## Contents

1 Getting Started 3  
2 User Documentation  11  
3 Development Guide  221  
4 Project Details  237  
5 Index  247  
   Bibliography  249  
   Python Module Index  251  
   Index  253  

PlasmaPy is an open source community-developed core Python 3.6+ package for plasma physics currently under development.
1.1 Installing PlasmaPy

**Note:** If you would like to contribute to PlasmaPy, please refer to the instructions on *installing PlasmaPy for development*.

1.1.1 Requirements

PlasmaPy requires Python version 3.6 or newer, and is not compatible with Python 2.7. PlasmaPy requires the following packages for installation:

- NumPy 1.14 or newer
- SciPy 0.19 or newer
- Astropy 3.1 or newer
- colorama 0.3 or newer

PlasmaPy also uses the following optional dependencies:

- matplotlib 2.0 or newer
- h5py 2.8 or newer
- mpmath 1.0 or newer
- lmfit 0.9.7 or newer

PlasmaPy can be installed with all of the optional dependencies via `pip install plasmapy[optional]`. 
1.1.2 Creating a conda environment

We highly recommend installing PlasmaPy from a Python environment created using conda. Conda allows us to create and switch between Python environments that are isolated from each other and the system installation (in contrast to this xkcd), while also simplifying the distribution of binary and compiled dependencies.

After installing conda, create a PlasmaPy environment by running:

```bash
conda create -n plasmapy python=3.7 numpy scipy astropy matplotlib h5py lmfit mpmath
  →colorama -c conda-forge
```

To activate this environment, run:

```bash
conda activate plasmapy
```

Once the environment is activated, then you may proceed with installation.

1.1.3 Installation

**Note:** We recommend that new users *create a conda environment*.

**Installation with pip**

To install the most recent release of PlasmaPy on PyPI with pip into an existing Python environment alongside all optional dependencies, run

```bash
pip install plasmapy[optional]
```

To install a minimal set of dependencies (which does not guarantee that everything will run and may result in `ImportError`'s, skip `[all]` and run simply

```bash
pip install plasmapy
```

**Installation with conda**

You can get PlasmaPy from conda via

```bash
conda install -c conda-forge plasmapy
```

1.1.4 Building and installing from source code

**Prerequisites**

Building PlasmaPy from source requires a C compiler such as gcc so that code generated by Cython may be compiled.
Obtaining source code

Stable release

The source code for the most recent stable release of PlasmaPy can be downloaded from PyPI.

Development version on GitHub

If you have git installed on your computer, you may clone PlasmaPy’s GitHub repository and access source code from the most recent development version by running:

\texttt{git clone https://github.com/PlasmaPy/PlasmaPy.git}

The above command uses HTTPS, which is the default protocol and more straightforward to set up. If you have set up an SSH key, then an equivalent and more secure command is:

\texttt{git clone git@github.com:PlasmaPy/PlasmaPy.git}

If you do not have git installed on your computer, then you may download the most recent source code from PlasmaPy’s GitHub repository by selecting “Clone or Download”, which will give you the option to download a zip file.

Note: The \textit{Contributing to PlasmaPy} guide has instructions on how to fork a repository so that you may make pull requests.

Building and installing

In the top level directory, run

\texttt{python setup.py install}

or

\texttt{pip install .}

1.2 Feedback and communication

1.2.1 Matrix chat

The primary communication channel for PlasmaPy is our Matrix room (which is bridged to Gitter).

If you have any questions, the quickest way to get a response is to ask on our Matrix/ channel.

1.2.2 Discourse group

PlasmaPy’s Discourse group is a great place to suggest ideas, bring up discussion topics, and ask questions in threaded public discussions.
1.2.3 Mailing list

We also have a mailing list that serves as a less volatile discussion forum.

1.2.4 Suggestion box

We have a suggestion box if you would like to (optionally anonymously) suggest a feature/topic for consideration. These will be reposted on the mailing list or directly in GitHub issues, as appropriate, for further discussion.

1.2.5 Weekly community meetings

We have approximately weekly community meetings in the PlasmaPy room on Jitsi. The schedule of our community meetings is on our calendar, and you may access the minutes and agendas. Any last minute changes will be discussed on Matrix. As of November 2019, our meetings are on Tuesdays at 19:00 UTC.

1.3 Contributing to PlasmaPy

There are numerous ways to contribute to PlasmaPy, including by providing code and documentation, suggesting and discussing ideas, submitting issues and bug reports, and engaging the broader plasma physics community.

1.3.1 Impostor syndrome disclaimer

We want your help. No, really.

There may be a little voice inside your head that is telling you that you’re not ready to be an open source contributor; that your skills aren’t nearly good enough to contribute. What could you possibly offer a project like this one?

We assure you - the little voice in your head is wrong. If you can write code at all, you can contribute code to open source. Contributing to open source projects is a fantastic way to advance one’s coding skills. Writing perfect code isn’t the measure of a good developer (that would disqualify all of us!); it’s trying to create something, making mistakes, and learning from those mistakes. That’s how we all improve, and we are happy to help others learn.

Being an open source contributor doesn’t just mean writing code, either. You can help out by writing documentation, tests, or even giving feedback about the project (and yes - that includes giving feedback about the contribution process). Some of these contributions may be the most valuable to the project as a whole, because you’re coming to the project with fresh eyes, so you can see the errors and assumptions that seasoned contributors have glossed over.

1.3.2 Contributing code or documentation to PlasmaPy

If you see something you’d like to work on amongst our issues, start hacking away on that! However, please announce your intent first in the relevant issue to make sure there is no work duplication.

Please note that PlasmaPy has a PlasmaPy Community Code of Conduct.

Issues marked by the community as help wanted mean just that - either they’re good contributions for outsiders or there’s an issue in the ongoing work that requires a second opinion. Please consider these first!

Work on PlasmaPy is done via GitHub, so you’ll need a (free) account. If you are new to git, helpful resources include documentation on git basics and an interactive git tutorial. You must also install git locally on your computer.

---

1 The impostor syndrome disclaimer was originally written by Adrienne Lowe for a PyCon talk. It was adapted in the README files for MetPy and yt, and was then adapted by PlasmaPy.
We highly recommend getting reasonably familiar with git by going through these tutorials or a Software Carpentry workshop prior to making code contributions. Do note that you can usually find help in the PlasmaPy Matrix chatroom. For actual guidelines for working on PlasmaPy, please see our PlasmaPy Development Guide.

### 1.3.3 Towncrier changelog entries

Every pull request should include a changelog entry. Please see changelog/README.rst for instructions.

To summarize, put a file like `<PULL REQUEST>.<TYPE>.rst`, where `<PULL REQUEST>` is a pull request number, and `<TYPE>` is one of breaking, feature, bugfix, doc, removal, trivial. If unsure, ask a maintainer.

Footnotes

#### 1.4 PlasmaPy Community Code of Conduct

The PlasmaPy community strives to follow the best practices in open source software development. New contributors are encouraged to join the team and contribute to the codebase. We anticipate/encourage a global participation from people with diverse backgrounds, skills, interests, and opinions. We believe that such diversity is critical in ensuring a growth of ideas in our community.

**1.4.1 Our Pledge**

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

**1.4.2 Our Standards**

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

We as a community pledge to abide by the following guidelines:

- We pledge to treat all people with respect and provide a harassment- and bullying-free environment, regardless of age, sex, sexual orientation and/or gender identity, disability, physical appearance, body size, race, nationality, ethnicity, religion, and level of experience. In particular, sexual language and imagery, sexist, racist, or otherwise exclusionary jokes are not appropriate.
- We pledge to respect the work of others by recognizing acknowledgment/citation requests of original authors. As authors, we pledge to be explicit about how we want our own work to be cited or acknowledged.
- We pledge to welcome those interested in joining the community, and realize that including people with a variety of opinions and backgrounds will only serve to enrich our community. In particular, discussions relating to pros/cons of various technologies, programming languages, and so on are welcome, but these should be done
with respect, taking proactive measure to ensure that all participants are heard and feel confident that they can freely express their opinions.

- We pledge to welcome questions and answer them respectfully, paying particular attention to those new to the community. We pledge to provide respectful criticisms and feedback in forums, especially in discussion threads resulting from code contributions.

- We pledge to be conscientious of the perceptions of the wider community and to respond to criticism respectfully. We will strive to model behaviors that encourage productive debate and disagreement, both within our community and where we are criticized. We will treat those outside our community with the same respect as people within our community.

- We pledge to work from the very beginning of this project to make PlasmaPy accessible to people with disabilities.

- We pledge to help the entire community follow these guidelines, and to not remain silent when we see violations of them. We will take action when members of our community violate these guidelines. Members of the PlasmaPy community may contact any member of the Coordinating Committee to report violations. Members of the Coordinating Committee will treat these reports in the strictest confidence. The Coordinating Committee will develop formal procedures for how to handle reported violations.

### 1.4.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

### 1.4.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

### 1.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting Nick Murphy at namurphy@cfa.harvard.edu or any member of the Coordinating Committee. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Project team members should recuse themselves from enforcement of the code of conduct for a given incident if they have an actual or apparent conflict of interest. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project’s leadership.

### 1.4.6 Attribution

Parts of these guidelines have been adapted from the Contributor Covenant (version 1.4), the Astropy Community Code of Conduct, and the Python Software Foundation code of conduct.
1.5 Acknowledging and Citing PlasmaPy

If you use PlasmaPy for a project resulting in a publication, we ask that you cite both the specific version of PlasmaPy used in your project and an informational reference. Including these references provides credit to developers of PlasmaPy and enables greater scientific reproducibility.

Version 0.2.0 of PlasmaPy may be cited using the following reference.


The current standard informational reference for PlasmaPy is


These references may be made by adding the following line to the methods or acknowledgements sections of a paper.

• This research made use of PlasmaPy version 0.2.0, a community-developed open source Python package for plasma physics (PlasmaPy Community et al. 2018, 2019).

All public releases of PlasmaPy are openly archived in the PlasmaPy Community on Zenodo.

We encourage authors to acknowledge the packages that PlasmaPy depends on, including but not limited to Astropy, NumPy, and SciPy.

• PlasmaPy’s GitHub repository
• PlasmaPy website
• Using astropy.units
2.1 Formulary

The formulary subpackage aims to cover the NRL Plasma Physics Formulary.

2.1.1 Formulary (plasmapy.formulary)

plasmapy.formulary provides theoretical formulas for calculation of physical quantities helpful for plasma physics.

Mathematics (plasmapy.formulary.mathematics)

This module gathers highly theoretical mathematical formulas relevant to plasma physics. Usually, those are used somewhere else in the code but were deemed general enough for the mathematical apparatus to be abstracted from the main function interface.

Functions

Fermi_integral(x, int, complex, ...) Calculate the complete Fermi-Dirac integral.

Fermi_integral

plasmapy.formulary.mathematics.Fermi_integral(x: Union[Union[float, int, complex, numpy.ndarray], j: Union[float, int, complex, numpy.ndarray]]) → Union[float, complex, numpy.ndarray]

Calculate the complete Fermi-Dirac integral.

Parameters
• **x** *(float, int, complex, or ndarray)* – Argument of the Fermi-Dirac integral function.

• **j** *(float, int, complex, or ndarray)* – Order/index of the Fermi-Dirac integral function.

**Returns**  
integral – Complete Fermi-Dirac integral for given argument and order.

**Return type**  
float, complex, or ndarray

**Raises**

• **TypeError** – If the argument is invalid.

• **UnitsError** – If the argument is a `Quantity` but is not dimensionless.

• **ValueError** – If the argument is not entirely finite.

**Notes**

The complete Fermi-Dirac integral is defined as:

\[ F_j(x) = \frac{1}{\Gamma(j + 1)} \int_0^\infty \frac{t^j}{\exp(t - x) + 1} dt \]

for \( j > 0 \).

This is equivalent to the following polylogarithm function:

\[ F_j(x) = -Li_{j+1}(-e^x) \]

Warning: at present this function is limited to relatively small arguments due to limitations in the `mpmath` package’s implementation of polylog.

**Examples**

```python
>>> Fermi_integral(0, 0)
(0.6931471805599453-0j)
>>> Fermi_integral(1, 0)
(1.3132616875182228-0j)
>>> Fermi_integral(1, 1)
(1.8062860704447743-0j)
```

**Examples:**

• *The plasma dispersion function*

**Dielectric functions** *(plasmapy.formulary.dielectric)*

Functions to calculate plasma dielectric parameters

**Functions**
cold_plasma_permittivity_SDP

Magnetized Cold Plasma Dielectric Permittivity Tensor Elements.

Elements (S, D, P) are given in the “Stix” frame, ie. with B // z.
The \( \exp(-i\omega t) \) time-harmonic convention is assumed.

Parameters

- \( B \) (Quantity) – Magnetic field magnitude in units convertible to tesla.
- \( \text{species} \) (list of str) – List of the plasma particle species e.g.: [’e’, ‘D+’] or [’e’, ‘D+’, ‘He+’].
- \( n \) (list of astropy.units.Quantity) – list of species density in units convertible to per cubic meter. The order of the species densities should follow species.
- \( \omega \) (Quantity) – Electromagnetic wave frequency in rad/s.

Returns

- \( \text{sum} \) (astropy.units.Quantity) – S (“Sum”) dielectric tensor element.
- \( \text{difference} \) (astropy.units.Quantity) – D (“Difference”) dielectric tensor element.
- \( \text{plasma} \) (astropy.units.Quantity) – P (“Plasma”) dielectric tensor element.

Notes

The dielectric permittivity tensor is expressed in the Stix frame with the \( \exp(-i\omega t) \) time-harmonic convention as \( \varepsilon = \varepsilon_0 A \), with \( A \) being

\[
\varepsilon = \varepsilon_0 \begin{pmatrix}
S & -iD & 0 \\
+iD & S & 0 \\
0 & 0 & P
\end{pmatrix}
\]

where:

\[
S = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega^2 - \Omega_{c,s}^2}
\]
\[
D = \sum_s \frac{\Omega_{c,s} \omega_{p,s}^2}{\omega^2 - \Omega_{c,s}^2}
\]
\[
P = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega^2}
\]

where \( \omega_{p,s} \) is the plasma frequency and \( \Omega_{c,s} \) is the signed version of the cyclotron frequency for the species \( s \).
References


Examples

```python
>>> from astropy import units as u
>>> from numpy import pi

>>> B = 2*u.T
>>> species = ['e', 'D+']
>>> n = [1e18*u.m**-3, 1e18*u.m**-3]
>>> omega = 3.7e9*(2*pi)*(u.rad/u.s)

>>> permittivity = S, D, P = plasmapy.formulary.dielectric.cold_plasma_permittivity_SDPS(DP, species, n, omega)

>>> S
<Quantity 1.02422...>

>>> permittivity.sum  # namedtuple-style access
<Quantity 1.02422...>

>>> D
<Quantity 0.39089...>

>>> P
<Quantity -4.8903...>
```

cold_plasma_permittivity_LRP

`plasmapy.formulary.dielectric.cold_plasma_permittivity_LRP` *(B: Unit("T"), species, n, omega: Unit("rad / s"))*

Magnetized Cold Plasma Dielectric Permittivity Tensor Elements.

Elements (L, R, P) are given in the “rotating” basis, ie. in the basis `(u+ , u-, u_z)`, where the tensor is diagonal and with B // z.

The `exp(-iωt)` time-harmonic convention is assumed.

Parameters

- **B** *(Quantity)* – Magnetic field magnitude in units convertible to tesla.
- **species** *(list of str)* – The plasma particle species (e.g.: ['e', 'D+] or ['e', 'D+', 'He+']).
- **n** *(list of ~astropy.units.Quantity)* – list of species density in units convertible to per cubic meter. The order of the species densities should follow species.
- **omega** *(Quantity)* – Electromagnetic wave frequency in rad/s.

Returns

- **left** *(~astropy.units.Quantity)* – L (“Left”) Left-handed circularly polarization tensor element.
- **right** *(~astropy.units.Quantity)* – R (“Right”) Right-handed circularly polarization tensor element.
- **plasma** *(~astropy.units.Quantity)* – P (“Plasma”) dielectric tensor element.
Notes

In the rotating frame defined by \((u_+, u_-, u_z)\) with \(u_\pm = (u_x \pm u_y)/\sqrt{2}\), the dielectric tensor takes a diagonal form with elements \(L\), \(R\), \(P\) with:

\[
L = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega(\omega - \Omega_{c,s})}
\]

\[
R = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega(\omega + \Omega_{c,s})}
\]

\[
P = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega^2}
\]

where \(\omega_{p,s}\) is the plasma frequency and \(\Omega_{c,s}\) is the signed version of the cyclotron frequency for the species \(s\).

References


Examples

```python
>>> from astropy import units as u
>>> from numpy import pi
>>> B = 2*u.T
>>> species = ['e', 'D+']
>>> n = [1e18*u.m**-3, 1e18*u.m**-3]
>>> omega = 3.7e9*(2*pi)*(u.rad/u.s)
>>> L, R, P = permittivity = cold_plasma_permittivity_LRP(B, species, n, omega)
>>> L
<Quantity 0.63333...>
>>> permittivity.left
# namedtuple-style access
<Quantity 0.63333...>
>>> R
<Quantity 1.41512...>
>>> P
<Quantity -4.8903...>
```

`permittivity_1D_Maxwellian`

`plasmapy.formulary.dielectric.permittivity_1D_Maxwellian`(omega: Unit("rad / s"), kWave: Unit("rad / m"), T: Unit("K"), n: Unit("1 / m^3"), particle, z_mean: Unit(dimensionless) = None) -> Unit(dimensionless)

The classical dielectric permittivity for a 1D Maxwellian plasma. This function can calculate both the ion and electron permittivities. No additional effects are considered (e.g. magnetic fields, relativistic effects, strongly coupled regime, etc.)

Parameters
• **omega** *(Quantity)* – The frequency in rad/s of the electromagnetic wave propagating through the plasma.

• **kWave** *(Quantity)* – The corresponding wavenumber, in rad/m, of the electromagnetic wave propagating through the plasma. This is often modulated by the dispersion of the plasma or by relativistic effects. See em_wave.py for ways to calculate this.

• **T** *(Quantity)* – The plasma temperature - this can be either the electron or the ion temperature, but should be consistent with density and particle.

• **n** *(Quantity)* – The plasma density - this can be either the electron or the ion density, but should be consistent with temperature and particle.

• **particle** *(str)* – The plasma particle species.

• **z_mean** *(str)* – The average ionization of the plasma. This is only required for calculating the ion permittivity.

**Returns chi** – The ion or the electron dielectric permittivity of the plasma. This is a dimensionless quantity.

**Return type** Quantity

**Notes**

The dielectric permittivities for a Maxwellian plasma are described by the following equations\(^1\)

\[
\chi_e(k, \omega) = -\frac{\alpha^2}{2} Z'(x_e)
\]

\[
\chi_i(k, \omega) = -\frac{\alpha^2}{2} Z'(x_i)
\]

\[
\alpha = \frac{\omega_p}{k v_{Th}}
\]

\[
x = \frac{\omega}{k v_{Th}}
\]

\(\chi_e\) and \(\chi_i\) are the electron and ion permittivities respectively. \(Z'\) is the derivative of the plasma dispersion function. \(\alpha\) is the scattering parameter which delineates the difference between the collective and non-collective Thomson scattering regimes. \(x\) is the dimensionless phase velocity of the EM wave propagating through the plasma.

**References**

**Example**

```python
>>> from astropy import units as u
>>> from numpy import pi
>>> from astropy.constants import c
>>> T = 30 * 11600 * u.K
>>> n = 1e18 * u.cm**-3
>>> particle = 'Ne'
>>> z_mean = 8 * u.dimensionless_unscaled
>>> vTh = parameters.thermal_speed(T, particle, method="most_probable")
>>> omega = 5.635e14 * 2 * pi * u.rad / u.s
```

>>> kWave = omega / vTh
>>> permittivity_1D_Maxwellian(omega, kWave, T, n, particle, z_mean)
<Quantity -6.72809\ldots e-08+5.76037\ldots e-07j>

Examples:

• Cold Magnetized Plasma Waves Tensor Elements (S, D, P in Stix’s notation)

**Dimensionless parameters** (**plasmapy.formulary.dimensionless**)

Module of dimensionless plasma parameters.

These are especially important for determining what regime a plasma is in. (e.g., turbulent, quantum, collisional, etc.).

For example, plasmas at high (much larger than 1) Reynolds numbers are highly turbulent, while turbulence is negligible at low Reynolds numbers.

**Functions**

```python
beta(T, n, B)
quantum_theta(T, n_e)
```

**beta**

**plasmapy.formulary.dimensionless.beta** *(T: Unit("K"), n: Unit("1 / m3"), B: Unit("T")) -> Unit(dimensionless)*

The ratio of thermal pressure to magnetic pressure.

**Parameters**

• **T** *(Quantity)* – The temperature of the plasma.
• **n** *(Quantity)* – The particle density of the plasma.
• **B** *(Quantity)* – The magnetic field in the plasma.

**Examples**

```python
>>> import astropy.units as u
>>> beta(1*u.eV, 1e20*u.m**-3, 1*u.T)
<Quantity 4.0267\ldots e-05>
>>> beta(8.8e3*u.eV, 1e20*u.m**-3, 5.3*u.T)
<Quantity 0.01261\ldots>
```

**Returns** **beta** – Dimensionless quantity.

**Return type** **Quantity**
quantum_theta

plasmapy.formulary.dimensionless.quantum_theta(T: Unit("K"), n_e: Unit("1 / m^3")) \rightarrow Unit(dimensionless)

Compares Fermi energy to thermal kinetic energy to check if quantum effects are important.

Parameters

- **T** (*Quantity*) – The temperature of the plasma.
- **n_e** (*Quantity*) – The electron number density of the plasma.

Examples

```python
>>> import astropy.units as u

>>> quantum_theta(1*u.eV, 1e20*u.m**-3)
<Quantity 127290.619...>

>>> quantum_theta(1*u.eV, 1e16*u.m**-3)
<Quantity 59083071...>

>>> quantum_theta(1*u.eV, 1e26*u.m**-3)
<Quantity 12.72906...>

>>> quantum_theta(1*u.K, 1e26*u.m**-3)
<Quantity 0.00109...>
```

Returns theta

Return type *Quantity*

Dispersion relations (*plasmapy.formulary.dispersionfunction*)

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plasma_dispersion_func(zeta, int, float,...)</td>
<td>Calculate the plasma dispersion function.</td>
</tr>
<tr>
<td>plasma_dispersion_func_deriv(zeta, int,...)</td>
<td>Calculate the derivative of the plasma dispersion function.</td>
</tr>
</tbody>
</table>

plasma_dispersion_func

plasmapy.formulary.dispersionfunction.plasma_dispersion_func(zeta:

Calculate the plasma dispersion function.

