PlasmaPy

PlasmaPy is an open source Python package for plasma research and education. After a brief recap of `astropy.units` 🌌, this tutorial will introduce `plasmapy.particles` ⚛️ and `plasmapy.formulary` 🧮.

The next cell contains preliminary imports & settings. To execute a cell in a Jupyter notebook, press `Shift + Enter`. If you need to restart this notebook, please execute the following cell again.

In [ ]:
```
import numpy as np
import astropy.units as u
from astropy import constants as const

from plasmapy.particles import *
from plasmapy.formulary import *

import warnings
warnings.simplefilter(action='ignore')
```

Quick review of Astropy units

PlasmaPy makes heavy use of `astropy.units`. We typically import `astropy.units` as `u`.

In [ ]:
```
import astropy.units as u
```

We can create a physical quantity by multiplying or dividing a number or array with a unit.

In [ ]:
```
60 * u.km
```

This operation creates a `Quantity`: a number, sequence, or array that has been assigned a physical unit. We can create `Quantity` objects with compound units.

In [ ]:
```
V = 88 * u.imperial.mile / u.hour
print(V)
```

Operations between `Quantity` objects handle unit conversions automatically. We can add `Quantity` objects together as long as their units have the same physical type.

In [ ]:
```
1 * u.m + 25 * u.cm
```

Units get handled automatically during operations like multiplication, division, and exponentiation.

In [ ]:
```
(2 * u.m) ** 3
```

The `to()` method allows us to convert a `Quantity` to different units of the same *physical type*. This method accepts strings that represent a unit (including compound units) or a unit object.

In [ ]:
```
V.to("m/s")
```

In [ ]:
```
V.to(u.km / u.hr)
```

Plasma scientists often use the electron-volt (eV) as a unit of temperature 🌡. This is a shortcut for describing the thermal energy per particle, or more accurately the temperature multiplied by the Boltzmann constant, k_B.

Because an electron-volt is a unit of energy rather than temperature, we cannot directly convert electron-volts to kelvin. To handle non-standard unit conversions, `astropy.units` allows the use of equivalencies. The conversion from eV to K can be done by using the `temperature_energy()` equivalency.

In [ ]:
```
(1 * u.eV).to("K", equivalencies=u.temperature_energy())
```

`astropy.constants` provides access the most commonly needed physical constants.

In [ ]:
```
import astropy.constants as const
```

In [ ]:
```
const.c
```

In [ ]:
```
const.m_e
```

Particles

The `plasmapy.particles` subpackage contains functions to access basic particle data, and classes to represent particles.

In [ ]:
```
from plasmapy.particles import *
```

Particle properties

There are several functions that provide information about different particles that might be present in a plasma. The input of these functions is a representation of a particle, such as a string for the atomic symbol or the element name.

In [ ]:
```
atomic_number("Fe")
```

We can provide a number that represents the atomic number.

In [ ]:
```
element_name(26)
```

We can also provide standard symbols or the names of particles.

In [ ]:
```
is_stable("e-")
```

In [ ]:
```
charge_number("proton")
```

The symbols for many particles can even be used directly, such as for an alpha particle ⚛. To create an "α" in a Jupyter notebook, type `\alpha` and press `tab`.

In [ ]:
```
particle_mass("α")
```

We can represent isotopes with the atomic symbol followed by a hyphen and the mass number. Let's use `half_life` to return the half-life of a radioactive particle in seconds as a `Quantity`.

In [ ]:
```
half_life("C-14")
```

We typically represent an ion in a string by putting together the atomic symbol or isotope symbol, a space, the charge number, and the sign of the charge.

In [ ]:
```
charge_number("Fe-56 13+")
```

Functions in `plasmapy.particles` are quite flexible in terms of string inputs representing particles. An input is *particle-like* if it can be used to represent a physical particle.

In [ ]:
```
particle_mass("iron-56 +13")
```

In [ ]:
```
particle_mass("iron-56+++++++++++++")
```

Most of these functions take additional arguments, with `Z` representing the charge number of an ion and `mass_numb` representing the mass number of an isotope. These arguments are *keyword-only* to avoid ambiguity.

In [ ]:
```
particle_mass("Fe", Z=13, mass_numb=56)
```

Particle objects

Up until now, we have been using functions that accept representations of particles and then return particle properties. With the `Particle` class, we can create objects that represent physical particles.

