plasmapy-tutorial-completed

October 27, 2023

1 PlasmaPy Tutorial

Thank you for coming to this interactive tutorial. We are excited for you to be here!

PlasmaPy is an open source Python package for plasma research and education. We will start off today by reviewing `astropy.units` , we'll go through some interactive examples of using `plasmapy.particles` and `plasmapy.formulary` .

Let's start with some preliminary imports & settings. To execute a cell in a Jupyter notebook, press `Shift + Enter`. Let's do that for the next cell.

If using Google Colab, click ***Run anyway*** when prompted, and then ***Restart runtime*** when the installation finishes.

[ ]:
```
import sys

if 'google.colab' in str(get_ipython()):
    if 'plasmapy' not in sys.modules:
        !pip install plasmapy==2023.10.0 requests==2.27.1

import numpy as np
import astropy.units as u
from astropy import constants as const
from plasmapy.particles import *
from plasmapy.formulary import *
```

1.1 Astropy units

PlasmaPy makes heavy use of `astropy.units`, which is my favorite part of the scientific pythoniverse! We typically import this subpackage as `u`.

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

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

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

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

[ ]:
```
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.

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

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

[ ]:
```
(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.

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

[ ]:
```
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.

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

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

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

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

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

1.2 Particles

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

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

1.2.1 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.

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

We can provide a number that represents the atomic number.

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

We can provide standard symbols or the names of particles.

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

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

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`.

[ ]:
```
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.

[ ]:
```
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.

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

[ ]:
```
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 often keyword-only to avoid ambiguity.

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

1.2.2 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.

[ ]:
```
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.

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

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

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

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

1.2.3 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 or a dust particle. For that we can use `CustomParticle`.

[ ]:
```
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`.

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

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

[ ]:
```
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.

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

`CustomParticle` objects can provided to most of the commonly used functions in `plasmapy.formulary`, and we're planning to improve interoperability in future releases of PlasmaPy.

1.2.4 Particle lists

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

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

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

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

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

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

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

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

We can also get an average particle.

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

`ParticleList` objects can also be provided to the most commonly used functions in `plasmapy.formulary`, with more complete interoperability expected in the future.

1.2.5 Nuclear reactions

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

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

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

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

If the nuclear reaction is invalid, then an exception is raised that states the reason why.

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

1.3 PlasmaPy formulary

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

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

1.3.1 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** is the characteristic length scale for a particle to get accelerated or decelerated by electromagnetic 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**, $d_e \equiv c/\omega_{pi}$. (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.

1.3.2 Length scales

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

[ ]:
```
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.

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

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

[ ]:
```
inertial_length(n=n, particle="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

[ ]:
```
gyroradius(B=B, particle=["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 Hall physics during reconnection in the magnetosphere.

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

[ ]:
```
plasma_frequency(n=n, particle=["p+", "e-"])
```

[ ]:
```
gyrofrequency(B=B, particle=["p+", "e-"])
```

[ ]:
```
lower_hybrid_frequency(B=B, n_i=n, ion="p+", to_hz=True)
```

Most of the functions in `plasmapy.formulary` have descriptions of the plasma parameters that include the formula and physical interpretation. If we ever forget what the `lower_hybrid_frequency` represents, we can check out its documentation page!

1.4 Final thoughts

Thank you for coming to this tutorial! If you'd like to learn more about PlasmaPy's current capabilities, please check out PlasmaPy's online documentation.

PlasmaPy is a community-developed project, and we invite you to contribute! Please check out our contributor guide to learn more, including the pages on getting ready to contribute and the code contribution workflow. We've also labeled good first issues for new contributors. Possibilities include improving the documentation for a plasma parameter or adding a new example Jupyter notebook.

If there is a feature that you would really like added to PlasmaPy, or if you find a bug or a place that the docs could be improved, please raise an issue! We deeply appreciate it, and doing so helps guide the future of the project.

We thank you once again!