Parameters **zeta** (*complex, int, float, ndarray, or Quantity*) – Argument of plasma dispersion function.

Returns **Z** – Value of plasma dispersion function.
Return type  complex, float, or ndarray

Raises

• TypeError – If the argument is of an invalid type.
• UnitsError – If the argument is a Quantity but is not dimensionless.
• ValueError – If the argument is not entirely finite.

See also:

plasma_dispersion_func_deriv()

Notes

The plasma dispersion function is defined as:

\[ Z(\zeta) = \pi^{-0.5} \int_{-\infty}^{+\infty} \frac{e^{-x^2}}{x - \zeta} dx \]

where the argument is a complex number [fried.conte-1961].

In plasma wave theory, the plasma dispersion function appears frequently when the background medium has a Maxwellian distribution function. The argument of this function then refers to the ratio of a wave’s phase velocity to a thermal velocity.

References

Examples

```python
>>> plasma_dispersion_func(0)
1.7724538509055159j
>>> plasma_dispersion_func(1j)
0.757872156141312j
>>> plasma_dispersion_func(-1.52+0.47j)
(0.6088888957234254+0.33494583882874024j)
```

plasma_dispersion_func_deriv

plasmapy.formulary.dispersionfunction.plasma_dispersion_func_deriv(zeta: Union[complex, int, float, numpy.ndarray, astropy.units.quantity.Quantity]) -> Union[complex, float, numpy.ndarray, astropy.units.quantity.Quantity]

Calculate the derivative of the plasma dispersion function.

Parameters  

zeta (complex, int, float, ndarray, or Quantity) – Argument of plasma dispersion function.
Returns **Zprime** – First derivative of plasma dispersion function.

**Return type** complex, float, or ndarray

**Raises**
- **TypeError** – If the argument is invalid.
- **UnitsError** – If the argument is a `Quantity` but is not dimensionless.
- **ValueError** – If the argument is not entirely finite.

**See also:**
`plasma_dispersion_func()`

### Notes

The derivative of the plasma dispersion function is defined as:

\[
Z' (\zeta) = \pi^{-1/2} \int_{-\infty}^{+\infty} \frac{e^{-x^2}}{(x - \zeta)^2} dx
\]

where the argument is a complex number [fried.conte-1961].

### Examples

```python
>>> plasma_dispersion_func_deriv(0)
(-2+0j)
>>> plasma_dispersion_func_deriv(1j)
(-0.484255687717376...+0j)
>>> plasma_dispersion_func_deriv(-1.52+0.47j)
(0.165871331498228...+0.445879788059350...j)
```

**Examples:**

- *The plasma dispersion function*

### Distribution functions (**plasmapy.formulary.distribution**)  

Common distribution functions for plasmas, such as the Maxwellean or Kappa distributions. Functionality is intended to include generation, fitting and calculation.

#### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Maxwellian_1D(v, T[, particle, v_drift, ...])</code></td>
<td>Probability distribution function of velocity for a Maxwellian distribution in 1D.</td>
</tr>
<tr>
<td><code>Maxwellian_velocity_2D(vx, vy, T[, ...])</code></td>
<td>Probability distribution function of velocity for a Maxwellian distribution in 2D.</td>
</tr>
<tr>
<td><code>Maxwellian_velocity_3D(vx, vy, vz, T[, ...])</code></td>
<td>Probability distribution function of velocity for a Maxwellian distribution in 3D.</td>
</tr>
</tbody>
</table>
### Maxwellian_1D

**plasmapy.formulary.distribution.Maxwellian_1D**

Returns the probability density function of velocity for a Maxwellian distribution in 1D.

**Parameters**

- **v** (*Quantity*) – The velocity in units convertible to m/s.
- **T** (*Quantity*) – The temperature in Kelvin.
- **particle** (*str, optional*) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for \( \text{He}^+ \))
- **v_drift** (*Quantity, optional*) – The drift velocity in units convertible to m/s.
- **vTh** (*Quantity, optional*) – Thermal velocity (most probable velocity) in m/s. This is used for optimization purposes to avoid re-calculating vTh, for example when integrating over velocity-space.
- **units** (*str, optional*) – Selects whether to run function with units and unit checks (when equal to “units”) or to run as unitless (when equal to “unitless”). The unitless version is substantially faster for intensive computations.

**Returns**

- **f** – Probability density in units of Velocity^-1, normalized so that \( \int_{-\infty}^{+\infty} f(v)dv = 1 \).

**Return type** *Quantity*

**Raises**

- **TypeError** – The parameter arguments are not Quantities and cannot be converted into Quantities.
- **UnitConversionError** – If the parameters are not in appropriate units.
- **ValueError** – If the temperature is negative, or the particle mass or charge state cannot be found.

---

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maxwellian_speed_1D</td>
<td>Probability distribution function of speed for a Maxwellian distribution in 1D</td>
</tr>
<tr>
<td>Maxwellian_speed_2D</td>
<td>Probability distribution function of speed for a Maxwellian distribution in 2D</td>
</tr>
<tr>
<td>Maxwellian_speed_3D</td>
<td>Probability distribution function of speed for a Maxwellian distribution in 3D</td>
</tr>
<tr>
<td>kappa_velocity_1D</td>
<td>Return the probability density at the velocity ( v ) in m/s to find a particle ( \text{particle} ) in a plasma of temperature ( T ) following the Kappa distribution function in 1D.</td>
</tr>
<tr>
<td>kappa_velocity_3D</td>
<td>Return the probability density function for finding a particle with velocity components ( v_x, v_y, \text{and } v_z ) in m/s in a suprathermal plasma of temperature ( T ) and parameter ‘kappa’ which follows the 3D Kappa distribution function.</td>
</tr>
</tbody>
</table>
Notes

In one dimension, the Maxwellian distribution function for a particle of mass $m$, velocity $v$, a drift velocity $V$ and with temperature $T$ is:

$$f = \sqrt{\frac{m}{2\pi k_B T}} e^{-\frac{m(v-V)^2}{2k_B T}} \equiv \frac{1}{\sqrt{\pi v_{th}^2}} e^{-\frac{(v-v_{\text{drift}})^2}{v_{th}^2}}$$

where $v_{th} = \sqrt{2k_B T/m}$ is the thermal speed

Examples

```python
>>> from astropy import units as u
>>> v=1*u.m/u.s
>>> Maxwellian_1D(v=v, T=30000 * u.K, particle='e', v_drift=0 * u.m / u.s)
<Quantity 5.9163...e-07 s / m>
```

Maxwellian_velocity_2D

`plasmapy.formulary.distribution.Maxwellian_velocity_2D(vx, vy, T, particle='e', vx_drift=0, vy_drift=0, vTh=nan, units='units')`

Probability distribution function of velocity for a Maxwellian distribution in 2D.

Return the probability density function for finding a particle with velocity components $vx$ and $vy$ in m/s in an equilibrium plasma of temperature $T$ which follows the 2D Maxwellian distribution function. This function assumes Cartesian coordinates.

Parameters

- $vx$ (*Quantity*) – The velocity in x-direction units convertible to m/s.
- $vy$ (*Quantity*) – The velocity in y-direction units convertible to m/s.
- $T$ (*Quantity*) – The temperature, preferably in Kelvin.
- `particle` (*str, optional*) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for $\text{He}_4^+$ (singly ionized helium-4)), which defaults to electrons.
- $vx_{\text{drift}}$ (*Quantity, optional*) – The drift velocity in x-direction units convertible to m/s.
- $vy_{\text{drift}}$ (*Quantity, optional*) – The drift velocity in y-direction units convertible to m/s.
- $vTh$ (*Quantity, optional*) – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating $vTh$, for example when integrating over velocity-space.
- `units` (*str, optional*) – Selects whether to run function with units and unit checks (when equal to “units”) or to run as unitless (when equal to “unitless”). The unitless version is substantially faster for intensive computations.

Returns $f$ – Probability density in Velocity^-1, normalized so that $\int \int \int_0^\infty f(\vec{v})d\vec{v} = 1$.

Return type *Quantity*

Raises
• **TypeError** – A parameter argument is not a *Quantity* and cannot be converted into a *Quantity*.
• **UnitConversionError** – If the parameters is not in appropriate units.
• **ValueError** – If the temperature is negative, or the particle mass or charge state cannot be found.

**Notes**

In 2D, the Maxwellian velocity distribution function describing the distribution of particles with speed \( v \) in a plasma with temperature \( T \) is given by:

\[
    f = (\pi v_{Th}^2)^{-1} \exp \left[ -\left( \vec{v} - \vec{V}_{drift} \right)^2 / v_{Th}^2 \right]
\]

where \( v_{Th} = \sqrt{2k_BT/m} \) is the thermal speed.

See also:

*Maxwellian_1D()*

**Example**

```python
>>> from astropy import units as u
>>> v=1 * u.m / u.s
>>> Maxwellian_velocity_2D(vx=v, ... vy=v, ... T=30000*u.K, ... particle='e', ... vx_drift=0 * u.m / u.s, ... vy_drift=0 * u.m / u.s)
<Quantity 3.5002...e-13 s2 / m2>
```

**Maxwellian_velocity_3D**

plasmapy.formulary.distribution.Maxwellian_velocity_3D(vx, vy, vz, T, particle='e', vx_drift=0, vy_drift=0, vz_drift=0, vTh=nan, units='units')

Probability distribution function of velocity for a Maxwellian distribution in 3D.

Return the probability density function for finding a particle with velocity components \( vx, vy, \) and \( vz \) in m/s in an equilibrium plasma of temperature \( T \) which follows the 3D Maxwellian distribution function. This function assumes Cartesian coordinates.

**Parameters**

- **vx** (*Quantity*) – The velocity in x-direction units convertible to m/s.
- **vy** (*Quantity*) – The velocity in y-direction units convertible to m/s.
- **vz** (*Quantity*) – The velocity in z-direction units convertible to m/s.
- **T** (*Quantity*) – The temperature, preferably in Kelvin.
- **particle** (*str, optional*) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for \( \text{He}^{+1} \) (singly ionized helium-4)), which defaults to electrons.
• **vx_drift** *(Quantity, optional)* – The drift velocity in x-direction units convertible to m/s.

• **vy_drift** *(Quantity, optional)* – The drift velocity in y-direction units convertible to m/s.

• **vz_drift** *(Quantity, optional)* – The drift velocity in z-direction units convertible to m/s.

• **vTh** *(Quantity, optional)* – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating vTh, for example when integrating over velocity-space.

• **units** *(str, optional)* – Selects whether to run function with units and unit checks (when equal to “units”) or to run as unitless (when equal to “unitless”). The unitless version is substantially faster for intensive computations.

Returns **f** – Probability density in Velocity^-1, normalized so that \( \int \int \int \int_0^\infty f(\vec{v})d\vec{v} = 1 \).

Return type **Quantity**

Raises

• **TypeError** – A parameter argument is not a Quantity and cannot be converted into a Quantity.

• **UnitConversionError** – If the parameters is not in appropriate units.

• **ValueError** – If the temperature is negative, or the particle mass or charge state cannot be found.

Notes

In 3D, the Maxwellian speed distribution function describing the distribution of particles with speed \( v \) in a plasma with temperature \( T \) is given by:

\[
f = (\pi v_{Th}^2)^{-3/2} \exp \left[ -\frac{\vec{v} - \vec{V}_{drift}^2}{v_{Th}^2} \right]
\]

where \( v_{Th} = \sqrt{2k_BT/m} \) is the thermal speed.

See also:

*Maxwellian_1D()*

Example

```python
t>>> from astropy import units as u
t>>> v=1 * u.m / u.s
t>>> Maxwellian_velocity_3D(vx=v,
... vy=v,
... vz=v,
... T=30000 * u.K,
... particle='e',
... vx_drift=0 * u.m / u.s,
... vy_drift=0 * u.m / u.s,
... vz_drift=0 * u.m / u.s)
t<Quantity 2.0708...e-19 s3 / m3>
```
Maxwellian\_speed\_1D

\texttt{plasmapy.formulary.distribution.Maxwellian\_speed\_1D(v, \ T, \ particle='e', \ v\_drift=0, \ vTh=nan, \ units='units')}

Probability distribution function of speed for a Maxwellian distribution in 1D.

Return the probability density function for finding a particle with speed $v$ in m/s in an equilibrium plasma of temperature $T$ which follows the Maxwellian distribution function.

**Parameters**

- $v$ (\texttt{Quantity}) – The speed in units convertible to m/s.
- $T$ (\texttt{Quantity}) – The temperature, preferably in Kelvin.
- \texttt{particle} (\texttt{str}, optional) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for $\text{He}_4^{+1}$ (singly ionized helium-4)), which defaults to electrons.
- $v\_drift$ (\texttt{Quantity}) – The drift speed in units convertible to m/s.
- $vTh$ (\texttt{Quantity}, optional) – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating $v\_Th$, for example when integrating over velocity-space.
- \texttt{units} (\texttt{str}, optional) – Selects whether to run function with units and unit checks (when equal to "units") or to run as unitless (when equal to "unitless"). The unitless version is substantially faster for intensive computations.

**Returns**

- $f$ – Probability density in speed\(^{-1}\), normalized so that $\int_0^\infty f(v)dv = 1$.

**Return type** \texttt{Quantity}

**Raises**

- \texttt{TypeError} – The parameter arguments are not Quantities and cannot be converted into Quantities.
- \texttt{UnitConversionError} – If the parameters is not in appropriate units.
- \texttt{ValueError} – If the temperature is negative, or the particle mass or charge state cannot be found.

**Notes**

In one dimension, the Maxwellian speed distribution function describing the distribution of particles with speed $v$ in a plasma with temperature $T$ is given by:

$$f(v) = 2 \frac{1}{(\pi v_{Th}^2)^{1/2}} \exp\left(-\frac{(v - V_{drift})^2}{v_{Th}^2}\right)$$

where $v_{Th} = \sqrt{2k_B T/m}$ is the thermal speed.

**Example**

```python
>>> from astropy import units as u
>>> v=1 * u.m / u.s
>>> Maxwellian_speed_1D(v=v, T=30000 * u.K, particle='e', v_drift=0 * u.m / u.s)
<Quantity 1.1832...e-06 s / m>
```
Maxwellian_speed_2D

plasmapy.formulary.distribution.Maxwellian_speed_2D(v, T, particle='e', v_drift=0, vTh=nan, units='units')

Probability distribution function of speed for a Maxwellian distribution in 2D.

Return the probability density function of finding a particle with speed components \(v_x\) and \(v_y\) in m/s in an equilibrium plasma of temperature \(T\) which follows the 2D Maxwellian distribution function. This function assumes Cartesian coordinates.

Parameters

- \(v\) (Quantity) – The speed in units convertible to m/s.
- \(T\) (Quantity) – The temperature, preferably in Kelvin.
- \(\text{particle}\) (str, optional) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for \(\text{He}^+_{4}\) (singly ionized helium-4)), which defaults to electrons.
- \(v\text{_drift}\) (Quantity) – The drift speed in units convertible to m/s.
- \(v\text{Th}\) (Quantity, optional) – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating \(v\text{Th}\), for example when integrating over velocity-space.
- \(\text{units}\) (str, optional) – Selects whether to run function with units and unit checks (when equal to “units”) or to run as unitless (when equal to “unitless”). The unitless version is substantially faster for intensive computations.

Returns \(f\) – Probability density in speed^{-1}, normalized so that: \(\int\int\int_{0}^{\infty} f(\vec{v})d\vec{v} = 1\).

Return type Quantity

Raises

- TypeError – A parameter argument is not a Quantity and cannot be converted into a Quantity.
- UnitConversionError – If the parameters is not in appropriate units.
- ValueError – If the temperature is negative, or the particle mass or charge state cannot be found.

Notes

In 2D, the Maxwellian speed distribution function describing the distribution of particles with speed \(v\) in a plasma with temperature \(T\) is given by:

\[
f = 2\pi v (\pi v_{Th}^2)^{-1} \exp\left(-\frac{v^2}{v_{Th}^2}\right)
\]

where \(v_{Th} = \sqrt{2k_B T/m}\) is the thermal speed.

See also:

Maxwellian_speed_1D()
Maxwellian speed 3D

plasmapy.formulary.distribution.Maxwellian_speed_3D(v, T, particle='e', v_drift=0, vTh=nan, units='units')

Probability distribution function of speed for a Maxwellian distribution in 3D.

Return the probability density function for finding a particle with speed components vx, vy, and vz in m/s in an equilibrium plasma of temperature T which follows the 3D Maxwellian distribution function. This function assumes Cartesian coordinates.

Parameters

- v (Quantity) – The speed in units convertible to m/s.
- T (Quantity) – The temperature, preferably in Kelvin.
- particle (str, optional) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for He4+1 (singly ionized helium-4)), which defaults to electrons.
- v_drift (Quantity) – The drift speed in units convertible to m/s.
- vTh (Quantity, optional) – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating vTh, for example when integrating over velocity-space.
- units (str, optional) – Selects whether to run function with units and unit checks (when equal to "units") or to run as unitless (when equal to "unitless"). The unitless version is substantially faster for intensive computations.

Returns f – Probability density in speed^-1, normalized so that: \( \int_{0}^{\infty} f(\vec{v})d\vec{v} = 1 \).

Return type Quantity

Raises

- TypeError – A parameter argument is not a Quantity and cannot be converted into a Quantity.
- UnitConversionError – If the parameters is not in appropriate units.
- ValueError – If the temperature is negative, or the particle mass or charge state cannot be found.

Notes

In 3D, the Maxwellian speed distribution function describing the distribution of particles with speed \( v \) in a plasma with temperature \( T \) is given by:

\[
f = 4\pi \nu^2 (\pi v_{Th}^2)^{-3/2} \exp(-\nu^2/v_{Th}^2)
\]

where \( v_{Th} = \sqrt{2k_B T/m} \) is the thermal speed.

See also:

Maxwellian_speed_1D()
Example

```python
>>> from astropy import units as u
>>> v=1 * u.m / u.s
>>> Maxwellian_speed_3D(v=v, T=30000*u.K, particle='e', v_drift=0 * u.m / u.s)
<Quantity 2.60235...e-18 s / m>
```

**kappa_velocity_1D**

`plasmapy.formulary.distribution.kappa_velocity_1D(v, T, kappa, particle='e', v_drift=0, vTh=nan, units='units')`

Return the probability density at the velocity \( v \) in m/s to find a particle \( \text{particle} \) in a plasma of temperature \( T \) following the Kappa distribution function in 1D. The slope of the tail of the Kappa distribution function is set by ‘kappa’, which must be greater than 1/2.

**Parameters**

- \( v \) (Quantity) – The velocity in units convertible to m/s.
- \( T \) (Quantity) – The temperature in Kelvin.
- \( \text{kappa} \) (Quantity) – The kappa parameter is a dimensionless number which sets the slope of the energy spectrum of suprathermal particles forming the tail of the Kappa velocity distribution function. Kappa must be greater than 3/2.
- \( \text{particle} \) (str, optional) – Representation of the particle species(e.g., 'p for protons, 'D+' for deuterium, or 'He-4 +1' for \( \text{He}^+_4 \) (singly ionized helium-4)), which defaults to electrons.
- \( v\_\text{drift} \) (Quantity, optional) – The drift velocity in units convertible to m/s.
- \( v\text{Th} \) (Quantity, optional) – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating \( v\text{Th} \), for example when integrating over velocity-space.
- \( \text{units} \) (str, optional) – Selects whether to run function with units and unit checks (when equal to “units”) or to run as unitless (when equal to “unitless”). The unitless version is substantially faster for intensive computations.

**Returns** \( f \) – Probability density in Velocity^-1, normalized so that \( \int_{-\infty}^{+\infty} f(v)dv = 1 \).

**Return type** Quantity

**Raises**

- TypeError – A parameter argument is not a Quantity and cannot be converted into a Quantity.
- UnitConversionError – If the parameters is not in appropriate units.
- ValueError – If the temperature is negative, or the particle mass or charge state cannot be found.
Notes

In one dimension, the Kappa velocity distribution function describing the distribution of particles with speed $v$ in a plasma with temperature $T$ and suprathermal parameter $\kappa$ is given by:

$$f = A_{\kappa} \left(1 + \frac{(\vec{v} - \vec{V}_{\text{drift}})^2}{\kappa v_{Th}^2, \kappa^2} \right)^{-\kappa}$$

where $v_{Th}, \kappa$ is the kappa thermal speed and $A_{\kappa} = \frac{1}{\sqrt{\pi \kappa^{3/2} v_{Th}^2, \kappa^2} \Gamma(\kappa + 1)}$ is the normalization constant.

As $\kappa$ approaches infinity, the kappa distribution function converges to the Maxwellian distribution function.

Examples

```python
>>> from astropy import units as u
>>> v=1 * u.m / u.s
>>> kappa_velocity_1D(v=v, T=30000*u.K, kappa=4, particle='e', v_drift=0 * u.m / u.s)
<Quantity 6.75549...e-07 s / m>
```

See also:

`kappa_velocity_3D()`, `kappa_thermal_speed()`

**kappa_velocity_3D**

plasmapy.formulary.distribution.kappa_velocity_3D(vx, vy, vz, T, kappa, particle='e', vx_drift=0, vy_drift=0, vz_drift=0, vTh=nan, units='units')

Return the probability density function for finding a particle with velocity components $v_x, v_y, v_z$ in m/s in a suprathermal plasma of temperature $T$ and parameter $'kappa$ which follows the 3D Kappa distribution function. This function assumes Cartesian coordinates.

Parameters

- **vx** *(Quantity)* – The velocity in x-direction units convertible to m/s.
- **vy** *(Quantity)* – The velocity in y-direction units convertible to m/s.
- **vz** *(Quantity)* – The velocity in z-direction units convertible to m/s.
- **T** *(Quantity)* – The temperature, preferably in Kelvin.
- **kappa** *(Quantity)* – The kappa parameter is a dimensionless number which sets the slope of the energy spectrum of suprathermal particles forming the tail of the Kappa velocity distribution function. Kappa must be greater than 3/2.
- **particle** *(str, optional)* – Representation of the particle species(e.g., ‘p’ for protons, ‘D+’ for deuterium, or ‘He-4 +1’ for $\text{He}^+_{4} :$ singly ionized helium-4)), which defaults to electrons.
- **vx_drift** *(Quantity, optional)* – The drift velocity in x-direction units convertible to m/s.
- **vy_drift** *(Quantity, optional)* – The drift velocity in y-direction units convertible to m/s.
• **vz** (Quantity, optional) – The drift velocity in z-direction units convertible to m/s.