In [ ]:
```
proton = Particle("p+")
electron = Particle("electron")
iron56_nuclide = Particle("Fe", Z=26, mass_numb=56)
```

Particle properties can be accessed via attributes of the `Particle` class.

In [ ]:
```
proton.mass
```

In [ ]:
```
electron.charge
```

In [ ]:
```
electron.charge_number
```

In [ ]:
```
iron56_nuclide.binding_energy
```

Antiparticles

We can get antiparticles of fundamental particles by using the `antiparticle` attribute of a `Particle`.

In [ ]:
```
electron.antiparticle
```

We can also use the tilde (~) operator on a `Particle` to get its antiparticle.

In [ ]:
```
~proton
```

Ionization and recombination

The `recombine()` and `ionize()` methods of a `Particle` representing an ion or neutral atom will return a different `Particle` with fewer or more electrons.

In [ ]:
```
deuterium = Particle("D 0+")
deuterium.ionize()
```

When provided with a number, these methods tell how many bound electrons to add or remove.

In [ ]:
```
alpha = Particle("alpha")
alpha.recombine(2)
```

If the `inplace` keyword is set to `True`, then the `Particle` will be replaced with the new particle.

In [ ]:
```
argon = Particle("Ar 0+")
argon.ionize(inplace=True)
print(argon)
```

Custom particles

Sometimes we want to use a particle with custom properties. For example, we might want to represent an average ion in a multi-species plasma. For that we can use `CustomParticle`.

In [ ]:
```
cp = CustomParticle(9e-26 * u.kg, 2.18e-18 * u.C, symbol="Fe 13.6+")
```

Many of the attributes of `CustomParticle` are the same as in `Particle`.

In [ ]:
```
cp.mass
```

In [ ]:
```
cp.charge
```

In [ ]:
```
cp.symbol
```

If we do not include one of the physical quantities, it gets set to `numpy.nan` (not a number) in the appropriate units.

In [ ]:
```
CustomParticle(9.27e-26 * u.kg).charge
```

`CustomParticle` objects can be used with many of the functions in `plasmapy.formulary`, with greater compatibility expected in the future.

Particle lists

The `ParticleList` class is a container for `Particle` and `CustomParticle` objects.

In [ ]:
```
iron_ions = ParticleList(["Fe 12+", "Fe 13+", "Fe 14+"])
```

By using a `ParticleList`, we can access the properties of multiple particles at once.

In [ ]:
```
iron_ions.mass
```

In [ ]:
```
iron_ions.charge
```

In [ ]:
```
iron_ions.symbols
```

We can also create a `ParticleList` by adding `Particle` and/or `CustomParticle` objects together.

In [ ]:
```
proton + electron
```

We can also get an average particle.

In [ ]:
```
iron_ions.average_particle()
```

`ParticleList` objects are also compatible with many of the functions in `plasmapy.formulary`, with improvements likely in the future.

Particle categorization

The `categories` attribute of a `Particle` provides a set of the categories that the `Particle` belongs to.

In [ ]:
```
muon = Particle("muon")
muon.categories
```

The `is_category()` method lets us determine if a `Particle` belongs to one or more categories.

In [ ]:
```
muon.is_category("lepton")
```

If we need to be more specific, we can use the `require` keyword for categories that a `Particle` must belong to, the `exclude` keyword for categories that the `Particle` cannot belong to, and the `any_of` keyword for categories of which a `Particle` needs to belong to at least one.

In [ ]:
```
electron.is_category(require="lepton", exclude="baryon", any_of={"boson", "f
```

Nuclear reactions

We can use `plasmapy.particles` to calculate the energy of a nuclear reaction using the `>` operator.

In [ ]:
```
deuteron = Particle("D+")
triton = Particle("T+")
alpha = Particle("α")
neutron = Particle("n")
```

In [ ]:
```
energy = deuteron + triton > alpha + neutron
```

In [ ]:
```
energy.to("MeV")
```

If the nuclear reaction is invalid, then an exception is raised with an error message that says why.

In [ ]:
```
deuteron + triton > alpha + 3 * neutron
```

PlasmaPy formulary

The `plasmapy.formulary` subpackage contains a broad variety of formulas needed by plasma scientists across disciplines, in particular to calculate plasma parameters.

