many observables in galactic rotation, galaxy cluster interactions, and cosmology. One of the earliest pieces of evidence was that of galaxy rotation curves, in which they are observed to be flat out at some radius, i.e., stars and gas are moving at the same material whether they are at 10 or 20 kpc, for example.

The Navarro-Frenk-White (NFW; 1995c) density profile is a model for how the mass density ρ of a dark matter halo behaves as a function of radius r. They proposed a form

$$\rho(r) \propto \frac{1}{(r/r_s)(1+r/r_s)^2}, \tag{1}$$

where r_s is a characteristic radius that is further parameterized as $r_s = r_{200}/c$, where r_{200} is the "virial radius" where the density reaches 200 times the critical density of the Universe, $\rho_{\rm crit} = 3H_0^2/8\pi G$ (so $200\rho_{\rm crit} = M_{200}/(4\pi r_{200}^3/3)$), and c is a dimensionless "concentration" factor. The higher c is, the smaller r_s is and so the more concentrated the halo.

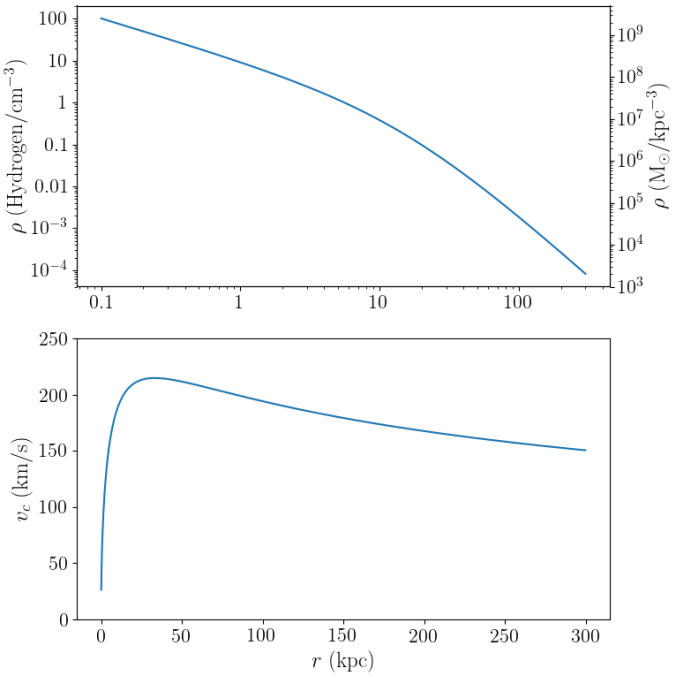
The circular velocity predicted from this fit is given by

$$\left[\frac{v_c(r)}{v_{200}}\right]^2 = \frac{1}{x}\frac{\ln(1+cx) - \frac{cx}{1+cx}}{\ln(1+c) - \frac{c}{1+c}} \tag{2}$$

where v_{200} is the value of the velocity at the radius r_{200} and $x \equiv r/r_{200}$. The circular velocity is related to the mass interior to that radius by equating the gravitational force with the centripetal force, i.e.,

$$v_c(r) = \sqrt{\frac{GM_{\rm enc}(r)}{r}} \quad \rightarrow \quad M_{\rm enc}(r) = \frac{rv_c^2(r)}{G}. \tag{3}$$

And so, the mass of the halo interior to r_{200} sets the circular velocity v_{200}. The two functions might look something like what is shown to the right ($c = 15$, $v_{200} = 160$ km/s, $r_{200} = 230$ kpc).



[1.] Write a numerical calculus library, one that can perform at least one derivative (e.g., the symmetric one) and at least three integration functions inclding the midpoint rule, trapezoidal rule, and Simpson's rule.

[2.] Using your chosen parameters of c, v_{200}, and r_{200}, numerically determine the mass enclosed $M_{\rm enc}(r)$ (and plot), the total mass of the dark matter halo M, and then also $M(r)$, the amount of mass in a little shell around $r \pm \Delta r$ (i.e., what does the mass profile look like?), and $dM(r)/dr$. Compare this by repeating, holding c fixed and changing v_{200}. Again compare the first by repeating, holding v_{200} fixed and changing c. Feel free to use your previous tools from last time if you find that you need to (though I don't think you should); in the future you'll be able to use built-in functions for those.

[3.] In preparation for an actual astrophysical calculation moved into the next homework, write a matrix library that includes a `Matrix` class. There's lots of material online on how to write a class. Your class should be able to:

1. add two matrices together (you can overload `__add__()`, see below),
2. multiply two matrices together (again, can use `__mult__`)
3. transpose a matrix,
4. invert a matrix,
5. calculate the trace of a matrix,
6. calculate the determinant of a matrix,
7. return the LU decomposition of a matrix (should return two `Matrix` objects representing L and U).

You may find it useful to also define some function that returns a single element i, j, swap rows, etc., whatever else you think might be useful but don't feel that you must. Internally, feel free to use `numpy`'s `np.array` or even `np.matrix` if you so choose, but obviously don't just have your `Matrix` class just call the functions to ad, transpose, etc. As usual, please document, make sure to regularly commit to your repository, etc.

Some notes on operator overloading

Operator overloading involves taking known operators, for example the + symbol, and defining or changing how they behave. Documentation for operators can be found here: https://docs.python.org/3/reference/datamodel.html For example, let's say you have a Fraction class that has a structure sort of like below:

```
class Fraction:
  def __init__(self, numer, denom):
    self.numer = numer
    self.denom = denom

  def __add__(self, other):
    # check if other is an int, otherwise a Fraction
    pass

  def __gt__(self, other):
    return float(self.numer)/float(self.denom) >
           float(other.numer)/float(other.denom)

  def __eq__(self, other):
    #need to check if they are the same even in reduced form
    pass
```

In this case, you can define a specific fraction like `Fraction(7, 5)`, which would represent 7/5. The `__init__` method defines how a class is initialized, and in this case you are taking the arguments `numer` and `denom` and storing them inside itself with the `self` argument. By convention, every function inside a class takes an argument called `self` so that a class can reference itself - it need not be called `self`, but again, this is a convention.

In the above, I can overload the + symbol with the `__add__()` function. The way that I've set it up to behave, without actually showing any code, is to have the function have `self` as its first argument, by construction, and then for it to take another variable I'm calling `other`. If `other` is an integer, then one could check for that and add to the `Fraction`. So for example, one could do `Fraction(7, 5) + 2` and we would return inside a new `Fraction` object that would be identical to `Fraction(12, 5)`. In order to check this, I would want to also overload == so that `Fraction(7, 5) + 2 == Fraction(12, 5)` would return `True` - by default this would not be the case because Python would really check if the objects are the same (they aren't) rather than if the underlying *values* are the same. I've also shown the overloading of > with `__gt__()`, in which I have in this case actually performed the logic to test if, say, `Fraction(7, 5) > Fraction(12, 5)`, which should return `False`.

Of course, you can write any other method like you would with any other class without overloading operators. For example, `Fraction` could have a `simplify()` function, where if you did

```
f = Fraction(8, 4)
print(f.simplify())
```

then maybe the code would return `Fraction(2, 1)` or perhaps just `2`, depending on the behavior you want to implement.