• **vTh** (Quantity, optional) – Thermal velocity (most probable) in m/s. This is used for optimization purposes to avoid re-calculating vTh, for example when integrating over velocity-space.

• **units** (str, optional) – Selects whether to run function with units and unit checks (when equal to “units”) or to run as unitless (when equal to “unitless”). The unitless version is substantially faster for intensive computations.

**Returns**

f – Probability density in Velocity^-1, normalized so that: \( \int_0^\infty f(\vec{v})d\vec{v} = 1 \)

**Return type** Quantity

**Raises**

• TypeError – The parameter arguments are not Quantities and cannot be converted into Quantities.

• UnitConversionError – If the parameters is not in appropriate units.

• ValueError – If the temperature is negative, or the particle mass or charge state cannot be found.

**Notes**

In three dimensions, the Kappa velocity distribution function describing the distribution of particles with speed \( v \) in a plasma with temperature \( T \) and suprathermal parameter \( \kappa \) is given by:

\[
f = A_\kappa \left( 1 + \frac{(\vec{v} - \vec{V}_{\text{drift}})^2}{\kappa v_{Th}^2} \right)^{-(\kappa+1)}
\]

where \( v_{Th}, \kappa \) is the kappa thermal speed and \( A_\kappa = \frac{1}{2\pi(v_{Th},\kappa)^{3/2}} \Gamma(\kappa+1) \Gamma(\kappa-1/2) \Gamma(3/2) \) is the normalization constant.

As \( \kappa \) approaches infinity, the kappa distribution function converges to the Maxwellian distribution function.

See also:

kappa_velocity_1D(), kappa_thermal_speed()

**Example**

```python
>>> from astropy import units as u
>>> v=1 * u.m / u.s
>>> kappa_velocity_3D(vx=v, vy=v, vz=v, T=30000 * u.K, kappa=4, particle='e', vx_drift=0 * u.m / u.s, vy_drift=0 * u.m / u.s, vz_drift=0 * u.m / u.s)
<Quantity 3.7833...e-19 s^3 / m^3>
```
Examples:

- 1D Maxwellian distribution function

Particle drifts (`plasmapy.formulary.drifts`)

Formulas for calculating particle drifts.

Functions

- `ExB_drift(E, B)` Calculate the “electric cross magnetic” particle drift.
- `force_drift(F, B, q)` Calculate the general force drift for a particle in a magnetic field.

ExB_drift

`plasmapy.formulary.drifts.ExB_drift(E: Unit("V / m"), B: Unit("T")) -> Unit("m / s")`

Calculate the “electric cross magnetic” particle drift.

**Parameters**

- E (`Quantity`) – Electric field vector
- B (`Quantity`) – Magnetic field vector

**Returns**

- v – Drift velocity, in m/s

**Return type** `Quantity`

Examples

```python
>>> import astropy.units as u
>>> from astropy.constants.si import g0, e, m_e
>>> ex = np.array([1, 0, 0])
>>> ez = np.array([0, 0, 1])
>>> force_drift(-ez*g0*m_e, ex*0.01*u.T, e)
<Quantity [ 0.0000000e+00, -5.5756984e-09, 0.0000000e+00] m / s>
>>> force_drift(-ez*g0*m_e, ez*0.01*u.T, e)
<Quantity [ 0., -0., 0.] m / s>
>>> force_drift(-ez*g0*m_e, ex*u.T, e)
<Quantity [ 0.0000000e+00, -5.5756984e-11, 0.0000000e+00] m / s>
```

Notes

The E cross B drift is given by

\[ \vec{v} = \frac{\vec{E} \times \vec{B}}{|B|^2} \]

and is independent of particle charge.
References

- PM Bellan, Fundamentals of Plasma Physics, 3.57

force_drift

plasmapy.formulary.drifts.force_drift (F: Unit("N"); B: Unit("T"); q: Unit("C")) -> Unit("m/s")

Calculate the general force drift for a particle in a magnetic field.

Parameters

- \( F (\text{Quantity}) \) – Force acting on particle
- \( B (\text{Quantity}) \) – Magnetic field
- \( q (\text{Quantity}) \) – Particle charge

Examples

```python
>>> import astropy.units as u

>>> ex = np.array([1, 0, 0])
>>> ey = np.array([0, 1, 0])

>>> ExB_drift(ex * u.V/u.m, ey * u.T)
<Quantity [0., 0., 1.] m / s>

>>> ExB_drift(ex * u.V/u.m, ex * u.T)
<Quantity [0., 0., 0.] m / s>

>>> ExB_drift(ex * u.V/u.m, 100 * ey * u.T)
<Quantity [0. , 0. , 0.01] m / s>
```

Returns \( v \) – Drift velocity, in m/s

Return type Quantity

Notes

The particle drift in a magnetic field and with a general force (e.g. gravity) applied to it is given by

\[
v = \frac{F \times B}{q |B|^2}
\]

Note the charge dependency.

References

- PM Bellan, Fundamentals of Plasma Physics, 3.58

Magnetostatics (plasmapy.formulary.magnetostatics)

Define MagneticStatics class to calculate common static magnetic fields as first raised in issue #100.
Classes

- **CircularWire**(normal, center, radius, current) Circular wire(coil) class
- **FiniteStraightWire**(p1, p2, current) Finite length straight wire class.
- **GeneralWire**(parametric_eq, t1, t2, current) General wire class described by its parametric vector equation
- **InfiniteStraightWire**(direction, p0, current) Infinite straight wire class.
- **MagneticDipole**(moment, p0) Simple magnetic dipole - two nearby opposite point charges.
- **MagnetoStatics** Abstract class for all kinds of magnetic static fields
- **Wire** Abstract wire class for concrete wires to be inherited from.

CircularWire

class plasmapy.formulary.magnetostatics.CircularWire (normal: Unit("m"), center: Unit("m"), radius: Unit("m"), current: Unit("A"), n=300)

Bases: plasmapy.formulary.magnetostatics.Wire

Circular wire(coil) class

Parameters

- **normal** – three-dimensional normal vector of the circular coil
- **center**(astropy.units.Quantity) – three-dimensional position vector of the circular coil’s center
- **radius**(astropy.units.Quantity) – radius of the circular coil
- **current**(astropy.units.Quantity) – electric current

Methods Summary

- **magnetic_field**(p) -> Unit("T") Calculate magnetic field generated by this wire at position p
- **to_GeneralWire**() Convert this Wire into a GeneralWire.

Methods Documentation

**magnetic_field**(p) -> Unit("T")

Calculate magnetic field generated by this wire at position p

Parameters p(astropy.units.Quantity) – three-dimensional position vector

Returns B – magnetic field at the specified position

Return type astropy.units.Quantity

Notes

We use n points Gauss-Legendre quadrature to compute the integral. The default n is 300.
to_GeneralWire()
    Convert this Wire into a GeneralWire.

FiniteStraightWire
class plasmapy.formulary.magnetostatics.FiniteStraightWire(p1: Unit("m"), p2: Unit("m"), current: Unit("A"))
Bases: plasmapy.formulary.magnetostatics.Wire
Finite length straight wire class.
p1 to p2 direction is the positive current direction.

Parameters

- **p1** (astropy.units.Quantity) – three-dimensional Cartesian coordinate of one end of the straight wire
- **p2** (astropy.units.Quantity) – three-dimensional Cartesian coordinate of another end of the straight wire
- **current** (astropy.units.Quantity) – electric current

Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>magnetic_field(p)</td>
<td>Calculate magnetic field generated by this wire at position p</td>
</tr>
<tr>
<td>to_GeneralWire()</td>
<td>Convert this Wire into a GeneralWire.</td>
</tr>
</tbody>
</table>

Methods Documentation

magnetic_field(p) -> Unit("T")
    Calculate magnetic field generated by this wire at position p

    Parameters
    p (astropy.units.Quantity) – three-dimensional position vector

    Returns
    B – magnetic field at the specified position

    Return type
    astropy.units.Quantity

Notes

Let $P_f$ be the foot of perpendicular, $\theta_1$ and $\theta_2$ be the angles between $\overrightarrow{PP_1}$ and $\overrightarrow{PP_2}$.

to_GeneralWire()
    Convert this Wire into a GeneralWire.

GeneralWire
class plasmapy.formulary.magnetostatics.GeneralWire(parametric_eq, t1, t2, current: Unit("A"))
Bases: plasmapy.formulary.magnetostatics.Wire
General wire class described by its parametric vector equation

**Parameters**

- `parametric_eq (Callable)` – A vector-valued (with units of position) function of a single real parameter.
- `t1 (float)` – lower bound of the parameter, smaller than t2
- `t2 (float)` – upper bound of the parameter, larger than t1
- `current (astropy.units.Quantity)` – electric current

**Methods Summary**

```python
magnetic_field(p, n) Calculate magnetic field generated by this wire at position p
```

**Methods Documentation**

```python
magnetic_field(p: Unit("m"), n: numbers.Integral = 1000) -> Unit("T")
```

Calculate magnetic field generated by this wire at position `p`

**Parameters**

- `p (astropy.units.Quantity)` – three-dimensional position vector
- `n (int, optional)` – Number of segments for Wire calculation (defaults to 1000)

**Returns**

- `B` – magnetic field at the specified position

**Return type**

`astropy.units.Quantity`

**Notes**

For simplicity, we segment the wire into n equal pieces, and assume each segment is straight. Default n is 1000.

\[
\vec{B} \approx \frac{\mu_0 I}{4\pi} \sum_{i=1}^{n} \left( \vec{r}(t_i) - \vec{r}(t_{i-1}) \right) \times \left[ \frac{\vec{p} - \vec{r}(t_i) + \vec{r}(t_{i-1})}{2} \right]^{3}, \quad \text{where} \quad t_i = t_{\text{min}} + i/n \times (t_{\text{max}} - t_{\text{min}})
\]

**InfiniteStraightWire**

```python
class plasmapy.formulary.magnetostatics.InfiniteStraightWire (direction, p0: Unit("m"), current: Unit("A"))
```

**Bases:** `plasmapy.formulary.magnetostatics.Wire`

Infinite straight wire class.

**Parameters**

- `direction` – three-dimensional direction vector of the wire, also the positive current direction
- `p0 (astropy.units.Quantity)` – one point on the wire
• **current**(astropy.units.Quantity) – electric current

**Methods Summary**

| magnetic_field(p) | Calculate magnetic field generated by this wire at position p |

**Methods Documentation**

**magnetic_field**(p) -> Unit("T")
Calculate magnetic field generated by this wire at position p

- **Parameters**
  - p (astropy.units.Quantity) – three-dimensional position vector
- **Returns**
  - B – magnetic field at the specified position
- **Return type**
  - astropy.units.Quantity

**Notes**

**MagneticDipole**

class plasmapy.formulary.magnetostatics.MagneticDipole(moment: Unit("A m^2"), p0: Unit("m"))

Bases: plasmapy.formulary.magnetostatics.MagnetoStatics

Simple magnetic dipole - two nearby opposite point charges.

- **Parameters**
  - moment (astropy.units.Quantity) – Magnetic moment vector, in units of A * m^2
  - p0 (astropy.units.Quantity) – Position of the dipole

**Methods Summary**

| magnetic_field(p) | Calculate magnetic field generated by this wire at position p |

**Methods Documentation**

**magnetic_field**(p: Unit("m")) -> Unit("T")
Calculate magnetic field generated by this wire at position p

- **Parameters**
  - p (astropy.units.Quantity) – three-dimensional position vector
- **Returns**
  - B – magnetic field at the specified position
- **Return type**
  - astropy.units.Quantity

**MagnetoStatics**

class plasmapy.formulary.magnetostatics.MagnetoStatics

Bases: abc.ABC
Abstract class for all kinds of magnetic static fields

Methods Summary

| `magnetic_field(p)` | Calculate magnetic field generated by this wire at position `p` |

Methods Documentation

`magnetic_field(p: Unit("m")) -> Unit("T")`
Calculate magnetic field generated by this wire at position `p`

- **Parameters** `p` (*astropy.units.Quantity*) – three-dimensional position vector
- **Returns** `B` – magnetic field at the specified position
- **Return type** `astropy.units.Quantity`

Wire

class plasmapy.formulary.magnetostatics.Wire
Bases: `plasmapy.formulary.magnetostatics.MagnetoStatics`
Abstract wire class for concrete wires to be inherited from.

Class Inheritance Diagram

Plasma parameters (*plasmapy.formulary.parameters*)
This module gathers basic and general plasma parameters such as the plasma frequency or Debye length.

Functions

2.1. Formulary
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mass_density</code></td>
<td>Utility function to merge two possible inputs for particle charge.</td>
</tr>
<tr>
<td><code>Alfven_speed</code></td>
<td>Return the Alfvén speed.</td>
</tr>
<tr>
<td><code>ion_sound_speed</code></td>
<td>Return the ion sound speed for an electron-ion plasma.</td>
</tr>
<tr>
<td><code>thermal_speed</code></td>
<td>Return the most probable speed for a particle within a Maxwellian distribution.</td>
</tr>
<tr>
<td><code>thermal_pressure</code></td>
<td>Return the thermal pressure for a Maxwellian distribution.</td>
</tr>
<tr>
<td><code>kappa_thermal_speed</code></td>
<td>Return the most probable speed for a particle within a Kappa distribution.</td>
</tr>
<tr>
<td><code>Hall_parameter</code></td>
<td>Calculate the ratio between the particle gyrofrequency and the particle-ion particle collision rate.</td>
</tr>
<tr>
<td><code>gyrofrequency</code></td>
<td>Calculate the particle gyrofrequency in units of radians per second.</td>
</tr>
<tr>
<td><code>gyroradius</code></td>
<td>Return the particle gyroradius.</td>
</tr>
<tr>
<td><code>plasma_frequency</code></td>
<td>Calculate the particle plasma frequency.</td>
</tr>
<tr>
<td><code>Debye_length</code></td>
<td>Calculate the characteristic decay length for electric fields.</td>
</tr>
<tr>
<td><code>Debye_number</code></td>
<td>Return the number of electrons within a sphere with a radius of the Debye length.</td>
</tr>
<tr>
<td><code>inertial_length</code></td>
<td>Calculate a charged particle’s inertial length.</td>
</tr>
<tr>
<td><code>magnetic_pressure</code></td>
<td>Calculate the magnetic pressure.</td>
</tr>
<tr>
<td><code>magnetic_energy_density</code></td>
<td>Calculate the magnetic energy density.</td>
</tr>
<tr>
<td><code>upper_hybrid_frequency</code></td>
<td>Return the upper hybrid frequency.</td>
</tr>
<tr>
<td><code>lower_hybrid_frequency</code></td>
<td>Return the lower hybrid frequency.</td>
</tr>
</tbody>
</table>

### mass_density

```python
plasmapy.formulary.parameters.mass_density(density: [Unit("I / m3"), Unit("kg / m3")],
particle: Optional[str] = None, z_mean: Optional[numbers.Real] = None) -> Unit("kg / m3")
```

Utility function to merge two possible inputs for particle charge.

#### Parameters

- **density** *(Quantity)* – Either a particle density (number of particles per unit volume, in units of $1/m^3$) or a mass density (in units of $kg/m^3$ or equivalent).
- **particle** *(str, optional)* – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for singly ionized helium-4), which defaults to electrons. If no charge state information is provided, then the particles are assumed to be singly charged.
- **z_mean** *(float)* – An optional float describing the average ionization of a particle species.

#### Raises

- **ValueError** – If the `density` has units inconvertible to either a particle density or a mass density, or if you pass in a number density without a particle.

#### Returns

The mass density calculated from all the provided sources of information.

#### Return type

*Quantity*
Examples

```python
>>> from astropy import units as u
>>> mass_density(1 * u.m ** -3, 'p')
<Quantity 1.67353...e-27 kg / m³>
>>> mass_density(4 * u.m ** -3, 'D+')
<Quantity 1.33779...e-26 kg / m³>
```

Alfvén_speed

`plasmapy.formulary.parameters.Alfven_speed(B: Unit("T"), density: [Unit("1 / m^3"), Unit("kg / m^3")], ion='p+', z_mean=None) - Unit("m / s")`

Return the Alfvén speed.

Parameters

- **B** *(Quantity)* – The magnetic field magnitude in units convertible to tesla.
- **density** *(Quantity)* – Either the ion number density in units convertible to 1 / m**3**, or the mass density in units convertible to kg / m**3**.
- **ion**(str, optional) – Representation of the ion species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for singly ionized helium-4), which defaults to protons. If no charge state information is provided, then the ions are assumed to be singly charged.
- **z_mean** *(Quantity, optional)* – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. If this quantity is not given then the atomic charge state (integer) of the ion is used. This is effectively an average Alfvén speed for the plasma where multiple charge states are present.

Returns **V_A** – The Alfvén speed of the plasma in units of meters per second.

Return type  Quantity with units of speed

Raises

- **TypeError** – The magnetic field and density arguments are not instances of *Quantity* and cannot be converted into those.
- **UnitConversionError** – If the magnetic field or density is not in appropriate units.
- **RelativityError** – If the Alfvén velocity is greater than or equal to the speed of light
- **ValueError** – If the density is negative, or the ion mass or charge state cannot be found.

Warns

- **plasmapy.utils.RelativityWarning** – If the Alfvén velocity exceeds 5% of the speed of light
- **astropy.units.UnitsWarning** – if units are not provided, SI units are assumed.

Notes

The Alfvén speed \( V_A \) is the typical propagation speed of magnetic disturbances in a plasma, and is given by:

\[
V_A = \frac{B}{\sqrt{\mu_0 \rho}}
\]
where the mass density is \( \rho = n_i m_i + n_e m_e \).

This expression does not account for relativistic effects, and loses validity when the resulting speed is a significant fraction of the speed of light.

**Examples**

```python
>>> from astropy import units as u
>>> from astropy.constants.si import m_p, m_e
>>> B = 0.014 * u.T
>>> n = 5e19 * u.m**-3
>>> rho = n * (m_p + m_e)
>>> ion = 'p'
>>> Alfven_speed(B, n, ion)
<Quantity 43173.870... m / s>
>>> Alfven_speed(B, rho, ion)
<Quantity 43173.870... m / s>
>>> Alfven_speed(B, rho, ion).to(u.cm/u.us)
<Quantity 4.31738... cm / us>
```

**ion_sound_speed**

`plasmapy.formulary.parameters.ion_sound_speed(T_e: Unit("K"), T_i: Unit("K"), n_e: Unit("1 / m^3") = None, k: Unit("1 / m") = None, gamma_e=1, gamma_i=3, ion='p+', z_mean=None) -> Unit("m / s")`

Return the ion sound speed for an electron-ion plasma.

**Parameters**

- **T_e (Quantity)** – Electron temperature in units of temperature or energy per particle. If this is not given, then the electron temperature is assumed to be zero.

- **T_i (Quantity)** – Ion temperature in units of temperature or energy per particle. If this is not given, then the ion temperature is assumed to be zero.

- **n_e (Quantity)** – Electron number density. If this is not given, then ion_sound_speed will be approximated in the non-dispersive limit \((k^2 \lambda_D^2)\) will be assumed zero). If n_e is given, a value for k must also be given.

- **k (Quantity)** – Wavenumber (in units of inverse length, e.g. per meter). If this is not given, then ion_sound_speed will be approximated in the non-dispersive limit \((k^2 \lambda_D^2)\) will be assumed zero). If k is given, a value for n_e must also be given.

- **gamma_e (float or int)** – The adiabatic index for electrons, which defaults to 1. This value assumes that the electrons are able to equalize their temperature rapidly enough that the electrons are effectively isothermal.

- **gamma_i (float or int)** – The adiabatic index for ions, which defaults to 3. This value assumes that ion motion has only one degree of freedom, namely along magnetic field lines.

- **ion (str, optional)** – Representation of the ion species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for singly ionized helium-4), which defaults to protons. If no charge state information is provided, then the ions are assumed to be singly charged.
• **z_mean** *(Quantity, optional)* – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. If this quantity is not given then the atomic charge state (integer) of the ion is used. This is effectively an average ion sound speed for the plasma where multiple charge states are present.

**Returns** $V_S$ – The ion sound speed in units of meters per second.

**Return type** Quantity

**Raises**

- **TypeError** – If any of the arguments are not entered as keyword arguments or are of an incorrect type.
- **ValueError** – If the ion mass, adiabatic index, or temperature are invalid.
- **PhysicsError** – If an adiabatic index is less than one.
- **UnitConversionError** – If the temperature, electron number density, or wavenumber is in incorrect units.

**Warns**

- **RelativityWarning** – If the ion sound speed exceeds 5% of the speed of light.
- **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed.
- **PhysicsWarning** – If only one of $(k, n_e)$ is given, the non-dispersive limit is assumed.

**Notes**

The ion sound speed $V_S$ is given by

$$V_S = \sqrt{\frac{\gamma_e k_B T_e + \gamma_i k_B T_i}{m_i (1 + k^2 \lambda_D^2)}}$$

where $\gamma_e$ and $\gamma_i$ are the electron and ion adiabatic indices, $k_B$ is the Boltzmann constant, $T_e$ and $T_i$ are the electron and ion temperatures, $Z$ is the charge state of the ion, $m_i$ is the ion mass, $\lambda_D$ is the Debye length, and $k$ is the wavenumber.

In the non-dispersive limit ($k^2 \lambda_D^2$ is small) the equation for $V_S$ is approximated (the denominator reduces to $m_i$).

When the electron temperature is much greater than the ion temperature, the ion sound velocity reduces to $\sqrt{\gamma_e k_B T_e/m_i}$. Ion acoustic waves can therefore occur even when the ion temperature is zero.

**Example**

```python
>>> from astropy import units as u
>>> n = 5e19 * u.m**-3
>>> k_1 = 3e1 * u.m**-1
>>> k_2 = 3e7 * u.m**-1
>>> ion_sound_speed(T_e=5e6 * u.K, T_i=0 * u.K, ion='p', gamma_e=1, gamma_i=3)
<Quantity 203155... m / s>
>>> ion_sound_speed(T_e=5e6 * u.K, T_i=0 * u.K, n_e=n, k=k_1, ion='p', gamma_e=1, gamma_i=3)
<Quantity 203155... m / s>
>>> ion_sound_speed(T_e=5e6 * u.K, T_i=0 * u.K, n_e=n, k=k_2, ion='p', gamma_e=1, gamma_i=3)
<Quantity 203155... m / s>
```

(continues on next page)
thermal_speed

plasmapy.formulary.parameters.thermal_speed(T: Unit("K"), particle: plasmapy.atomic.particle_class.Particle = 'e-', method='most_probable', mass: Unit("kg") = <Quantity nan kg>) -> Unit("m / s")

Return the most probable speed for a particle within a Maxwellian distribution.