In [ ]:
```
from plasmapy.formulary import *
```

Plasma beta in the solar atmosphere

Plasma beta (β) is one of the most fundamental plasma parameters. β is the ratio of the plasma (gas) pressure to the magnetic pressure. How a plasma behaves depends strongly on β. When $\beta \gg 1$, the magnetic field is not strong enough to exert much of a force on the plasma, so its motions start to resemble a gas. When $\beta \ll 1$, magnetic tension and pressure are the dominant macroscopic forces.

Let's use `plasmapy.formulary` to calculate plasma β in different regions of the solar atmosphere and see what we can learn.

Solar corona

Let's start by defining some plasma parameters for an active region in the solar corona.

In [ ]:
```
B_corona = 50 * u.G
n_corona = 1e9 * u.cm ** -3
T_corona = 1 * u.MK
```

When we use these parameters in `beta`, we find that β is quite small so that the corona is *magnetically dominated*.

In [ ]:
```
beta(T_corona, n_corona, B_corona)
```

Solar photosphere

Let's specify some characteristic plasma parameters for the solar photosphere, away from any sunspots.

In [ ]:
```
T_photosphere = 5800 * u.K
B_photosphere = 400 * u.G
n_photosphere = 1e17 * u.cm ** -3
```

When we calculate β for the photosphere, we find that it is an order of magnitude larger than 1, so plasma pressure forces are more important than magnetic tension and pressure.

In [ ]:
```
beta(T_photosphere, n_photosphere, B_photosphere)
```

Plasma parameters in Earth's magnetosphere

The *Magnetospheric Multiscale Mission* (*MMS*) is a constellation of four identical spacecraft. The goal of *MMS* is to investigate the small-scale physics of magnetic reconnection in Earth's magnetosphere. In order to do this, the spacecraft need to orbit in a tight configuration. But how tight does the tetrahedron have to be? Let's use `plasmapy.formulary` to find out.

Physics background

Magnetic reconnection is the fundamental plasma process that converts stored magnetic energy into kinetic energy, thermal energy, and particle acceleration. Reconnection powers solar flares and is a key component of geomagnetic storms in Earth's magnetosphere. Reconnection can also degrade confinement in fusion devices such as tokamaks.

The **inertial length** for a particle is the characteristic length scale for getting accelerated or decelerated by forces in a plasma.

When the reconnection layer thickness is shorter than the **ion inertial length**, $d_i \equiv c/\omega_{pi}$, collisionless effects and the Hall effect enable reconnection to be **fast** (Zweibel & Yamada 2009). The inner electron diffusion region has a thickness of about the **electron inertial length**, $d_e \equiv c/\omega_{pe}$. (Here, ω_{pi} and ω_{pe} are the ion and electron plasma frequencies.)

Our goal: calculate d_i and d_e to get an idea of how far the MMS spacecraft should be separated from each other to investigate reconnection.

Length scales

Let's choose some characteristic plasma parameters for the magnetosphere.

In [ ]:
```
n = 1 * u.cm ** -3
B = 5 * u.nT
T = 10 ** 4.5 * u.K
```

Let's calculate the ion inertial length, d_i. On length scales shorter than d_i, the Hall effect becomes important as the ions and electrons decouple from each other.

In [ ]:
```
inertial_length(n, "p+").to("km")
```

The reconnection regions should therefore be a few hundred kilometers thick. Let's calculate the electron inertial length next.

In [ ]:
```
inertial_length(n, "e-").to("km")
```

The electron diffusion region should therefore have a characteristic length scale of a few kilometers, which is significantly smaller than the ion diffusion region.

We can also calculate the gyroradii for different particles. In the most recent version of PlasmaPy, we can calculate the gyroradii for multiple particles at the same time.

In [ ]:
```
gyroradius(B, ["e-", "p+"], T=T).to("km")
```

The four *MMS* spacecraft have separations of ten to hundreds of kilometers, and thus are well-positioned to investigate reconnection in the magnetosphere.

Frequencies

We can also calculate some of the fundamental frequencies associated with magnetospheric plasma.

In [ ]:
```
plasma_frequency(n, "p+")
```

In [ ]:
```
plasma_frequency(n, "e-")
```

In [ ]:
```
gyrofrequency(B, "p+")
```

In [ ]:
```
gyrofrequency(B, "e-")
```

In [ ]:
```
lower_hybrid_frequency(B, n, "p+")
```