Parameters

- **T** (*Quantity*) – The particle temperature in either kelvin or energy per particle
- **particle** (*str, optional*) – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for singly ionized helium-4), which defaults to electrons. If no charge state information is provided, then the particles are assumed to be singly charged.
- **method** (*str, optional*) – Method to be used for calculating the thermal speed. Options are 'most_probable' (default), 'rms', and 'mean_magnitude'.
- **mass** (*Quantity*) – The particle’s mass override. Defaults to NaN and if so, doesn’t do anything, but if set, overrides mass acquired from particle. Useful with relative velocities of particles.

Returns **V** – particle thermal speed

Return type **Quantity**

Raises

- **TypeError** – The particle temperature is not a ~astropy.units.Quantity
- **UnitConversionError** – If the particle temperature is not in units of temperature or energy per particle
- **ValueError** – The particle temperature is invalid or particle cannot be used to identify an isotope or particle

Warns

- **RelativityWarning** – If the ion sound speed exceeds 5% of the speed of light, or
- **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed.

Notes

The particle thermal speed is given by:

\[ V_{th,i} = \sqrt{\frac{2k_BT_i}{m_i}} \]
This function yields the most probable speed within a distribution function. However, the definition of thermal velocity varies by the square root of two depending on whether or not this velocity absorbs that factor in the expression for a Maxwellian distribution. In particular, the expression given in the NRL Plasma Formulary [1] is a square root of two smaller than the result from this function.

**Examples**

```python
>>> from astropy import units as u
>>> thermal_speed(5*u.eV, 'p')
<Quantity 30949.6... m / s>
>>> thermal_speed(1e6*u.K, particle='p')
<Quantity 128486... m / s>
>>> thermal_speed(5*u.eV, particle='e-')
<Quantity 132620... m / s>
>>> thermal_speed(1e6*u.K, particle='e-')
<Quantity 550569... m / s>
>>> thermal_speed(1e6*u.K, method="rms")
<Quantity 674307... m / s>
>>> thermal_speed(1e6*u.K, method="mean_magnitude")
<Quantity 621251... m / s>
```

**thermal_pressure**

```python
def thermal_pressure(T, n):
    T = T.to('K')
    n = n.to('1 / m**3')
    return T**4 / (16 * np.pi**3 * m**3)
```

Return the thermal pressure for a Maxwellian distribution.

**Parameters**

- `T` ([Quantity]) – The particle temperature in kelvin or energy per particle
- `n` ([Quantity]) – The particle number density in units convertible to m**-3.

**Examples**

```python
>>> import astropy.units as u
>>> thermal_pressure(1*u.eV, 1e20/u.m**3)
<Quantity 16.021... Pa>
>>> thermal_pressure(10*u.eV, 1e20/u.m**3)
<Quantity 160.21... Pa>
```

_Returns_ `p_th` – Thermal pressure.

_Return type_ `Quantity`

_Raises_

- `TypeError` – The temperature or number density is not a `Quantity`.
- `UnitConversionError` – If the particle temperature is not in units of temperature or energy per particle.
Notes

The thermal pressure is given by:

\[ T_{th} = n k_B T \]

kappa_thermal_speed

`plasmapy.formulary.parameters.kappa_thermal_speed(T: Unit("K"), kappa, particle='e-', method='most_probable') -> Unit("m / s")`

Return the most probable speed for a particle within a Kappa distribution.

Parameters

- **T** (*Quantity*) – The particle temperature in either kelvin or energy per particle
- **kappa** (*float*) – The kappa parameter is a dimensionless number which sets the slope of the energy spectrum of suprathermal particles forming the tail of the Kappa velocity distribution function. Kappa must be greater than 3/2.
- **particle** (*str, optional*) – Representation of the particle species (e.g., ‘p’ for protons, ‘D+’ for deuterium, or ‘He-4 +1’ for singly ionized helium-4), which defaults to electrons. If no charge state information is provided, then the particles are assumed to be singly charged.
- **method** (*str, optional*) – Method to be used for calculating the thermal speed. Options are ‘most_probable’ (default), ‘rms’, and ‘mean_magnitude’.

Returns **V** – Particle thermal speed

Return type **Quantity**

Raises

- **TypeError** – The particle temperature is not a ~astropy.units.Quantity.
- **astropy.units.UnitConversionError** – If the particle temperature is not in units of temperature or energy per particle.
- **ValueError** – The particle temperature is invalid or particle cannot be used to identify an isotope or particle.

Warns

- **RelativityWarning** – If the particle thermal speed exceeds 5% of the speed of light, or
- **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed.

Notes

The particle thermal speed is given by:

\[ V_{th,i} = \sqrt{(2\kappa - 3) \frac{2k_B T_i}{\kappa m_i}} \]

For more discussion on the mean_magnitude calculation method, see\(^1\).

\(^1\) PlasmaPy Issue #186, https://github.com/PlasmaPy/PlasmaPy/issues/186
Examples

```python
>>> from astropy import units as u

>>> kappa_thermal_speed(5*u.eV, 4, 'p')  # defaults to most probable
<Quantity 24467.87... m / s>

>>> kappa_thermal_speed(5*u.eV, 4, 'p', 'rms')
<Quantity 37905.47... m / s>

>>> kappa_thermal_speed(5*u.eV, 4, 'p', 'mean_magnitude')
<Quantity 34922.98... m / s>
```

References

See also:

- `plasmapy.formulary.kappa_thermal_speed()`
- `plasmapy.formulary.kappa_velocity_1D()`

Hall parameter

```python
plasmapy.formulary.parameters.Hall_parameter(n: Unit("1 / m3"), T: Unit("K"), B: Unit("T"), ion_particle, particle='e-', coulomb_log=None, V=None, coulomb_log_method='classical')
```

Calculate the ratio between the particle gyrofrequency and the particle-ion particle collision rate.

All parameters apply to `particle`.

**Parameters**

- `n (Quantity)` – The density of particle s
- `T (Quantity)` – The temperature of particles
- `B (Quantity)` – The magnetic field
- `ion_particle (str)` – String signifying the type of ion.
- `particle (str, optional)` – String signifying the type of particles. Defaults to electrons.
- `coulomb_log (float, optional)` – Preset value for the Coulomb logarithm. Used mostly for testing purposes.
- `V (Quantity)` – The relative velocity between `particle` and ion particles.
- `coulomb_log_method (str, optional)` – Method used for Coulomb logarithm calculation. Refer to its documentation.

See also:

- `plasmapy.formulary.parameters.gyrofrequency()`
- `plasmapy.formulary.parameters.fundamental_electron_collision_freq()`
- `plasmapy.formulary.collisions.Coulomb_logarithm()`

**Returns**

- `Return type astropy.units.quantity.Quantity`
Examples

```python
>>> from astropy import units as u
>>> Hall_parameter(1e10 * u.m**-3, 2.8e3 * u.eV, 2.3 * u.T, 'He-4 +1')
<Quantity 7.26446...e+16>
>>> Hall_parameter(1e10 * u.m**-3, 5.8e3 * u.eV, 2.3 * u.T, 'He-4 +1')
<Quantity 2.11158...e+17>
```

gyrofrequency


Calculate the particle gyrofrequency in units of radians per second.

**Parameters**

- **B** (`Quantity`) – The magnetic field magnitude in units convertible to tesla.
- **particle** (`str`, optional) – Representation of the particle species (e.g., ‘p’ for protons, ‘D+’ for deuterium, or ‘He-4 +1’ for singly ionized helium-4), which defaults to electrons. If no charge state information is provided, then the particles are assumed to be singly charged.
- **signed** (`bool`, optional) – The gyrofrequency can be defined as signed (negative for electron, positive for ion). Default is `False` (unsigned, i.e. always positive).
- **Z** (`float or Quantity`, optional) – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. If this quantity is not given then the atomic charge state (integer) of the ion is used. This is effectively an average gyrofrequency for the plasma where multiple charge states are present, and should not be interpreted as the gyrofrequency for any single particle. If not provided, it defaults to the integer charge of the particle.

**Returns** `omega_c` – The particle gyrofrequency in units of radians per second

**Return type** `Quantity`

**Raises**

- **TypeError** – If the magnetic field is not a `Quantity` or particle is not of an appropriate type
- **ValueError** – If the magnetic field contains invalid values or particle cannot be used to identify an particle or isotope

**Warns** `~astropy.units.UnitsWarning` – If units are not provided, SI units are assumed

**Notes**

The particle gyrofrequency is the angular frequency of particle gyration around magnetic field lines and is given by:

\[ \omega_{ci} = \frac{ZeB}{m_i} \]

The particle gyrofrequency is also known as the particle cyclotron frequency or the particle Larmor frequency.

The recommended way to convert from angular frequency to frequency is to use an equivalency between cycles per second and Hertz, as `Astropy's dimensionless_angles equivalency does not account for the factor of 2pi`. 

---

PlasmaPy Documentation, Release 0.2.0

Chapter 2. User Documentation
2*pi needed during this conversion. The dimensionless_angles equivalency is appropriate when dividing a velocity by an angular frequency to get a length scale.

Examples

```python
>>> from astropy import units as u
>>> gyrofrequency(0.1*u.T)
<Quantity 1.7588...e+10 rad / s>
>>> gyrofrequency(0.1*u.T, to_hz=True)
<Quantity 2.79924...e+09 Hz>
>>> gyrofrequency(0.1*u.T, signed=True)
<Quantity -1.75882...e+10 rad / s>
>>> gyrofrequency(0.01*u.T, 'p')
<Quantity 957883.32... rad / s>
>>> gyrofrequency(0.01*u.T, 'p', signed=True)
<Quantity 957883.32... rad / s>
>>> gyrofrequency(0.01*u.T, particle='T+')
<Quantity 319964.5... rad / s>
>>> gyrofrequency(0.01*u.T, particle='T+', to_hz=True)
<Quantity 50923.9... Hz>
>>> omega_ce = gyrofrequency(0.1*u.T)
>>> print(omega_ce)
1758820... rad / s
>>> f_ce = omega_ce.to(u.Hz, equivalencies=[(u.cy/u.s, u.Hz)])
>>> print(f_ce)
279924... Hz
```

Other Parameters to_hz (bool) – Set True to to convert function output from angular frequency to Hz

**gyroradius**

plasmapy.formulary.parameters.gyroradius(B: Unit("T"), particle='e-', *, Vperp: Unit("m / s") = <Quantity nan m / s>, T_i: Unit("K") = <Quantity nan K>) -> Unit("m")

Return the particle gyroradius.

Parameters

- **B (Quantity)** – The magnetic field magnitude in units convertible to tesla.
- **particle (str, optional)** – Representation of the particle species (e.g., 'p' for protons, 'D+' for deuterium, or 'He-4 +1' for singly ionized helium-4), which defaults to electrons. If no charge state information is provided, then the particles are assumed to be singly charged.
- **Vperp (Quantity, optional)** – The component of particle velocity that is perpendicular to the magnetic field in units convertible to meters per second. Must be input as a keyword argument.
- **T_i (Quantity, optional)** – The particle temperature in units convertible to kelvin. Must be input as a keyword argument.

Returns r_Li – The particle gyroradius in units of meters. This ~astropy.units.Quantity will be based on either the perpendicular component of particle velocity as inputted, or the most probable speed for an particle within a Maxwellian distribution for the particle temperature.

2.1. Formulary
Return type  Quantity

Raises
• `TypeError` – The arguments are of an incorrect type
• `UnitConversionError` – The arguments do not have appropriate units
• `ValueError` – If any argument contains invalid values

Warns  `-astropy.units.UnitsWarning` – If units are not provided, SI units are assumed

Notes
One but not both of V perp and T_i must be inputted.

If any of B, V perp, or T_i is a number rather than a Quantity, then SI units will be assumed and a warning will be raised.

The particle gyroradius is also known as the particle Larmor radius and is given by

\[ r_L = \frac{V_\perp}{\omega_{ce}} \]

where \( V_\perp \) is the component of particle velocity that is perpendicular to the magnetic field and \( \omega_{ce} \) is the particle gyrofrequency. If a temperature is provided, then \( V_\perp \) will be the most probable thermal velocity of an particle at that temperature.

Examples

```python
>>> from astropy import units as u
>>> gyroradius(0.2*u.T, particle='p+', T_i=1e5*u.K)
<Quantity 0.002120... m>
>>> gyroradius(0.2*u.T, particle='p+', T_i=1e5*u.K)
<Quantity 0.002120... m>
>>> gyroradius(5*u.uG, particle='alpha', T_i=1*u.eV)
<Quantity 288002.38... m>
>>> gyroradius(400*u.G, particle='Fe+++', V_perp=1e7*u.m/u.s)
<Quantity 48.23129... m>
>>> gyroradius(B=0.01*u.T, T_i=1e6*u.K)
<Quantity 0.003130... m>
>>> gyroradius(B=0.01*u.T, V_perp=1e6*u.m/u.s)
<Quantity 0.000568... m>
>>> gyroradius(0.2*u.T, T_i=1e5*u.K)
<Quantity 4.94949...e-05 m>
>>> gyroradius(5*u.uG, T_i=1*u.eV)
<Quantity 6744.25... m>
>>> gyroradius(400*u.G, V_perp=1e7*u.m/u.s)
<Quantity 0.001421... m>
```

plasma_frequency

plasmapy.formulary.parameters.plasma_frequency(n: Unit(“1 / m3”), particle=’e-’, z_mean=None, to_hz=False) -> Unit(“rad / s”)

Calculate the particle plasma frequency.
Parameters

- **n** (*Quantity*) – Particle number density in units convertible to per cubic meter

- **particle** (*str, optional*) – Representation of the particle species (e.g., ‘p’ for protons, ‘D+’ for deuterium, or ‘He-4 +1’ for singly ionized helium-4), which defaults to electrons. If no charge state information is provided, then the particles are assumed to be singly charged.

- **z_mean** (*Quantity, optional*) – The average ionization (arithmetic mean) for a plasma where a macroscopic description is valid. If this quantity is not given then the atomic charge state (*int*) of the ion is used. This is effectively an average plasma frequency for the plasma where multiple charge states are present.

**Returns**  
*omega_p* – The particle plasma frequency in radians per second.

**Return type**  
*Quantity*

**Raises**

- **TypeError** – If n_i is not a *Quantity* or particle is not of an appropriate type.

- **UnitConversionError** – If n_i is not in correct units

- **ValueError** – If n_i contains invalid values or particle cannot be used to identify an particle or isotope.

**Warns**  
~astropy.units.UnitsWarning – If units are not provided, SI units are assumed

**Notes**

The particle plasma frequency is

\[ \omega_{\text{pl}} = Z e \sqrt{\frac{n_i}{\epsilon_0 m_i}} \]

At present, astropy.units does not allow direct conversions from radians/second for angular frequency to 1/second or Hz for frequency. The dimensionless_angles equivalency allows that conversion, but does not account for the factor of 2*pi. The alternatives are to convert to cycle/second or to do the conversion manually, as shown in the examples.

**Example**

```python
>>> from astropy import units as u
>>> plasma_frequency(1e19*u.m**-3, particle='p')
<Quantity 4.16329...e+09 rad / s>
>>> plasma_frequency(1e19*u.m**-3, particle='p', to_hz=True)
<Quantity 6.62608...e+08 Hz>
>>> plasma_frequency(1e19*u.m**-3, particle='D+')
<Quantity 2.94462...e+09 rad / s>
>>> plasma_frequency(1e19*u.m**-3)
<Quantity 1.78398...e+11 rad / s>
>>> plasma_frequency(1e19*u.m**-3, to_hz=True)
<Quantity 2.83930...e+10 Hz>
```

**Other Parameters**  
**to_hz** (*bool*) – Set True to convert function output from angular frequency to Hz
Debye_length

`plasmapy.formulary.parameters.Debye_length(T_e: Unit("K"), n_e: Unit("1 / m^3")) -> Unit("m")`

Calculate the characteristic decay length for electric fields, due to charge screening.

**Parameters**

- **T_e** (*Quantity*) – Electron temperature
- **n_e** (*Quantity*) – Electron number density

**Returns**  
**lambda_D** – The Debye length in meters

**Return type**  *Quantity*

**Raises**

- **TypeError** – If either argument is not a *Quantity*
- **UnitConversionError** – If either argument is in incorrect units
- **ValueError** – If either argument contains invalid values

**Warns**  
- *astropy.units.UnitsWarning* – If units are not provided, SI units are assumed

**Notes**

The Debye length is the exponential scale length for charge screening and is given by

\[ \lambda_D = \sqrt{\frac{\varepsilon_0 k_B T_e}{n_e e^2}} \]

for an electron plasma with nearly stationary ions.

The electrical potential will drop by a factor of 1/e every Debye length.

Plasmas will generally be quasineutral on length scales significantly larger than the Debye length.

**See also:**  
*Debye_number()*

**Example**

```python
>>> from astropy import units as u
>>> Debye_length(5e6*u.K, 5e15*u.m**-3)
<Quantity 0.002182... m>
```

Debye_number

`plasmapy.formulary.parameters.Debye_number(T_e: Unit("K"), n_e: Unit("1 / m^3")) -> Unit("dimensionless")`

Return the number of electrons within a sphere with a radius of the Debye length.

**Parameters**

- **T_e** (*Quantity*) – Electron temperature
• \texttt{n\_e} (\texttt{Quantity}) – Electron number density

**Raises**

• \texttt{TypeError} – If either argument is not a \texttt{Quantity}
• \texttt{astropy.units.UnitConversionError} – If either argument is in incorrect units
• \texttt{ValueError} – If either argument contains invalid values

**Warns** \texttt{astropy.units.UnitsWarning} – If units are not provided, SI units are assumed

**Returns** \texttt{N\_D} – Number of electrons within a sphere with a radius of the Debye length

**Return type** \texttt{Quantity}

**Notes**

The Debye number is the number of electrons contained within a sphere with a radius of a Debye length and is given by

\[
N_D = \frac{4\pi}{3} n_e \lambda_D^3
\]

The Debye number is also known as the plasma parameter.

Collective behavior requires a Debye number significantly larger than one.

**See also:**

\texttt{Debye\_length()}

**Example**

```python
>>> from astropy import units as u
>>> Debye_number(5e6*u.K, 5e9*u.cm**-3)
<Quantity 2.17658...e+08>
```

**inertial\_length**

\texttt{plasmapy.formulary.parameters.inertial\_length}(\texttt{n: Unit(“1 / m3”), particle: plasmapy.atomic.particle\_class.Particle}) \rightarrow \texttt{Unit(“m”)}

Calculate a charged particle’s inertial length.

**Parameters**

• \texttt{n} (\texttt{Quantity}) – Particle number density in units convertible to m **\texttt{.3}.
• \texttt{particle} (\texttt{str}, optional) – Representation of the particle species (e.g., ‘p+’ for protons, ‘D+’ for deuterium, or ‘He-4 +1’ for singly ionized helium-4).

**Returns** \texttt{d} – The particle’s inertial length in meters.

**Return type** \texttt{Quantity}

**Raises**

• \texttt{TypeError} – If \texttt{n} not a \texttt{Quantity} or particle is not a string.
• \texttt{UnitConversionError} – If \texttt{n} is not in units of a number density.
• `ValueError` – The particle density does not have an appropriate value.

**Warns** `-astropy.units.UnitsWarning` – If units are not provided and SI units are assumed.

**Notes**

The inertial length of a particle of species \( s \) is given by

\[
d = \frac{c}{\omega_{ps}}
\]

The inertial length is the characteristic length scale for a particle to be accelerated in a plasma. The Hall effect becomes important on length scales shorter than the ion inertial length.

The inertial length is also known as the skin depth.

**Example**

```python
>>> from astropy import units as u
>>> inertial_length(5 * u.m ** -3, 'He+')
<Quantity 2.02985...e+08 m>
>>> inertial_length(5 * u.m ** -3, 'e-')
<Quantity 2376534.75... m>
```

`magnetic_pressure`

`plasmapy.formulary.parameters.magnetic_pressure(B: Unit("T")) -> Unit("Pa")`

Calculate the magnetic pressure.

**Parameters** `B` (*Quantity*) – The magnetic field in units convertible to tesla.

**Returns** `p_B` – The magnetic pressure in units in pascals (newtons per square meter).

**Return type** *Quantity*

**Raises**

- `TypeError` – If the input is not a `Quantity`.

- `UnitConversionError` – If the input is not in units convertible to tesla.

- `ValueError` – If the magnetic field strength is not a real number between +/- infinity.

**Warns** `-astropy.units.UnitsWarning` – If units are not provided, SI units are assumed

**Notes**

The magnetic pressure is given by:

\[
p_B = \frac{B^2}{2\mu_0}
\]

The motivation behind having two separate functions for magnetic pressure and magnetic energy density is that it allows greater insight into the physics that are being considered by the user and thus more readable code.

**See also:**

`magnetic_energy_density()` returns an equivalent `Quantity`, except in units of joules per cubic meter.
Example

```python
>>> from astropy import units as u
>>> magnetic_pressure(0.1*u.T).to(u.Pa)
<Quantity 3978.87... Pa>
```

**magnetic_energy_density**

Calculate the magnetic energy density.

**Parameters**

- **B** (*Quantity*) - The magnetic field in units convertible to tesla.

**Returns**

- **E_B** - The magnetic energy density in units of joules per cubic meter.

**Return type**

*Quantity*

**Raises**

- **TypeError** - If the input is not a Quantity.
- **UnitConversionError** - If the input is not in units convertible to tesla.
- **ValueError** - If the magnetic field strength does not have an appropriate value.

**Notes**

The magnetic energy density is given by:

\[ E_B = \frac{B^2}{2\mu_0} \]

The motivation behind having two separate functions for magnetic pressure and magnetic energy density is that it allows greater insight into the physics that are being considered by the user and thus more readable code.

**See also:**

- **magnetic_pressure()** Returns an equivalent Quantity, except in units of pascals.

Example

```python
>>> from astropy import units as u
>>> magnetic_energy_density(0.1*u.T)
<Quantity 3978.87... J / m^3>
```

**upper_hybrid_frequency**

Return the upper hybrid frequency.

**Parameters**
• **B** (*Quantity*) – The magnetic field magnitude in units convertible to tesla.

• **n_e** (*Quantity*) – The electron number density.

**Returns** `omega_uh` – The upper hybrid frequency in radians per second.

**Return type** *Quantity*

**Raises**

• **TypeError** – If either of **B** or **n_e** is not a *Quantity*.

• **UnitConversionError** – If either of **B** or **n_e** is in incorrect units.

• **ValueError** – If either of **B** or **n_e** contains invalid values or are of incompatible dimensions.

**Warns** *astropy.units.UnitsWarning* – If units are not provided, SI units are assumed.

**Notes**

The upper hybrid frequency is given through the relation

\[
\omega_{uh}^2 = \omega_{ce}^2 + \omega_{pe}^2
\]

where \( \omega_{ce} \) is the electron gyrofrequency and \( \omega_{pe} \) is the electron plasma frequency.

**Example**

```python
>>> from astropy import units as u
>>> upper_hybrid_frequency(0.2*u.T, n_e=5e19*u.m**-3)
<Quantity 4.00459...e+11 rad / s>
>>> upper_hybrid_frequency(0.2*u.T, n_e=5e19*u.m**-3, to_hz=True)
<Quantity 6.37350...e+10 Hz>
```

**Other Parameters**

• **to_hz** (*bool*) – Set *True* to convert function output from angular frequency to **Hz**

**lower_hybrid_frequency**

```python
plasmapy.formulary.parameters.lower_hybrid_frequency(B: Unit("T"), n_i: Unit("1 / m3"), ion='p+', to_hz=False) -> Unit("rad / s")
```

Return the lower hybrid frequency.

**Parameters**

• **B** (*Quantity*) – The magnetic field magnitude in units convertible to tesla.

• **n_i** (*Quantity*) – Ion number density.

• **ion** (*str, optional*) – Representation of the ion species (e.g., ‘p’ for protons, ‘D+’ for deuterium, or ‘He-4 +1’ for singly ionized helium-4), which defaults to protons. If no charge state information is provided, then the ions are assumed to be singly charged.

**Returns** `omega_lh` – The lower hybrid frequency in radians per second.

**Return type** *Quantity*
Raises

- **TypeError** – If either of \( B \) or \( n_i \) is not a `Quantity`, or ion is of an inappropriate type.
- **UnitConversionError** – If either of \( B \) or \( n_i \) is in incorrect units.
- **ValueError** – If either of \( B \) or \( n_i \) contains invalid values or are of incompatible dimensions, or ion cannot be used to identify an ion or isotope.

Warns - `astropy.units.UnitsWarning` – If units are not provided, SI units are assumed

Notes

The lower hybrid frequency is given through the relation

\[
\frac{1}{\omega_{lh}^2} = \frac{1}{\omega_{ci}^2} + \frac{1}{\omega_{pi}^2} + \frac{1}{\omega_{ci}\omega_{ce}}
\]

where \( \omega_{ci} \) is the ion gyrofrequency, \( \omega_{ce} \) is the electron gyrofrequency, and \( \omega_{pi} \) is the ion plasma frequency.

Example

```python
>>> from astropy import units as u
>>> lower_hybrid_frequency(0.2*u.T, n_i=5e19*u.m**-3, ion='D+')
<Quantity 5.78372...e+08 rad / s>
>>> lower_hybrid_frequency(0.2*u.T, n_i=5e19*u.m**-3, ion='D+', to_hz = True)
<Quantity 92050879.3... Hz>
```

Other Parameters `to_hz` (bool) – Set `True` to to convert function output from angular frequency to Hz

Examples:

- 1D Maxwellian distribution function

Quantum physics functions (`plasmapy.formulary.quantum`)

Functions for quantum parameters, including electron degenerate gases and warm dense matter.

Functions

- `deBroglie_wavelength(V, particle)` Calculates the de Broglie wavelength.
- `thermal_deBroglie_wavelength(T_e)` Calculate the thermal deBroglie wavelength for electrons.
- `Fermi_energy(n_e)` Calculate the kinetic energy in a degenerate electron gas.
- `Thomas_Fermi_length(n_e)` Calculate the exponential scale length for charge screening for cold and dense plasmas.
- `Wigner_Seitz_radius(n)` Calculate the Wigner-Seitz radius, which approximates the inter- particle spacing.
- `chemical_potential(n_e, T)` Calculate the ideal chemical potential.
**deBroglie_wavelength**

`plasmapy.formulary.quantum.deBroglie_wavelength(V: Unit("m / s"), particle) -> Unit("m")`

Calculates the de Broglie wavelength.

Parameters

- `V` (`Quantity`) – Particle velocity in units convertible to meters per second.
- `particle` (`str` or `Quantity`) – Representation of the particle species (e.g., 'e', 'p', 'D+', or 'He-4 1+', or the particle mass in units convertible to kilograms).

Returns `lambda_dB` – The de Broglie wavelength in units of meters.

Return type `Quantity`

Raises

- `TypeError` – The velocity is not a `Quantity` and cannot be converted into a `~astropy.units.Quantity`.
- `UnitConversionError` – If the velocity is not in appropriate units.
- `RelativityError` – If the magnitude of `V` is faster than the speed of light.

Warns `~astropy.units.UnitsWarning` – If units are not provided, SI units are assumed.

Notes

The de Broglie wavelength is given by

\[ \lambda_{dB} = \frac{h}{p} = \frac{h}{\gamma m V} \]

where \( h \) is the Planck constant, \( p \) is the relativistic momentum of the particle, \( \gamma \) is the Lorentz factor, \( m \) is the particle’s mass, and \( V \) is the particle’s velocity.

Examples

```python
>>> from astropy import units as u
>>> velocity = 1.4e7 * u.m / u.s
>>> deBroglie_wavelength(velocity, 'e')
<Quantity 5.18997095e-11 m>
```

**thermal_deBroglie_wavelength**

`plasmapy.formulary.quantum.thermal_deBroglie_wavelength(T_e: Unit("K")) -> Unit("m")`

Calculate the thermal deBroglie wavelength for electrons.

Parameters `T_e` (`Quantity`) – Electron temperature.

Returns `lambda_dbTh` – The thermal deBroglie wavelength for electrons in meters.

Return type `Quantity`
PlasmaPy Documentation, Release 0.2.0

Raises

- **TypeError** – If argument is not a **Quantity**.
- **UnitConversionError** – If argument is in incorrect units.
- **ValueError** – If argument contains invalid values.

**Warns** - *astropy.units.UnitsWarning* – If units are not provided, SI units are assumed.

Notes

The thermal de Broglie wavelength is approximately the average de Broglie wavelength for electrons in an ideal gas and is given by

$$\lambda_{dbTh} = \frac{\hbar}{\sqrt{2\pi m_e k_B T_e}}$$

Example

```python
>>> from astropy import units as u
>>> thermal_deBroglie_wavelength(1 * u.eV)
<Quantity 6.9193675e-10 m>
```

**Fermi_energy**

plasmapy.formulary.quantum.Fermi_energy(n_e: Unit("1 / m^3")) -> Unit("J")

Calculate the kinetic energy in a degenerate electron gas.

**Parameters**

- **n_e** (Quantity) – Electron number density.

**Returns**

- **energy_F** – The Fermi energy in Joules.

**Return type**

**Quantity**

**Raises**

- **TypeError** – If argument is not a **Quantity**.
- **UnitConversionError** – If argument is in incorrect units.
- **ValueError** – If argument contains invalid values.

**Warns** - *astropy.units.UnitsWarning* – If units are not provided, SI units are assumed.

Notes

The Fermi energy is the kinetic energy in a degenerate electron gas and is given by

$$E_F = \frac{\pi^2 \hbar^2}{2m_e} \left( \frac{3n_e}{\pi} \right)^{2/3}$$

This quantity is often used in place of thermal energy for analysis of cold, dense plasmas (e.g. warm dense matter, condensed matter).

**See also:**

- Thomas_Fermi_length()
**Example**

```python
>>> from astropy import units as u
>>> Fermi_energy(1e23 * u.cm**-3)
<Quantity 1.2586761e-18 J>
```

**Thomas_Fermi_length**

`plasmapy.formulary.quantum.Thomas_Fermi_length(n_e: Unit("1 / m3")) -> Unit("m")`

Calculate the exponential scale length for charge screening for cold and dense plasmas.

**Parameters**

- `n_e (Quantity)` – Electron number density.

**Returns**

- `lambda_TF` – The Thomas-Fermi screening length in meters.

**Return type**

`Quantity`

**Raises**

- `TypeError` – If argument is not a `Quantity`.
- `UnitConversionError` – If argument is in incorrect units.
- `ValueError` – If argument contains invalid values.

**Notes**

The Thomas-Fermi screening length is the exponential scale length for charge screening and is given by

\[
\lambda_{TF} = \sqrt{\frac{2e_0 E_F}{3n_e e^2}}
\]

for an electron degenerate gas.

This quantity is often used in place of the Debye length for analysis of cold, dense plasmas (e.g. warm dense matter, condensed matter).

The electrical potential will drop by a factor of 1/e every Thomas-Fermi screening length.

Plasmas will generally be quasineutral on length scales significantly larger than the Thomas-Fermi screening length.

**See also:**

`Fermi_energy()`, `plasmapy.formulary.Debye_length()`

**Example**

```python
>>> from astropy import units as u
>>> Thomas_Fermi_length(1e23 * u.cm**-3)
<Quantity 5.37991409e-11 m>
```
Wigner_Seitz_radius

Calculate the Wigner-Seitz radius, which approximates the inter-particle spacing. It is the radius of a sphere whose volume is equal to the mean volume per atom in a solid. This parameter is often used to calculate the coupling parameter. When ion density is used, this is the ion sphere radius, i.e., the space occupied by a single ion with no other ions in that space. Higher density means less space for each ion, so the radius is smaller.

**Parameters**

- **n** (*Quantity*) – Particle number density.

**Returns**

- **radius** – The Wigner-Seitz radius in meters.

**Return type**

*Quantity*

**Raises**

- **TypeError** – If argument is not a ~astropy.units.Quantity.
- **UnitConversionError** – If argument is in incorrect units.
- **ValueError** – If argument contains invalid values.

**Warns**

- ~astropy.units.UnitsWarning – If units are not provided, SI units are assumed.

**Notes**

The Wigner-Seitz radius approximates the interparticle spacing. It is the radius of a sphere whose volume is equal to the mean volume per atom in a solid:

$$r = \left( \frac{3}{4\pi n} \right)^{1/3}$$

**See also:**

Fermi_energy()

**Example**

```python
>>> from astropy import units as u
>>> Wigner_Seitz_radius(1e29 * u.m**-3)
<Quantity 1.33650462e-10 m>
```

chemical_potential

Calculate the ideal chemical potential.

**Parameters**

- **n_e** (*Quantity*) – Electron number density.
- **T** (*Quantity*) – The temperature.

**Returns**

- **beta_mu** – The dimensionless ideal chemical potential. That is the ratio of the ideal chemical potential to the thermal energy.

**Return type**

*Quantity*
PlasmaPy Documentation, Release 0.2.0

Raises

• **TypeError** – If argument is not a *Quantity*.
• **UnitConversionError** – If argument is in incorrect units.
• **ValueError** – If argument contains invalid values.

**Warns** *-astropy.units.UnitsWarning* – If units are not provided, SI units are assumed.

**Notes**

The ideal chemical potential is given by¹:

\[
\chi_a = I_{1/2}(\beta \mu_{\text{ideal}})
\]

where \( \chi \) is the degeneracy parameter, \( I_{1/2} \) is the Fermi integral with order 1/2, \( \beta \) is the inverse thermal energy \( \beta = 1/(k_B T) \), and \( \mu_{\text{ideal}} \) is the ideal chemical potential.

The definition for the ideal chemical potential is implicit, so it must be obtained numerically by solving for the Fermi integral for values of chemical potential approaching the degeneracy parameter. Since values returned from the Fermi_integral are complex, a nonlinear Levenberg-Marquardt least squares method is used to iteratively approach a value of \( \mu \) which minimizes \( I_{1/2}(\beta \mu_{\text{ideal}}) - \chi_a \)

This function returns \( \beta \mu_{\text{ideal}} \) the dimensionless ideal chemical potential.

Warning: at present this function is limited to relatively small arguments due to limitations in the *mpmath* package’s implementation of *polylog*, which PlasmaPy uses in calculating the Fermi integral.

**References**

**Example**

```python
>>> from astropy import units as u
>>> chemical_potential(n_e=1e21*u.cm**-3,T=11000*u.K)  # doctest: +SKIP
<Quantity 2.00039985e-12>
```

**Relativistic functions (plasmapy.formulary.relativity)**

This module currently provides ample room for the Lorentz factor, as it turned out we didn’t really have much else of the relativistic variety to add just yet! This is expected to change in the future.

**Functions**

<table>
<thead>
<tr>
<th><strong>Lorentz_factor(V)</strong></th>
<th>Return the Lorentz factor.</th>
</tr>
</thead>
</table>

**Lorentz_factor**

*plasmapy.formulary.relativity.Lorentz_factor (V: Unit("m / s"))*

Return the Lorentz factor.

Parameters $\mathbf{V}$ (Quantity) – The velocity in units convertible to meters per second.

Returns gamma – The Lorentz factor associated with the inputted velocities.

Return type float or ndarray

Raises

- TypeError – The $\mathbf{V}$ is not a Quantity and cannot be converted into a ~astropy.units.Quantity.
- UnitConversionError – If the $\mathbf{V}$ is not in appropriate units.
- ValueError – If the magnitude of $\mathbf{V}$ is faster than the speed of light.

Warns ~astropy.units.UnitsWarning – If units are not provided, SI units are assumed.

Notes

The Lorentz factor is a dimensionless number given by

$$\gamma = \frac{1}{\sqrt{1 - \frac{V^2}{c^2}}}$$

The Lorentz factor is approximately one for sub-relativistic velocities, and goes to infinity as the velocity approaches the speed of light.

Examples

```python
>>> from astropy import units as u
>>> velocity = 1.4e8 * u.m / u.s
>>> Lorentz_factor(velocity)
1.130885603948959
>>> Lorentz_factor(299792458*u.m/u.s)
inf
```

Classical transport theory (plasmapy.formulary.braginskii)

Functions to calculate classical transport coefficients.

Examples:

- Braginskii coefficients

Introduction

Classical transport theory is derived by using kinetic theory to close the plasma two-fluid (electron and ion fluid) equations in the collisional limit. The first complete model in this form was done by S. I. Braginskii

This module uses fitting functions from literature \(^1,^2,^3,^4,^5\) and the next section) to calculate the transport coefficients, which are the resistivity, thermoelectric conductivity, thermal conductivity, and viscosity.

Keep in mind the following assumptions under which the transport equations are derived:

1. The plasma is fully ionized, only consisting of ions and electrons. Neutral atoms are neglected.
2. Turbulent transport does not dominate

3. **The velocity distribution is close to Maxwellian. This implies:**
   - i. Collisional mean free path << gradient scale length along field
   - ii. Gyroradius << gradient scale length perpendicular to field
4. The plasma is highly collisional: collisional frequency >> gyrofrequency

When classical transport is not valid, e.g. due to the presence of strong gradients or turbulent transport, the transport is significantly increased by these other effects. Thus classical transport often serves as a lower bound on the losses / transport encountered in a plasma.

**Transport Variables**

For documentation on the individual transport variables, please take the following links to documentation of methods of `ClassicalTransport`.

- Resistivity
- Thermoelectric conductivity
- Ion thermal conductivity
- Electron thermal conductivity
- Ion viscosity
- Electron viscosity

**Using the module**

Given that many of the transport variables share a lot of the same computation and many are often needed to be calculated simultaneously, this module provides a `ClassicalTransport` class that can be initialized once with all of the variables necessary for calculation. It then provides all of the functionality as methods (please refer to its documentation).

If you only wish to calculate a single transport variable (or if just don’t like object oriented interfaces), we have also provided wrapper functions in the main module namespace that use `ClassicalTransport` under the hood (see below, in the Functions section).

**Warning:** Please note that as PlasmaPy is a very new package, this API should not be considered stable yet.

\(^3\) Physics of Fully Ionized Gases, L. Spitzer (1962)
Classical transport models

In this section, we present a broad overview of classical transport models implemented within this module.

Braginskii

The original Braginskii treatment as presented in the highly cited review paper from 1965. Coefficients are found from expansion of the kinetic equation in Laguerre polynomials, truncated at the second term in their series expansion ($k = 2$). This theory allows for arbitrary Hall parameter and include results for $Z = 1, 2, 3, 4,$ and infinity (the case of Lorentz gas completely stripped of electrons, and the stationary ion approximation).

Spitzer-Harm

These coefficients were obtained from a numerical solution of the Fokker-Planck equation. They give one of the earliest and most accurate (in the Fokker-Planck sense) results for electron transport in simple plasma. They principally apply in the unmagnetized / parallel field case, although for resistivity Spitzer also calculated a famous result for a strong perpendicular magnetic field. Results are for $Z = 1, 2, 4, 16,$ and infinity (Lorentz gas / stationary ion approximation).

Epperlein-Haines

Not yet implemented.

Ji-Held

This is a modern treatment of the classical transport problem that has been carried out with laudable care. It allows for arbitrary hall parameter and arbitrary $Z$ for all coefficients. Similar to the Epperlein-Haines model, it corrects some known inaccuracies in the original Braginskii results, notably the asymptotic behavior of $\alpha_{cross}$ and $\beta_{perp}$ as $\text{Hall} \to +\infty$. It also studies effects of electron collisions in the ion terms, which all other treatments have not. To neglect electron-electron collisions, leave $\mu = 0$. To consider them, specify $\mu$ and $\theta$.

References

Functions

- `resistivity(T_e, n_e, T_i, n_i, ion_particle)` Calculate the resistivity.
- `thermoelectric_conductivity(T_e, n_e, T_i, ...)` Calculate the thermoelectric conductivity.
- `ion_thermal_conductivity(T_e, n_e, T_i, n_i, ...)` Calculate the thermal conductivity for ions.
- `electron_thermal_conductivity(T_e, n_e, T_i, ...)` Calculate the thermal conductivity for electrons.
- `ion_viscosity(T_e, n_e, T_i, n_i, ion_particle)` Calculate the ion viscosity.
- `electron_viscosity(T_e, n_e, T_i, n_i, ...)` Calculate the electron viscosity.
resistivity

plasmapy.formulary.braginskii.resistivity(T_e, n_e, T_i, n_i, ion_particle, m_i=None, Z=None, B: Unit("T") = <Quantity 0. T>, model='Braginskii', field_orientation='parallel', mu=None, theta=None, coulomb_log_method='classical') -> Unit("m Ohm")

Calculate the resistivity.

Notes

The resistivity here is defined similarly to solid conductors, and thus represents the classical plasmas’ property
to resist the flow of electrical current. The result is in units of ohm * m, so if you assume where the current is
flowing in the plasma (length and cross-sectional area), you could calculate a DC resistance of the plasma in
ohms as resistivity * length / cross-sectional area.

Experimentalists with plasma discharges may observe different V = IR Ohm’s law behavior than suggested by
the resistance calculated here, for reasons such as the occurrence of plasma sheath layers at the electrodes or the
plasma not satisfying the classical assumptions.

Returns

Return type: astropy.units.quantity.Quantity

thermoelectric_conductivity

plasmapy.formulary.braginskii.thermoelectric_conductivity(T_e, n_e, T_i, n_i, ion_particle, m_i=None, Z=None, B: Unit("T") = <Quantity 0. T>, model='Braginskii', field_orientation='parallel', mu=None, theta=None, coulomb_log_method='classical')

Calculate the thermoelectric conductivity.

ion_thermal_conductivity

plasmapy.formulary.braginskii.ion_thermal_conductivity(T_e, n_e, T_i, n_i, ion_particle, m_i=None, Z=None, B: Unit("T") = <Quantity 0. T>, model='Braginskii', field_orientation='parallel', mu=None, theta=None, coulomb_log_method='classical') -> Unit("W / (K m)")

Calculate the thermal conductivity for ions.
### Notes

This is the classical plasma ions’ ability to conduct energy and heat, defined similarly to other materials. The result is a conductivity in units of $\text{W} / \text{m} / \text{K}$, so if you assume you know where the heat is flowing (temperature gradient, cross-sectional area) you can calculate the energy transport in Watts as conductivity * cross-sectional area * temperature gradient. In lab plasmas, typically the energy is flowing out of your high-temperature plasma to something else, like the walls of your device, and you are sad about this.

**Returns**

- **Return type**: `astropy.units.quantity.Quantity`

**See also:**

- `ion_thermal_conductivity()`

### electron_thermal_conductivity

```
plasmapy.formulary.braginskii.electron_thermal_conductivity(T_e, n_e, T_i, n_i, ion_particle, m_i=None, Z=None, B: Unit("T") = <Quantity 0. T>, model='Braginskii', field_orientation='parallel', mu=None, theta=None, coulomb_log_method='classical') -> Unit("W / (K m)")
```

Calculate the thermal conductivity for electrons.

### Notes

This is quite similar to the ion thermal conductivity, except that it’s for the plasma electrons. In a typical unmagnetized plasma, the electron thermal conductivity is much higher than the ions and will dominate, due to the electrons’ low mass and fast speeds.

In a strongly magnetized plasma, following the classical transport analysis, you calculate that the perpendicular-field thermal conductivity becomes greatly reduced for the ions and electrons, with the electrons actually being restrained even more than the ions due to their low mass and small gyroradius. In reality, the electrons and ions are pulling on each other strongly due to their opposing charges, so you have the situation of ambipolar diffusion.

This situation has been likened to an energetic little child (the electrons) not wanting to be pulled away from the playground (the magnetic field) by the parents (the ions).

The ultimate rate must typically be in between the individual rates for electrons and ions, so at least you can get some bounds from this type of analysis.

**Returns**

- **Return type**: `astropy.units.quantity.Quantity`

**See also:**

- `ion_thermal_conductivity()`
**ion viscosity**

plasmapy.formulary.braginskii.ion_viscosity(T_e, n_e, T_i, n_i, ion_particle, m_i=None, Z=None, B: Unit("T") = <Quantity 0. T>, model='Braginskii', field_orientation='parallel', mu=None, theta=None, coulomb_log_method='classical') -> Unit("Pa s")

Calculate the ion viscosity.

**Notes**

This is the dynamic viscosity that you find for ions in the classical plasma, similar to the viscosity of air or water or honey. The big effect is the $T^{5/2}$ dependence, so as classical plasmas get hotter they become dramatically more viscous. The ion viscosity typically dominates over the electron viscosity.

**Returns**

**Return type**  astropy.units.quantity.Quantity

**See also:**

electron_viscosity()

**electron viscosity**

plasmapy.formulary.braginskii.electron_viscosity(T_e, n_e, T_i, n_i, ion_particle, m_i=None, Z=None, B: Unit("T") = <Quantity 0. T>, model='Braginskii', field_orientation='parallel', mu=None, theta=None, coulomb_log_method='classical') -> Unit("Pa s")

Calculate the electron viscosity.

**Notes**

This is the dynamic viscosity that you find for electrons in the classical plasma, similar to the viscosity of air or water or honey. The big effect is the $T^{5/2}$ dependence, so as classical plasmas get hotter they become dramatically more viscous. The ion viscosity typically dominates over the electron viscosity.

**Returns**

**Return type**  astropy.units.quantity.Quantity

**See also:**

ion_viscosity()
Classical transport coefficients (e.g. Braginskii, 1965).

Notes

Given that many of the transport variables share a lot of the same computation and many are often needed to be calculated simultaneously, this class can be initialized once with all of the variables necessary for calculation. It then provides all of the functionality as methods (please refer to their documentation).

Parameters

- **T_e (Quantity)** – Electron temperature in units of temperature or energy per particle
- **n_e (Quantity)** – The electron number density in units convertible to per cubic meter.
- **T_i (Quantity)** – Ion temperature in units of temperature or energy per particle
- **n_i (Quantity)** – The ion number density in units convertible to per cubic meter.
- **ion_particle (string)** – Representation of the ion species (e.g., ‘p’ for protons, ‘e’ for electrons, ‘D+’ for deuterium, or ‘He-4 +1’ for singly ionized helium-4). If no charge state information is provided, then the particles are assumed to be singly charged.
- **Z (int or np.inf, optional)** – The ion charge state. Overrides particle charge state if included. Different theories support different values of Z. For the original Braginskii model, Z can be any of [1, 2, 3, 4, infinity]. The Ji-Held model supports arbitrary Z. Average ionization states Z_mean can be input using this input and the Ji-Held model, although doing so may neglect effects caused by multiple ion populations.
- **B (Quantity, optional)** – The magnetic field strength in units convertible to tesla. Defaults to zero.
- **model (string)** – Indication of whose formulation from literature to use. Allowed values are:
  - ‘Braginskii’,
  - ‘Spitzer-Harm’,
- ‘Epperlein-Haines’ (not yet implemented),
- ‘Ji-Held’.

See refs.\(^1\),\(^2\),\(^3\),\(^4\), and\(^5\).

- **field_orientation** (string, defaults to ‘parallel’) – Either of ‘parallel’, ‘par’, ‘perpendicular’, ‘perp’, ‘cross’, or ‘all’, indicating the cardinal orientation of the magnetic field with respect to the transport direction of interest. Note that ‘perp’ refers to transport perpendicular to the field direction (in the direction of the temperature gradient), while ‘cross’ refers to the direction perpendicular to \(B\) and the gradient of temperature \((B \times \nabla(T))\). The option ‘all’ will return a Numpy array of all three, `np.array((par, perp, cross))`. Does not apply to viscosities.

- **coulomb_log_ei** (float or dimensionless `Quantity`, optional) – Force a particular value to be used for the electron-ion Coulomb logarithm (test electrons on field ions). If `None`, `Coulomb_logarithm` will be used. Useful for comparing calculations.

- **V_ei**(`Quantity`, optional) – The relative velocity between particles. Supplied to `Coulomb_logarithm` function, not otherwise used. If not provided, thermal velocity is assumed: \(\mu V^2 \sim 2k_B T\) where \(\mu\) is the reduced mass.

- **coulomb_log_ii** (float or dimensionless `Quantity`, optional) – Force a particular value to be used for the ion-ion Coulomb logarithm (test ions on field ions). If `None`, the PlasmaPy function `Coulomb_logarithm` will be used. Useful for comparing calculations.

- **V_ii**(`Quantity`, optional) – The relative velocity between particles. Supplied to `Coulomb_logarithm` function, not otherwise used. If not provided, thermal velocity is assumed: \(\mu V^2 \sim 2k_B T\) where \(\mu\) is the reduced mass.

- **hall_e** (float or dimensionless `Quantity`, optional) – Force a particular value to be used for the electron Hall parameter. If `None`, `Hall_parameter` will be used. Useful for comparing calculations.

- **hall_i** (float or dimensionless `Quantity`, optional) – Force a particular value to be used for the ion Hall parameter. If `None`, `Hall_parameter` will be used. Useful for comparing calculations.

- **mu** (optional, float or dimensionless `astropy.units.Quantity`) – Ji-Held model only, may be used to include ion-electron effects on the ion transport coefficients. Defaults to zero, thus disabling these effects.

- **theta** (optional, float or dimensionless `Quantity`) – Ji-Held model only, may be used to include ion-electron effects on the ion transport coefficients. Defaults to \(T_e / T_i\). Only has effect if \(\mu\) is non-zero.

**Raises**

- `ValueError` – On incorrect or unknown values of arguments.

- `plasmapy.utils.PhysicsError` – If input or calculated values for Coulomb logarithms are nonphysical.

---


\(^3\) Physics of Fully Ionized Gases, L. Spitzer (1962)


Examples

```python
>>> from astropy import units as u
>>> t = ClassicalTransport(1*u.eV, 1e20/u.m**3,
...                         1*u.eV, 1e20/u.m**3, 'p')
>>> t.resistivity
<Quantity 0.0003670... m Ohm>
>>> t.thermoelectric_conductivity
<Quantity 0.71108...>
>>> t.ion_thermal_conductivity
<Quantity 0.01552... W / (K m)>
>>> t.electron_thermal_conductivity
<Quantity 0.38064... W / (K m)>
>>> t.ion_viscosity
<Quantity [4.621297...e-07, 4.607248...e-07, 4.607248...e-07, 0.000000...e+00,
0.000000...e+00] Pa s>
>>> t.electron_viscosity
<Quantity [5.822738...e-09, 5.820820...e-09, 5.820820...e-09, 0.000000...e+00,
0.000000...e+00] Pa s>
```

References

Attributes Summary

<table>
<thead>
<tr>
<th>all_variables</th>
<th>Return all transport variables as a dictionary.</th>
</tr>
</thead>
<tbody>
<tr>
<td>electron_thermal_conductivity</td>
<td>Calculate the thermal conductivity for electrons.</td>
</tr>
<tr>
<td>electron_viscosity</td>
<td>Calculate the electron viscosity.</td>
</tr>
<tr>
<td>ion_thermal_conductivity</td>
<td>Calculate the thermal conductivity for ions.</td>
</tr>
<tr>
<td>ion_viscosity</td>
<td>Calculate the ion viscosity.</td>
</tr>
<tr>
<td>resistivity</td>
<td>Calculate the resistivity.</td>
</tr>
<tr>
<td>thermoelectric_conductivity</td>
<td>Calculate the thermoelectric conductivity.</td>
</tr>
</tbody>
</table>

Attributes Documentation

all_variables

Return all transport variables as a dictionary.

Returns

Return type dict

electron_thermal_conductivity

Calculate the thermal conductivity for electrons.

Notes

This is quite similar to the ion thermal conductivity, except that it’s for the plasma electrons. In a typical unmagnetized plasma, the electron thermal conductivity is much higher than the ions and will dominate, due to the electrons’ low mass and fast speeds.

In a strongly magnetized plasma, following the classical transport analysis, you calculate that the perpendicular-field thermal conductivity becomes greatly reduced for the ions and electrons, with the electrons actually being restrained even more than the ions due to their low mass and small gyroradius. In
PlasmaPy Documentation, Release 0.2.0

reality, the electrons and ions are pulling on each other strongly due to their opposing charges, so you have the situation of ambipolar diffusion.

This situation has been likened to an energetic little child (the electrons) not wanting to be pulled away from the playground (the magnetic field) by the parents (the ions).

The ultimate rate must typically be in between the individual rates for electrons and ions, so at least you can get some bounds from this type of analysis.

**Returns**

**Return type** astropy.units.quantity.Quantity

**See also:**

- ion_thermal_conductivity
- electron_viscosity

**Notes**

This is the dynamic viscosity that you find for electrons in the classical plasma, similar to the viscosity of air or water or honey. The big effect is the \( T^{5/2} \) dependence, so as classical plasmas get hotter they become dramatically more viscous. The ion viscosity typically dominates over the electron viscosity.

**Returns**

**Return type** astropy.units.quantity.Quantity

**See also:**

- ion_viscosity
- ion_thermal_conductivity

**Notes**

This is the classical plasma ions’ ability to conduct energy and heat, defined similarly to other materials. The result is a conductivity in units of \( \text{W} / \text{m} / \text{K} \), so if you assume you know where the heat is flowing (temperature gradient, cross-sectional area) you can calculate the energy transport in Watts as conductivity * cross-sectional area * temperature gradient. In lab plasmas, typically the energy is flowing out of your high-temperature plasma to something else, like the walls of your device, and you are sad about this.

**Returns**

**Return type** astropy.units.quantity.Quantity

**See also:**

- ion_thermal_conductivity
- ion_viscosity

**Notes**

Calculate the electron viscosity.

**Notes**

Calculate the thermal conductivity for ions.

**Notes**

Calculate the ion viscosity.
Notes

This is the dynamic viscosity that you find for ions in the classical plasma, similar to the viscosity of air or water or honey. The big effect is the $T^{5/2}$ dependence, so as classical plasmas get hotter they become dramatically more viscous. The ion viscosity typically dominates over the electron viscosity.

Returns

Return type  astropy.units.quantity.Quantity

See also:

electron_viscosity

resistivity

Calculate the resistivity.

Notes

The resistivity here is defined similarly to solid conductors, and thus represents the classical plasmas’ property to resist the flow of electrical current. The result is in units of ohm * m, so if you assume where the current is flowing in the plasma (length and cross-sectional area), you could calculate a DC resistance of the plasma in ohms as resistivity * length / cross-sectional area.

Experimentalists with plasma discharges may observe different $V = IR$ Ohm’s law behavior than suggested by the resistance calculated here, for reasons such as the occurrence of plasma sheath layers at the electrodes or the plasma not satisfying the classical assumptions.

Returns

Return type  astropy.units.quantity.Quantity

thermoelectric_conductivity

Calculate the thermoelectric conductivity.

Notes

To be improved.

Returns

Return type  astropy.units.quantity.Quantity

Class Inheritance Diagram
Collisions *(plasmapy.formulary.collisions)*

Functions to calculate transport coefficients.

This module includes a number of functions for handling Coulomb collisions spanning weakly coupled (low density) to strongly coupled (high density) regimes.

**Coulomb collisions**

Coulomb collisions are collisions where the interaction force is conveyed via the electric field, instead of any kind of contact force. They usually result in relatively small deflections of particle trajectories. However, given that there are many charged particles in a plasma, one has to take into account the cumulative effects of many such collisions.

**Coulomb logarithms**

Please see the documentation for the `Coulomb_logarithm` for a review of the many ways in which one can define and calculate that quantity.

**Collision rates**

The module gathers a few functions helpful for calculating collision rates between particles. The most general of these is `collision_frequency`, while if you need average values for a Maxwellian distribution, try out `collision_rate_electron_ion` and `collision_rate_ion_ion`. These use `collision_frequency` under the hood.

**Macroscopic properties**

These include:

- `Spitzer_resistivity`
- `mobility`
- `Knudsen_number`
- `coupling_parameter`

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Coulomb_logarithm(T, n_e, particles, ... [, ...])</code></td>
<td>Estimates the Coulomb logarithm.</td>
</tr>
<tr>
<td><code>impact_parameter_perp(T, particles, &lt;class &gt;), V)</code></td>
<td>Distance of closest approach for a 90 degree Coulomb collision.</td>
</tr>
<tr>
<td><code>impact_parameter(T, n_e, particles, z_mean, V)</code></td>
<td>Impact parameters for classical and quantum Coulomb collision</td>
</tr>
<tr>
<td><code>collision_frequency(T, n, particles, z_mean, V)</code></td>
<td>Collision frequency of particles in a plasma.</td>
</tr>
<tr>
<td><code>Coulomb_cross_section(impact_param)</code></td>
<td>Cross section for a large angle Coulomb collision.</td>
</tr>
<tr>
<td><code>fundamental_electron_collision_freq(T_e, ...)</code></td>
<td>Average momentum relaxation rate for a slowly flowing Maxwellian distribution of electrons.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 20 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fundamental_ion_collision_freq(T_i, n_i, ...)</td>
<td>Average momentum relaxation rate for a slowly flowing Maxwellian distribution of ions.</td>
</tr>
<tr>
<td>mean_free_path(T, n_e, particles, z_mean, V)</td>
<td>Collisional mean free path (m)</td>
</tr>
<tr>
<td>Spitzer_resistivity(T, n, particles, z_mean, V)</td>
<td>Spitzer resistivity of a plasma</td>
</tr>
<tr>
<td>mobility(T, n_e, particles, z_mean, V)</td>
<td>Electrical mobility (m^2/(V s))</td>
</tr>
<tr>
<td>Knudsen_number(characteristic_length, T, ...)</td>
<td>Knudsen number (dimensionless)</td>
</tr>
<tr>
<td>coupling_parameter(T, n_e, particles, z_mean, V)</td>
<td>Coupling parameter</td>
</tr>
</tbody>
</table>

Coulomb_logarithm

plasmapy.formulary.collisions.Coulomb_logarithm(T: Unit("K"), n_e: Unit("1 / m^3"), particles: ("<class 'plasmapy.atomic.particle_class.Particle'>", "<class 'plasmapy.atomic.particle_class.Particle'>"), z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical')

Estimates the Coulomb logarithm.

Parameters

- **T** *(Quantity)* – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle.
- **n_e** *(Quantity)* – The electron density in units convertible to per cubic meter.
- **particles** *(tuple)* – A tuple containing string representations of the test particle (listed first) and the target particle (listed second).
- **z_mean** *(Quantity, optional)* – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.
- **V** *(Quantity, optional)* – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.
- **method** *(str, optional)* – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

Returns **lnLambda** – An estimate of the Coulomb logarithm that is accurate to roughly its reciprocal.

Return type **float or numpy.ndarray**

Raises

- **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.
- **UnitConversionError** – If the units on any of the inputs are incorrect.
  - If the n_e, T, or V are not Quantities.
- **PhysicsError** – If the result is smaller than 1.
- **RelativityError** – If the input velocity is same or greater than the speed of light.
Warnings

- `~astropy.units.UnitsWarning` – If units are not provided, SI units are assumed
- `~plasmapy.utils.RelativityWarning` – If the input velocity is greater than 5% of the speed of light.

Notes

The classical Coulomb logarithm is given by

$$\ln \Lambda \equiv \ln \left( \frac{b_{\text{max}}}{b_{\text{min}}} \right)$$

where $b_{\text{min}}$ and $b_{\text{max}}$ are the inner and outer impact parameters for Coulomb collisions\(^1\).

The outer impact parameter is given by the Debye length: $b_{\text{min}} = \lambda_D$ which is a function of electron temperature and electron density. At distances greater than the Debye length, electric fields from other particles will be screened out due to electrons rearranging themselves.

The choice of inner impact parameter is either the distance of closest approach for a 90 degree Coulomb collision or the thermal deBroglie wavelength, whichever is larger. This is because Coulomb-style collisions cannot occur for impact parameters shorter than the deBroglie wavelength because quantum effects will change the fundamental nature of the collision\(^2,3\).

Errors associated with the classical Coulomb logarithm are of order its inverse. If the Coulomb logarithm is of order unity, then the assumptions made in the standard analysis of Coulomb collisions are invalid.

For dense plasmas where the classical Coulomb logarithm breaks down there are various extended methods. These can be found in D.O. Gericke et al’s paper, which has a table summarizing the methods\(^4\). The GMS-1 through GMS-6 methods correspond to the methods found it that table.

It should be noted that GMS-4 thru GMS-6 modify the Coulomb logarithm to the form:

$$\ln \Lambda \equiv 0.5 \ln \left( 1 + \frac{b_{\text{max}}^2}{b_{\text{min}}^2} \right)$$

This means the Coulomb logarithm will not break down for $\Lambda < 0$, which occurs for dense, cold plasmas.

\[\text{plasmapy.formulary.collisions.Classical()}\]

classical Landau-Spitzer approach. Fails for large coupling parameter where $\Lambda$ can become less than zero.

**GMS-1**

1st method listed in Table 1 of reference [3] Landau-Spitzer, but with interpolated bmin instead of bmin selected between deBroglie wavelength and distance of closest approach. Fails for large coupling parameter where $\Lambda$ can become less than zero.

**GMS-2**

2nd method listed in Table 1 of reference [3] Another Landau-Spitzer like approach, but now bmax is also being interpolated. The interpolation is between the Debye length and the ion sphere radius, allowing for descriptions of dilute plasmas. Fails for large coupling parameter where $\Lambda$ can become less than zero. 3rd method listed in Table 1 of reference [3] classical Landau-Spitzer fails for argument of Coulomb logarithm $\Lambda < 0$, therefore a clamp is placed at $\Lambda_{\text{min}} = 2$

---

\(^1\) Physics of Fully Ionized Gases, L. Spitzer (1962)
GMS-4

4th method listed in Table 1 of reference [3] Spitzer-like extension to Coulomb logarithm by noting that Coulomb collisions take hyperbolic trajectories. Removes divergence for small bmin issue in classical Landau-Spitzer approach, so bmin can be zero. Also doesn’t break down as Lambda < 0 is now impossible, even when coupling parameter is large.

GMS-5

5th method listed in Table 1 of reference [3] Similar to GMS-4, but setting bmin as distance of closest approach and bmax interpolated between Debye length and ion sphere radius. Lambda < 0 impossible.

GMS-6

6th method listed in Table 1 of reference [3] Similar to GMS-4 and GMS-5, but using interpolation methods for both bmin and bmax.

Examples

```python
>>> from astropy import units as u
>>> n = 1e19 * u.m**-3
>>> T = 1e6 * u.K
>>> particles = ('e', 'p')
>>> Coulomb_logarithm(T, n, particles)
14.545527...
>>> Coulomb_logarithm(T, n, particles, V=1e6 * u.m / u.s)
11.363478...
```

References

impact_parameter_perp

```python
plasmapy.formulary.collisions.impact_parameter_perp(T: Unit("K"), particles: ("e", "p")) -> Unit("m")
```

Distance of closest approach for a 90 degree Coulomb collision.

Parameters

- T (Quantity) – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle
- particles (tuple) – A tuple containing string representations of the test particle (listed first) and the target particle (listed second)
- V (Quantity, optional) – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.

Returns impact_parameter_perp – The distance of closest approach for a 90 degree Coulomb collision.

Return type float or numpy.ndarray

Raises

- ValueError – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.
• UnitConversionError – If the units on any of the inputs are incorrect
• TypeError – If T, or V are not Quantities.
• RelativityError – If the input velocity is same or greater than the speed of light.

Warns
• ~astropy.units.UnitsWarning – If units are not provided, SI units are assumed
• ~plasmapy.utils.RelativityWarning – If the input velocity is greater than 5% of the speed of light.

Notes

The distance of closest approach, impact_parameter_perp, is given by\(^1\)

\[
b_\perp = \frac{Z_1 Z_2}{4\pi\varepsilon_0 m v^2}
\]

Examples

```python
>>> from astropy import units as u
>>> T = 1e6*u.K
>>> particles = ('e', 'p')
>>> impact_parameter_perp(T, particles)
<Quantity 8.3550...e-12 m>
```

References

impact_parameter

plasmapy.formulary.collisions.impact_parameter(T: Unit("K"), n_e: Unit("1 / m3"), particles, z_mean: Unit(dimensionless)
= <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical')

Impact parameters for classical and quantum Coulomb collision

Parameters

• T (Quantity) – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle

• n_e (Quantity) – The electron density in units convertible to per cubic meter.

• particles (tuple) – A tuple containing string representations of the test particle (listed first) and the target particle (listed second)

• z_mean (Quantity, optional) – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.

• V (Quantity, optional) – The relative velocity between particles. If not provided, thermal velocity is assumed: \(\mu V^2 \sim 2k_B T\) where \(\mu\) is the reduced mass.

\(^1\) Francis, F. Chen. Introduction to plasma physics and controlled fusion 3rd edition. Ch 5 (Springer 2015).
• **method** *(str, optional)* – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

**Returns** *bmin, bmax* – The minimum and maximum impact parameters (distances) for a Coulomb collision.

**Return type** *tuple of floats*

**Raises**

• **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.

• **UnitConversionError** – If the units on any of the inputs are incorrect

• **TypeError** – If the n_e, T, or V are not Quantities.

• **RelativityError** – If the input velocity is same or greater than the speed of light.

**Warns**

• **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed

• **~plasmapy.utils.RelativityWarning** – If the input velocity is greater than 5% of the speed of light.

**Notes**

The minimum and maximum impact parameters may be calculated in a variety of ways. The maximum impact parameter is typically the Debye length.

For quantum plasmas the maximum impact parameter can be the quadratic sum of the debye length and ion radius (Wigner_Seitz)\(^1\)

\[
\begin{align*}
    b_{\text{max}} &= (\lambda_{\text{De}}^2 + a_i^2)^{1/2}
\end{align*}
\]

The minimum impact parameter is typically some combination of the thermal deBroglie wavelength and the distance of closest approach for a 90 degree Coulomb collision. A quadratic sum is used for all GMS methods, except for GMS-5, where b_min is simply set to the distance of closest approach\(^1\).

\[
\begin{align*}
    b_{\text{min}} &= (\Lambda_{\text{deBroglie}}^2 + \rho_{\perp}^2)^{1/2}
\end{align*}
\]

**Examples**

```python
>>> from astropy import units as u
>>> n = 1e19 * u.m**-3
>>> T = 1e6 * u.K
>>> particles = ('e', 'p')
>>> impact_parameter(T, n, particles)
(<Quantity 1.051...e-11 m>, <Quantity 2.182...e-05 m>)
>>> impact_parameter(T, n, particles, V=1e6 * u.m / u.s)
(<Quantity 2.534...e-10 m>, <Quantity 2.182...e-05 m>)
```

\(^1\) Dense plasma temperature equilibration in the binary collision approximation. D. O. Gericke et. al. PRE, 65, 036418 (2002). DOI: 10.1103/PhysRevE.65.036418
collision_frequency

plasmapy.formulary.collisions.collision_frequency (T: Unit("K"), n: Unit("1 / m3"), particles, z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical') -> Unit("Hz")

Collision frequency of particles in a plasma.

**Parameters**

**T** (*Quantity*) – Temperature in units of temperature. This should be the electron temperature for electron-electron and electron-ion collisions, and the ion temperature for ion-ion collisions.

**n** [*~astropy.units.Quantity*] The density in units convertible to per cubic meter. This should be the electron density for electron-electron collisions, and the ion density for electron-ion and ion-ion collisions.

**particles** [tuple] A tuple containing string representations of the test particle (listed first) and the target particle (listed second)

**z_mean** [*~astropy.units.Quantity*, optional] The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.

**V** [*~astropy.units.Quantity*, optional] The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.

**method**: str, optional Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

**Returns**

**freq** – The collision frequency of particles in a plasma.

**Return type** float or numpy.ndarray

**Raises**

- **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.
- **UnitConversionError** – If the units on any of the inputs are incorrect
- **TypeError** – If the n_e, T, or V are not Quantities.
- **RelativityError** – If the input velocity is same or greater than the speed of light.

**Warns**

- ~astropy.units.UnitsWarning – If units are not provided, SI units are assumed
- ~plasmapy.utils.RelativityWarning – If the input velocity is greater than 5% of the speed of light.

**Notes**

The collision frequency is given by

\[
\nu = n\sigma_v\ln \Lambda
\]

---

where \( n \) is the particle density, \( \sigma \) is the collisional cross-section, \( v \) is the inter-particle velocity (typically taken as the thermal velocity), and \( \ln \Lambda \) is the Coulomb logarithm accounting for small angle collisions.

See eq (2.14) in\(^2\).

**Examples**

```python
>>> from astropy import units as u
>>> n = 1e19*u.m**-3
>>> T = 1e6*u.K
>>> particles = ('e', 'p')
>>> collision_frequency(T, n, particles)
<Quantity 70249... Hz>
```

**References**

Coulomb_cross_section

`plasmapy.formulary.collisions.Coulomb_cross_section`

Cross section for a large angle Coulomb collision.

**Parameters**

- **impact_param** *(Quantity)* – Impact parameter for the collision.

**Examples**

```python
>>> Coulomb_cross_section(7e-10*u.m)
<Quantity 6.157...e-18 m2>
>>> Coulomb_cross_section(0.5*u.m)
<Quantity 3.141... m2>
```

**Notes**

The collisional cross-section (see\(^1\) for a graphical demonstration) for a 90 degree Coulomb collision is obtained by

\[ \sigma = \pi (2 \rho) ^ 2 \]

where \( \rho \) is the distance of closest approach for a 90 degree Coulomb collision. This function is a generalization of that calculation. Please note that it is not guaranteed to return the correct results for small angle collisions.

**Returns**

The Coulomb collision cross section area.

**Return type** *(Quantity)*
fundamental_electron_collision_freq

plasmapy.formulary.collisions.fundamental_electron_collision_freq(T_e:
    Unit("K"),
    n_e:
    Unit("1 / m3"),
    ion_particle, 
    coulomb_log=None,
    V=None,
    coulomb_log_method='classical') -> Unit("1 / s")

Average momentum relaxation rate for a slowly flowing Maxwellian distribution of electrons.

3 provides a derivation of this as an average collision frequency between electrons and ions for a Maxwellian distribution. It is thus a special case of the collision frequency with an averaging factor, and is on many occasions in transport theory the most relevant collision frequency that has to be considered. It is heavily related to diffusion and resistivity in plasmas.

Parameters

- **T_e** *(Quantity)* – The electron temperature of the Maxwellian test electrons
- **n_e** *(Quantity)* – The number density of the Maxwellian test electrons
- **ion_particle** *(str)* – String signifying a particle type of the field ions, including charge state information.
- **V** *(Quantity, optional)* – The relative velocity between particles. If not provided, thermal velocity is assumed: $\mu V^2 \sim 2k_B T$ where $\mu$ is the reduced mass.
- **coulomb_log** *(float or dimensionless ~astropy.units.Quantity, optional)* – Option to specify a Coulomb logarithm of the electrons on the ions. If not specified, the Coulomb log will is calculated using the Coulomb_logarithm function.
- **coulomb_log_method** *(string, optional)* – Method used for Coulomb logarithm calculation (see that function for more documentation). Choose from “classical” or “GMS-1” to “GMS-6”.

Notes

Equations (2.17) and (2.120) in3 provide the original source used to implement this formula, however, the simplest form that connects our average collision frequency to the general collision frequency is is this (from 2.17):  
$$\nu_e = \frac{4}{3\sqrt{\pi}} \nu(vT_e)$$

Where $\nu$ is the general collision frequency and $vT_e$ is the electron thermal velocity (the average, for a Maxwellian distribution).

This implementation of the average collision frequency is is equivalent to: * 1/tau_e from ref\(^1\) eqn (2.5e) pp. 215, * nu_e from ref\(^2\) pp. 33,

---

Examples

```python
>>> from astropy import units as u
>>> fundamental_electron_collision_freq(0.1 * u.eV, 1e6 / u.m ** 3, 'p')
<Quantity 0.001801... 1 / s>

>>> fundamental_electron_collision_freq(1e6 * u.K, 1e6 / u.m ** 3, 'p')
<Quantity 1.07221...e-07 1 / s>

>>> fundamental_electron_collision_freq(100 * u.eV, 1e20 / u.m ** 3, 'p')
<Quantity 3935958.7... 1 / s>

>>> fundamental_electron_collision_freq(0.1 * u.eV, 1e6 / u.m ** 3, 'p', coulomb_log_method = 'GMS-1')
<Quantity 3872815.5... 1 / s>

>>> fundamental_electron_collision_freq(0.1 * u.eV, 1e6 / u.m ** 3, 'p', V = c / 100)
<Quantity 5.6589...e-07 1 / s>

>>> fundamental_electron_collision_freq(100 * u.eV, 1e20 / u.m ** 3, 'p', coulomb_log = 20)
<Quantity 5812633... 1 / s>
```

See also:

collision_frequency(), fundamental_ion_collision_freq()

**fundamental_ion_collision_freq**

plasmapy.formulary.collisions.fundamental_ion_collision_freq(T_i: Unit("K"), n_i: Unit("1/m^3"), ion_particle, coulomb_log=None, V=None, coulomb_log_method='classical') -> Unit("1/s")

Average momentum relaxation rate for a slowly flowing Maxwellian distribution of ions.

3 provides a derivation of this as an average collision frequency between ions and ions for a Maxwellian distribution. It is thus a special case of the collision frequency with an averaging factor.

Parameters

- **T_i (Quantity)** – The electron temperature of the Maxwellian test ions
- **n_i (Quantity)** – The number density of the Maxwellian test ions
- **ion_particle (str)** – String signifying a particle type of the test and field ions, including charge state information. This function assumes the test and field ions are the same species.
- **V (Quantity, optional)** – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.
- **coulomb_log (float or dimensionless ~astropy.units.Quantity, optional)** – Option to specify a Coulomb logarithm of the electrons on the ions. If not specified, the Coulomb log will is calculated using the ~plasmapy.formulary.Coulomb_logarithm function.

https://www.nrl.navy.mil/ppd/content/nrl-plasma-formulary

• **coulomb_log_method** *(str, optional)* – Method used for Coulomb logarithm calculation (see that function for more documentation). Choose from “classical” or “GMS-1” to “GMS-6”.

**Notes**

Equations (2.36) and (2.122) in\(^3\) provide the original source used to implement this formula, however, in our implementation we use the very same process that leads to the fundamental electron collision rate (2.17), gaining simply a different coefficient:

\[
\nu_i = \frac{8}{3 \times 4 \times \sqrt{\pi}} \nu(v_{Ti})
\]

Where \(\nu\) is the general collision frequency and \(v_{Ti}\) is the ion thermal velocity (the average, for a Maxwellian distribution).

Note that in the derivation, it is assumed that electrons are present in such numbers as to establish quasineutrality, but the effects of the test ions colliding with them are not considered here. This is a very typical approximation in transport theory.

This result is an ion momentum relaxation rate, and is used in many classical transport expressions. It is equivalent to: * 1/\(\pi_i\) from ref\(^1\), equation (2.5i) pp. 215, * \(\nu_i\) from ref\(^2\) pp. 33,

**References**

**Examples**

```python
gfixture fundamental_ion_collision_freq(0.1 * u.eV, 1e6 / u.m ** 3, 'p')
<Quantity 2.868...e-05 1 / s>
gfixture fundamental_ion_collision_freq(1e6 * u.K, 1e6 / u.m ** 3, 'p')
<Quantity 1.741...e-09 1 / s>
gfixture fundamental_ion_collision_freq(100 * u.eV, 1e20 / u.m ** 3, 'p', coulomb_log_method='GMS-1')
<Quantity 63087.5... 1 / s>
gfixture fundamental_ion_collision_freq(100 * u.eV, 1e20 / u.m ** 3, 'p', V = c / 100)
<Quantity 9.111... 1 / s>
gfixture fundamental_ion_collision_freq(100 * u.eV, 1e20 / u.m ** 3, 'p', coulomb_log=20)
<Quantity 95918.7... 1 / s>
```

See also: `collision_frequency()`, `fundamental_electron_collision_freq()`

---


https://www.nrl.navy.mil/ppd/content/nrl-plasma-formulary
mean_free_path

plasmapy.formulary.collisions.mean_free_path(T: Unit("K"), n_e: Unit("1 / m3"), particles, z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical') -> Unit("m")

Collisional mean free path (m)

Parameters

• T (Quantity) – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle
• n_e (Quantity) – The electron density in units convertible to per cubic meter.
• particles (tuple) – A tuple containing string representations of the test particle (listed first) and the target particle (listed second)
• z_mean (Quantity, optional) – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.
• V (Quantity, optional) – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.
• method (str, optional) – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

Returns mfp – The collisional mean free path for particles in a plasma.

Return type float or numpy.ndarray

Raises

• ValueError – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.
• UnitConversionError – If the units on any of the inputs are incorrect
• TypeError – If the n_e, T, or V are not Quantities.
• RelativityError – If the input velocity is same or greater than the speed of light.

Warns

• ~astropy.units.UnitsWarning – If units are not provided, SI units are assumed
• ~plasmapy.utils.RelativityWarning – If the input velocity is greater than 5% of the speed of light.

Notes

The collisional mean free path is given by\(^1\)

\[
\lambda_{mfp} = \frac{v}{\nu}
\]

where \( v \) is the inter-particle velocity (typically taken to be the thermal velocity) and \( \nu \) is the collision frequency.

\(^1\) Francis, F. Chen. Introduction to plasma physics and controlled fusion 3rd edition. Ch 5 (Springer 2015).
Examples

```python
>>> from astropy import units as u
>>> n = 1e19 * u.m**-3
>>> T = 1e6 * u.K
>>> particles = ('e', 'p')
>>> mean_free_path(T, n, particles)
<Quantity 7.839... m>
>>> mean_free_path(T, n, particles, V=1e6 * u.m / u.s)
<Quantity 0.0109... m>
```

References

**Spitzer_resistivity**

Plasmapy.formulary.collisions.Spitzer_resistivity(T: Unit("K"), n: Unit("1 / m3"), particles, z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical') -> Unit("m Ohm")

Spitzer resistivity of a plasma

**Parameters**

- **T** (*Quantity*) – Temperature in units of temperature. This should be the electron temperature for electron-electron and electron-ion collisions, and the ion temperature for ion-ion collisions.
- **n** (*Quantity*) – The density in units convertible to per cubic meter. This should be the electron density for electron-electron collisions, and the ion density for electron-ion and ion-ion collisions.
- **z_mean** (*Quantity, optional*) – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.
- **particles** (*tuple*) – A tuple containing string representations of the test particle (listed first) and the target particle (listed second)
- **V** (*Quantity, optional*) – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.
- **method** (*str, optional*) – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

**Returns** **spitzer** – The resistivity of the plasma in Ohm meters.

**Return type** float or numpy.ndarray

**Raises**

- **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.
- **UnitConversionError** – If the units on any of the inputs are incorrect
- **TypeError** – If the \( n_e, T, \) or \( V \) are not Quantities.
• `RelativiError` – If the input velocity is same or greater than the speed of light.

**Warns**

• `~astropy.units.UnitsWarning` – If units are not provided, SI units are assumed
• `~plasmapy.utils.RelativityWarning` – If the input velocity is greater than 5% of the speed of light.

**Notes**

The Spitzer resistivity is given by

\[
\eta = \frac{m}{nZ_1Z_2q_e^2} \nu_{1,2}
\]

where \(m\) is the ion mass or the reduced mass, \(n\) is the ion density, \(Z\) is the particle charge state, \(q_e\) is the charge of an electron, \(\nu_{1,2}\) is the collisional frequency between particle species 1 and 2.

Typically, particle species 1 and 2 are selected to be an electron and an ion, since electron-ion collisions are inelastic and therefore produce resistivity in the plasma.

**Examples**

```python
>>> from astropy import units as u
>>> n = 1e19*u.m**-3
>>> T = 1e6*u.K
>>> particles = ('e', 'p')
>>> Spitzer_resistivity(T, n, particles)
<Quantity 2.4915...e-06 m Ohm>
>>> Spitzer_resistivity(T, n, particles, V=1e6 * u.m / u.s)
<Quantity 0.000324... m Ohm>
```

**References**

- mobility

`plasmapy.formulary.collisions.mobility` *(T: Unit("K"), n_e: Unit("1 / m3"), particles, z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical') -> Unit("m2 / (s V)")]*

Electrical mobility (m^2/(V s))

**Parameters**

- `T (Quantity)` – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle
- `n_e (Quantity)` – The electron density in units convertible to per cubic meter.
- `particles (tuple)` – A tuple containing string representations of the test particle (listed first) and the target particle (listed second)

---

- **z_mean** (*Quantity, optional*) – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters. It is also used to obtain the average mobility of a plasma with multiple charge state species. When z_mean is not given, the average charge between the two particles is used instead.

- **V** (*Quantity, optional*) – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.

- **method** (*str, optional*) – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

Returns **mobility_value** – The electrical mobility of particles in a collisional plasma.

Return type float or numpy.ndarray

Raises

- **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.

- **UnitConversionError** – If the units on any of the inputs are incorrect

- **TypeError** – If the n_e, T, or V are not Quantities.

- **RelativityError** – If the input velocity is same or greater than the speed of light.

Warns

- **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed

- **~plasmapy.utils.RelativityWarning** – If the input velocity is greater than 5% of the speed of light.

Notes

The mobility is given by

\[
\mu = \frac{q}{m\nu}
\]

where \( q \) is the particle charge, \( m \) is the particle mass and \( \nu \) is the collisional frequency of the particle in the plasma.

The mobility describes the forced diffusion of a particle in a collisional plasma which is under the influence of an electric field. The mobility is essentially the ratio of drift velocity due to collisions and the electric field driving the forced diffusion.

Examples

```python
>>> from astropy import units as u
>>> n = 1e19 * u.m**-3
>>> T = 1e6 * u.K
>>> particles = ('e', 'p')
>>> mobility(T, n, particles)
<Quantity 250505... m2 / (s V)>
>>> mobility(T, n, particles, V=1e6 * u.m / u.s)
<Quantity 1921.2784... m2 / (s V)>
```

References

Knudsen_number

plasmapy.formulary.collisions.Knudsen_number(characteristic_length, T: Unit("K"), n_e: Unit("1 / m^3"), particles, z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical') -> Unit(dimensionless)

Knudsen number (dimensionless)

Parameters

- **characteristic_length (Quantity)** – Rough order-of-magnitude estimate of the relevant size of the system.
- **T (Quantity)** – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle
- **n_e (Quantity)** – The electron density in units convertible to per cubic meter.
- **particles (tuple)** – A tuple containing string representations of the test particle (listed first) and the target particle (listed second)
- **z_mean (Quantity, optional)** – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.
- **V (Quantity, optional)** – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.
- **method (str, optional)** – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

Returns **knudsen_param** – The dimensionless Knudsen number.

**Return type** float or numpy.ndarray

**Raises**

- **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.
- **UnitConversionError** – If the units on any of the inputs are incorrect
- **TypeError** – If the n_e, T, or V are not Quantities.
- **RelativityError** – If the input velocity is same or greater than the speed of light.

**Warns**

- **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed
- **~plasmapy.utils.RelativityWarning** – If the input velocity is greater than 5% of the speed of light.
Notes

The Knudsen number is given by

$$Kn = \frac{\lambda_{mf}}{L}$$

where $\lambda_{mf}$ is the collisional mean free path for particles in a plasma and `$L$` is the characteristic scale length of interest.

Typically the characteristic scale length is the plasma size or the size of a diagnostic (such as the length or radius of a Langmuir probe tip). The Knudsen number tells us whether collisional effects are important on this scale length.

Examples

```python
>>> from astropy import units as u
>>> L = 1e-3 * u.m
>>> n = 1e19*u.m**-3
>>> T = 1e6*u.K
>>> particles = ('e', 'p')
>>> Knudsen_number(L, T, n, particles)
<Quantity 7839.5...>
>>> Knudsen_number(L, T, n, particles, V=1e6 * u.m / u.s)
<Quantity 10.91773...>
```

References

coupling_parameter

plasmapy.formulary.collisions.coupling_parameter(T: Unit("K"), n_e: Unit("1 / m^3"), particles, z_mean: Unit(dimensionless) = <Quantity nan>, V: Unit("m / s") = <Quantity nan m / s>, method='classical') -> Unit(dimensionless)

Coupling parameter. Coupling parameter compares Coulomb energy to kinetic energy (typically) thermal. Classical plasmas are weakly coupled $\Gamma \ll 1$, whereas dense plasmas tend to have significant to strong coupling $\Gamma \geq 1$.

Parameters

- **T (Quantity)** – Temperature in units of temperature or energy per particle, which is assumed to be equal for both the test particle and the target particle.
- **n_e (Quantity)** – The electron density in units convertible to per cubic meter.
- **particles (tuple)** – A tuple containing string representations of the test particle (listed first) and the target particle (listed second).
- **z_mean (Quantity, optional)** – The average ionization (arithmetic mean) for a plasma where the a macroscopic description is valid. This is used to recover the average ion density (given the average ionization and electron density) for calculating the ion sphere radius for non-classical impact parameters.

---

1 https://en.wikipedia.org/wiki/Knudsen_number
• **V** (*Quantity, optional*) – The relative velocity between particles. If not provided, thermal velocity is assumed: \( \mu V^2 \sim 2k_B T \) where \( \mu \) is the reduced mass.

• **method** (*str, optional*) – Selects which theory to use when calculating the Coulomb logarithm. Defaults to classical method.

Returns **coupling** – The coupling parameter for a plasma.

Return type *float* or *numpy.ndarray*

Raises

• **ValueError** – If the mass or charge of either particle cannot be found, or any of the inputs contain incorrect values.

• **UnitConversionError** – If the units on any of the inputs are incorrect

• **TypeError** – If the \( n_e \), \( T \), or \( V \) are not Quantities.

• **RelativityError** – If the input velocity is same or greater than the speed of light.

Warns

• **~astropy.units.UnitsWarning** – If units are not provided, SI units are assumed

• **~plasmapy.utils.RelativityWarning** – If the input velocity is greater than 5% of the speed of light.

Notes

The coupling parameter is given by

\[
\Gamma = \frac{E_{\text{Coulomb}}}{E_{\text{Kinetic}}}
\]

The Coulomb energy is given by

\[
E_{\text{Coulomb}} = \frac{Z_1 Z_2 q_e^2}{4\pi \varepsilon_0 r}
\]

where \( r \) is the Wigner-Seitz radius, and 1 and 2 refer to particle species 1 and 2 between which we want to determine the coupling.

In the classical case the kinetic energy is simply the thermal energy

\[
E_{\text{kinetic}} = k_B T_e
\]

The quantum case is more complex. The kinetic energy is dominated by the Fermi energy, modulated by a correction factor based on the ideal chemical potential. This is obtained more precisely by taking the the thermal kinetic energy and dividing by the degeneracy parameter, modulated by the Fermi integral

\[
E_{\text{kinetic}} = 2k_B T_e/\chi f_{3/2}(\mu_{\text{ideal}}/k_B T_e)
\]

where \( \chi \) is the degeneracy parameter, \( f_{3/2} \) is the Fermi integral, and \( \mu_{\text{ideal}} \) is the ideal chemical potential.

The degeneracy parameter is given by

\[
\chi = n_e \Lambda_{\text{deBroglie}}^3
\]

where \( n_e \) is the electron density and \( \Lambda_{\text{deBroglie}} \) is the thermal deBroglie wavelength.

See equations 1.2, 1.3 and footnote 5 in\(^2\) for details on the ideal chemical potential.

---


Examples

```python
>>> from astropy import units as u
>>> n = 1e19*u.m**-3
>>> T = 1e6*u.K
>>> particles = ('e', 'p')
>>> coupling_parameter(T, n, particles)
<Quantity 5.8033...e-05>
>>> coupling_parameter(T, n, particles, V=1e6 * u.m / u.s)
<Quantity 5.8033...e-05>
```

References

The subpackage makes heavy use of `astropy.units.Quantity` for handling conversions between different unit systems. This is especially important for electron-volts, commonly used in plasma physics to denote temperature, although it is technically a unit of energy.

Most functions expect `astropy.units.Quantity` as input, however some will use the `validate_quantities` decorator to automatically cast arguments to Quantities with appropriate units. If that happens, you will be notified via an `astropy.units.UnitsWarning`.

Please note that well maintained physical constant data with units and uncertainties can be found in `astropy.constants`.

For a general overview of how unit-based input works, take a look at the following example:

```
Examples:

• Analysing ITER parameters
```

Notes for developers

Values should be returned as an Astropy Quantity in SI units.

If a quantity has several names, then the function name should be the one that provides the most physical insight into what the quantity represents. For example, ‘gyrofrequency’ indicates gyration, while Larmor frequency indicates that this frequency is somehow related to a human (or perhaps a cat?) named Larmor. Similarly, using omega_ce as a function name for this quantity will make the code less readable to people who are unfamiliar with the notation or use a different symbol.

The docstrings for plasma parameter methods should describe the physics associated with these quantities in ways that are understandable to students who are taking their first course in plasma physics while still being useful to experienced plasma physicists.

2.2 Experimental Tools

The `diagnostics` package is in the early stages of development.

2.2.1 Plasma diagnostics (`plasmapy.diagnostics`)

 Defines the Langmuir analysis module as part of the diagnostics package.
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>swept_probe_analysis</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.swept_probe_analysis(probe_characteristic, probe_area: Unit(&quot;m2&quot;), gas_argument, bimaxwellian=False, visualize=False, plot_electron_fit=False, plot_EEDF=False)&lt;br&gt;</code></td>
<td>Attempt to perform a basic swept probe analysis based on the provided characteristic and probe data. Suitable for single cylindrical probes in low-pressure DC plasmas, since OML is applied.</td>
</tr>
<tr>
<td><strong>get_plasma_potential</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_plasma_potential(probe_characteristic[, ...]&lt;br&gt;</code></td>
<td>Implement the simplest but crudest method for obtaining an estimate of the plasma potential from the probe characteristic.</td>
</tr>
<tr>
<td><strong>get_floatiing_potential</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_floatiing_potential(probe_characteristic)&lt;br&gt;</code></td>
<td>Implement the simplest but crudest method for obtaining an estimate of the floating potential from the probe characteristic.</td>
</tr>
<tr>
<td><strong>get_electron_saturation_current</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_electron_saturation_current(...)&lt;br&gt;</code></td>
<td>Obtain an estimate of the electron saturation current corresponding to the obtained plasma potential.</td>
</tr>
<tr>
<td><strong>get_ion_saturation_current</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_ion_saturation_current(probe_characteristic)&lt;br&gt;</code></td>
<td>Implement the simplest but crudest method for obtaining an estimate of the ion saturation current from the probe characteristic.</td>
</tr>
<tr>
<td><strong>get_ion_density_LM</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_ion_density_LM(ion_saturation_current, ...)&lt;br&gt;</code></td>
<td>Implement the Langmuir-Mottley (LM) method of obtaining the ion density.</td>
</tr>
<tr>
<td><strong>get_electron_density_LM</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_electron_density_LM(...)&lt;br&gt;</code></td>
<td>Implement the Langmuir-Mottley (LM) method of obtaining the electron density.</td>
</tr>
<tr>
<td><strong>extract_exponential_section</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.extract_exponential_section(probe_characteristic)&lt;br&gt;</code></td>
<td>Extract the section of exponential electron current growth from the probe characteristic.</td>
</tr>
<tr>
<td><strong>extract_ion_section</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.extract_ion_section(probe_characteristic)&lt;br&gt;</code></td>
<td>Extract the section dominated by ion collection from the probe characteristic.</td>
</tr>
<tr>
<td><strong>get_electron_temperature</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_electron_temperature(exponential_section)&lt;br&gt;</code></td>
<td>Obtain the Maxwellian or bi-Maxwellian electron temperature using the exponential fit method.</td>
</tr>
<tr>
<td><strong>extrapolate_electron_current</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.extrapolate_electron_current(...[, ...])&lt;br&gt;</code></td>
<td>Extrapolate the electron current from the Maxwellian electron temperature obtained in the exponential growth region.</td>
</tr>
<tr>
<td><strong>reduce_bimaxwellian_temperature</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.reduce_bimaxwellian_temperature(T_e, ...)&lt;br&gt;</code></td>
<td>Reduce a bi-Maxwellian (dual) temperature to a single mean temperature for a given fraction.</td>
</tr>
<tr>
<td><strong>get_ion_density_OML</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_ion_density_OML(probe_characteristic,...)&lt;br&gt;</code></td>
<td>Implement the Orbital Motion Limit (OML) method of obtaining an estimate of the ion density.</td>
</tr>
<tr>
<td><strong>extrapolate_ion_current_OML</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.extrapolate_ion_current_OML(...[, visualize])&lt;br&gt;</code></td>
<td>Extrapolate the ion current from the ion density obtained with the OML method.</td>
</tr>
<tr>
<td><strong>get_EEDF</strong>&lt;br&gt;<code>python&lt;br&gt;plasmapy.diagnostics.langmuir.get_EEDF(probe_characteristic[, visualize])&lt;br&gt;</code></td>
<td>Implement the Druyvesteyn method of obtaining the normalized Electron Energy Distribution Function (EEDF).</td>
</tr>
</tbody>
</table>

### swept_probe_analysis

Plasmapy diagnostics.langmuir.swept_probe_analysis<br>

```python<br>plasmapy.diagnostics.langmuir.swept_probe_analysis(probe_characteristic, probe_area: Unit("m2"), gas_argument, bimaxwellian=False, visualize=False, plot_electron_fit=False, plot_EEDF=False)<br>``` Attempt to perform a basic swept probe analysis based on the provided characteristic and probe data. Suitable for single cylindrical probes in low-pressure DC plasmas, since OML is applied.

**Parameters**

- **probe_characteristic** (Characteristic) – The swept probe characteristic that is to be analyzed.
• **probe_area** *(Quantity)* – The area of the probe exposed to plasma in units convertible to m^2.

• **gas_argument** (argument to instantiate the Particle class.) – str, int, or Particle A string representing a particle, element, isotope, or ion; an integer representing the atomic number of an element; or a Particle instance.

• **visualize** *(bool, optional)* – Can be used to plot the characteristic and the obtained parameters. Default is False.

• **plot_electron_fit** *(bool, optional)* – If True, the fit of the electron current in the exponential section is shown. Default is False.

• **plot_EEDF** *(bool, optional)* – If True, the EEDF is computed and shown. Default is False.

**Returns**

• *Results are returned as Dictionary*

• "T_e" *(astropy.units.Quantity)* – Best estimate of the electron temperature in units of eV. Contains two values if bimaxwellian is True.

• "n_e" *(astropy.units.Quantity)* – Estimate of the electron density in units of m^-3. See the Notes on plasma densities.

• "n_i" *(astropy.units.Quantity)* – Estimate of the ion density in units of m^-3. See the Notes on plasma densities.

• "n_i_OML" *(astropy.units.Quantity)* – OML-theory estimate of the ion density in units of m^-3. See the Notes on plasma densities.

• "V_F" *(astropy.units.Quantity)* – Estimate of the floating potential in units of V.

• "V_P" *(astropy.units.Quantity)* – Estimate of the plasma potential in units of V.

• "I_es" *(astropy.units.Quantity)* – Estimate of the electron saturation current in units of Am^-2.

• "I_is" *(astropy.units.Quantity)* – Estimate of the ion saturation current in units of Am^-2.

• "hot_fraction" *(float)* – Estimate of the total hot (energetic) electron fraction.

**Notes**

This function combines the separate probe analysis functions into a single analysis. Results are returned as a Dictionary. On plasma densities: in an ideal quasi-neutral plasma all densities should be equal. However, in practice this will not be the case. The electron density is the poorest estimate due to the hard to obtain knee in the electron current. The density provided by OML theory is likely the best estimate as it is not dependent on the obtained electron temperature, given that the conditions for OML theory hold.

**get_plasma_potential**

`plasmapy.diagnostics.langmuir.get_plasma_potential(probe_characteristic, return_arg=False)`

Implement the simplest but crudest method for obtaining an estimate of the plasma potential from the probe characteristic.

**Parameters**
• **probe_characteristic** *(Characteristic)* – The probe characteristic that is being analyzed.

• **return_arg** *(bool, optional)* – Controls whether or not the argument of the plasma potential within the characteristic array should be returned instead of the value of the voltage. Default is False.

**Returns** $V_P$ – Estimate of the plasma potential in units convertible to V.

**Return type** Quantity

**Notes**

The method used in the function takes the maximum gradient of the probe current as the ‘knee’ of the transition from exponential increase into the electron the saturation region.

### get_floating_potential

```python
plasmapy.diagnostics.langmuir.get_floating_potential(probe_characteristic, return_arg=False)
```

Implement the simplest but crudest method for obtaining an estimate of the floating potential from the probe characteristic.

**Parameters**

• **probe_characteristic** *(Characteristic)* – The probe characteristic that is being analyzed.

• **return_arg** *(bool, optional)* – Controls whether or not the argument of the floating potential within the characteristic array should be returned instead of the value of the voltage. Default is False.

**Returns** $V_F$ – Estimate of the floating potential in units convertible to V.

**Return type** Quantity

**Notes**

The method used in this function takes the probe current closest to zero Amperes as the floating potential.

### get_electron_saturation_current

```python
plasmapy.diagnostics.langmuir.get_electron_saturation_current(probe_characteristic)
```

Obtain an estimate of the electron saturation current corresponding to the obtained plasma potential.

**Parameters** **probe_characteristic** *(Characteristic)* – The probe characteristic that is being analyzed.

**Returns** $I_{es}$ – Estimate of the electron saturation current in units convertible to A.

**Return type** Quantity

**Notes**

The function *get_plasma_potential* is used to obtain an estimate of the plasma potential. The corresponding electron saturation current is returned.
get_ion_saturation_current

plasmapy.diagnostics.langmuir.get_ion_saturation_current(probe_characteristic)

Implement the simplest but crudest method for obtaining an estimate of the ion saturation current from the probe characteristic.

Parameters probe_characteristic(Characteristic) – The probe characteristic that is being analyzed.

Returns I_is – Estimate of the ion saturation current in units convertible to A.

Return type Quantity

Notes

The method implemented in this function assumes the ion saturation current to be the smallest probe current in the characteristic. This assumes the bias range in the ion region is sufficiently negative for the ion current to saturate.

get_ion_density_LM

plasmapy.diagnostics.langmuir.get_ion_density_LM(ion_saturation_current, T_e, probe_area, gas)

Implement the Langmuir-Mottley (LM) method of obtaining the ion density.

Parameters

- ion_saturation_current(Quantity) – The ion saturation current in units convertible to A.
- T_e(Quantity) – The electron temperature in units convertible to eV.
- probe_area(Quantity) – The area of the probe exposed to plasma in units convertible to m^2.
- gas(Quantity) – The (mean) mass of the background gas in atomic mass units.

Returns n_i – Estimate of the ion density in units convertible to m^-3.

Return type Quantity

Notes

The method implemented in this function obtains the ion density from the ion saturation current density assuming that the ion current loss to the probe is equal to the Bohm loss. The acoustic Bohm velocity is obtained from the electron temperature and the ion mass.

The ion saturation current is given by

\[ I_{is} = 0.6eA_p n_i \sqrt{\frac{T_e}{m_i}}. \]
get\_electron\_density\_LM

plasmapy.diagnostics.langmuir.get\_electron\_density\_LM(electron\_saturation\_current:
Unit("A"), \( T_e \): Unit("eV"),
probe\_area: Unit("m^2") -> Unit("1 / m^3")

Implement the Langmuir-Mottley (LM) method of obtaining the electron density.

Parameters

- **electron\_saturation\_current** *(Quantity)* – The electron saturation current in units convertible to A.
- **\( T_e \)** *(Quantity)* – The electron temperature in units convertible to eV.
- **probe\_area** *(Quantity)* – The area of the probe exposed to plasma in units convertible to m^2.

Returns **\( n_e \)** – Estimate of the electron density in units convertible to m^-3.

Return type **Quantity**

Notes

The method implemented in this function obtains the electron density from the electron saturation current density, assuming a low plasma density. The electron saturation current is given by

\[
I_{es} = \frac{1}{4}en_eAp \sqrt{\frac{8T_e}{\pi m_e}}.
\]

Please note that the electron saturation current density is a hard parameter to acquire and it is usually better to measure the ion density, which should be identical to the electron density in quasineutral plasmas.

extract\_exponential\_section

plasmapy.diagnostics.langmuir.extract\_exponential\_section(probe\_characteristic,
\( T_e=\)None,
ion\_current=\)None)

Extract the section of exponential electron current growth from the probe characteristic.

Parameters

- **probe\_characteristic** *(Characteristic)* – The probe characteristic that is being analyzed.
- **\( T_e \)** *(Quantity, optional)* – If given, the electron temperature can improve the accuracy of the bounds of the exponential region.
- **ion\_current** *(Characteristic, optional)* – If given, the ion current will be subtracted from the probe characteristic to yield a better estimate of the electron current in the exponential region.

Returns **exponential\_section** – The exponential electron current growth section.

Return type **Characteristic**
Notes

This function extracts the region of exponential electron growth from the probe characteristic under the assumption that this bias region is bounded by the floating and plasma potentials. Additionally, an improvement in accuracy can be made when the electron temperature is supplied.

**extract_ion_section**

```python
plasmapy.diagnostics.langmuir.extract_ion_section(probe_characteristic)
```

Extract the section dominated by ion collection from the probe characteristic.

**Parameters**

- `probe_characteristic` (`Characteristic`) – The probe characteristic that is being analyzed.

**Returns**

- `ion_section` – The exponential electron current growth section.

**Return type**

`Characteristic`

Notes

This function extracts the region dominated by ion collection from the probe characteristic under the assumption that this bias region is only bounded by the floating potential on the right hand side.

**get_electron_temperature**

```python
plasmapy.diagnostics.langmuir.get_electron_temperature(exponential_section, bimaxwellian=False, visualize=False, return_fit=False, return_hot_fraction=False)
```

Obtain the Maxwellian or bi-Maxwellian electron temperature using the exponential fit method.

**Parameters**

- `exponential_section` (`Characteristic`) – The probe characteristic that is being analyzed.
- `bimaxwellian` (`bool`, optional) – If True the exponential section will be fit assuming bi-Maxwellian electron populations, as opposed to Maxwellian. Default is False.
- `visualize` (`bool`, optional) – If True a plot of the exponential fit is shown. Default is False.
- `return_fit` (`bool`, optional) – If True the parameters of the fit will be returned in addition to the electron temperature. Default is False.
- `return_hot_fraction` (`float`, optional) – If True the total fraction of hot electrons will be returned if the population is bi-Maxwellian. Default is False.

**Returns**

- `T_e` – The estimated electron temperature in eV. In case of a bi-Maxwellian plasma an array containing two Quantities is returned.

**Return type**

`Quantity`, (ndarray)
Notes

In the electron growth region of the probe characteristic the electron current grows exponentially with bias voltage:

\[ I_e = I_e \exp \left( -\frac{e(V_P - V)}{T_e} \right). \]

In log space the current in this region should be a straight line if the plasma electrons are fully Maxwellian, or exhibit a knee in a bi-Maxwellian case. The slope is inversely proportional to the temperature of the respective electron population:

\[ \log(I_e) \propto \frac{1}{T_e}. \]

`extrapolate_electron_current`

`plasmapy.diagnostics.langmuir.extrapolate_electron_current` (probe_characteristic, fit, bi-maxwellian=False, visualize=False)

Extrapolate the electron current from the Maxwellian electron temperature obtained in the exponential growth region.

Parameters

- `probe_characteristic` (Characteristic) – The probe characteristic that is being analyzed.
- `fit` (ndarray) – Polynomial fit coefficients returned by the electron temperature fit.
- `bimaxwellian` (bool, optional) – If True the electron current is extrapolated assuming bi-Maxwellian electron populations, as opposed to Maxwellian. Default is False.
- `visualize` (bool, optional) – If True a plot of the extracted electron current is shown. Default is False.

Returns `electron_current` – The extrapolated electron current characteristic.

Return type `Characteristic`

Notes

Assuming the electron population is fully Maxwellian the pure electron current is extrapolated from the fit of the exponential region for the entire bias range.

`reduce_bimaxwellian_temperature`

`plasmapy.diagnostics.langmuir.reduce_bimaxwellian_temperature` ($T_e$: Unit("eV"), hot_fraction: float) -> Unit("eV")

Reduce a bi-Maxwellian (dual) temperature to a single mean temperature for a given fraction.

Parameters
• **T_e (Quantity, ndarray)** – The bi-Maxwellian temperatures in eV. If a single temperature is provided, this is returned.

• **hot_fraction (float)** – Fraction of hot to total population. If this parameter is None the temperature is assumed to be singular Maxwellian.

**Returns** *T_e* – The reduced (mean) temperature in units of eV.

**Return type** *Quantity*

**Notes**

This function aids methods that take a single electron temperature in situations where the electron population is bi-Maxwellian. The reduced temperature is obtained as the weighted mean:

\[ T_{e,\text{red}} = T_c (1 - f_h) + T_h f_h \]

---

### get_ion_density_OML

`plasmapy.diagnostics.langmuir.get_ion_density_OML(probe_characteristic: plasmapy.diagnostics.langmuir.Characteristic, probe_area: Unit("m2"), gas, visualize=False, return_fit=False)`

Implement the Orbital Motion Limit (OML) method of obtaining an estimate of the ion density.

**Parameters**

• **probe_characteristic (Characteristic)** – The swept probe characteristic that is to be analyzed.

• **probe_area (Quantity)** – The area of the probe exposed to plasma in units convertible to m^2.

• **gas (Quantity)** – The (mean) mass of the background gas in atomic mass units.

• **visualize (bool, optional)** – If True a plot of the OML fit is shown. Default is False.

• **return_fit (bool, optional)** – If True the parameters of the fit will be returned in addition to the ion density. Default is False.

**Returns** *n_i_OML* – Estimated ion density in m^-3.

**Return type** *Quantity*

**Notes**

The method implemented in this function holds for cylindrical probes in a cold ion plasma, i.e. :math:`T_i=0` eV. With OML theory an expression is found for the ion current as function of probe bias independent of the electron temperature [mott-smith.langmuir-1926]:

\[ I_i \xrightarrow{T_i=0} A_p n_i e \sqrt{2} \frac{e(V_F - V)}{m_i} \]
References

extrapolate_ion_current_OML

plasmapy.diagnostics.langmuir.extrapolate_ion_current_OML(probe_characteristic, fit, visualize=False)

Extrapolate the ion current from the ion density obtained with the OML method.

Parameters

- **probe_characteristic** (Characteristic) – The probe characteristic that is being analyzed.
- **fit** (ndarray) – Fit coefficients returned by the OML method.
- **visualize** (bool, optional) – If True a plot of the extracted electron current is shown. Default is False.

Returns **ion_section** – The exponential electron current growth section.

Return type **Characteristic**

Notes

The exponential section of the probe characteristic should be a straight line if the plasma electrons are fully Maxwellian. The slope is then inversely proportional to the electron temperature.

get_EEDF

plasmapy.diagnostics.langmuir.get_EEDF(probe_characteristic, visualize=False)

Implement the Druyvesteyn method of obtaining the normalized Electron Energy Distribution Function (EEDF).

Parameters

- **probe_characteristic** (Characteristic) – The swept probe characteristic that is to be analyzed.
- **visualize** (bool, optional) – If True a plot of the extracted electron current is shown. Default is False.

Returns

- **energy** (astropy.units.Quantity, ndarray) – Array of potentials in V.
- **probability** (float, ndarray) – Array of floats corresponding to the potentials representing the EEDF in normalized probabilities.

Notes

The Druyvesteyn method requires the second derivative of the probe I-V characteristic, which inherently amplifies noise and measurement errors. Therefore it is advisable to smooth the I-V prior to the use of this function.

The Druyvesteyn analysis results in the following equation [druyvesteyn-1930]:

\[
N_e(\epsilon) = \frac{2}{A_p e^2} \sqrt{\frac{2m_e}{\epsilon}} \frac{d^2I}{dV^2}
\]
References

Classes

**Characteristic**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Characteristic</strong>(bias, current)</td>
<td>Class representing a single I-V probe characteristic for convenient experimental data access and computation.</td>
</tr>
</tbody>
</table>

**Characteristic**

```python
class plasmapy.diagnostics.langmuir.Characteristic(bias: Unit("V"), current: Unit("A"))
```

**Bases:** `object`

Class representing a single I-V probe characteristic for convenient experimental data access and computation. Supports units.

**bias**

Array of applied probe biases in units convertible to V.

Type: `astropy.units.Quantity, ndarray`

**current**

Array of applied probe currents in units convertible to A.

Type: `astropy.units.Quantity, ndarray`

**Methods Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get_padded_limit</strong>(padding[, log])</td>
<td>Return the limits of the current range for plotting, taking into account padding. Matplotlib lacks this functionality.</td>
</tr>
<tr>
<td><strong>get_unique_bias</strong>(inplace=False)</td>
<td>Remove any duplicate bias values through averaging.</td>
</tr>
<tr>
<td><strong>plot</strong>(inplace)</td>
<td>Plot the characteristic in matplotlib.</td>
</tr>
<tr>
<td><strong>sort</strong>(inplace)</td>
<td>Sort the characteristic by ascending bias.</td>
</tr>
</tbody>
</table>

**Methods Documentation**

**get_padded_limit**(padding[, log])

Return the limits of the current range for plotting, taking into account padding. Matplotlib lacks this functionality.

Parameters

- **padding**: float – The padding ratio as a float between 0.0 and 1.0.
- **log**: boolean, optional – If True the calculation will be performed on a logarithmic scale. Default is False.

**get_unique_bias**(inplace=False)

Remove any duplicate bias values through averaging.

**plot**(inplace)

Plot the characteristic in matplotlib.

**sort**(inplace)

Sort the characteristic by ascending bias.
2.3 Data Structures and Simulation

2.3.1 PlasmaPy Plasma

Overview

One of the core classes in PlasmaPy is `Plasma`. In order to make it easy to work with different plasma data in PlasmaPy, the `Plasma` object provides a number of methods for commonly existing plasmas in nature.

All Plasma objects are created using the Plasma factory `Plasma`.

A number of plasma data structures are supported by subclassing this base object. See `Plasma Subclasses` to see a list of all of them.

Creating Plasma Objects

Plasma objects are constructed using the special factory class `Plasma`:

```python
>>> x = plasmapy.classes.Plasma(T_e=T_e,
...                            n_e=n_e,
...                            Z=Z,
...                            particle=particle)
```

The result of a call to `Plasma` will be either a `GenericPlasma` object, or a subclass of `GenericPlasma` which deals with a specific type of data, e.g. `PlasmaBlob` or `Plasma3D` (see `Plasma Subclasses` to see a list of all of them).

```python
class plasmapy.classes.plasma_factory.PlasmaFactory (default_widget_type=None, 
                                                      additional_validation_functions=[], 
                                                      registry=None)
```

Plasma factory class. Used to create a variety of Plasma objects. Valid plasma structures are specified by registering them with the factory.
Using Plasma Objects

Once a Plasma object has been created using Plasma it will be a instance or a subclass of the GenericPlasma class. The documentation of GenericPlasma lists the attributes and methods that are available on all Plasma objects.

Plasma Classes

Defined in plasmapy.classes.sources are a set of GenericPlasma subclasses which convert the keyword arguments data to the standard GenericPlasma interface. These subclasses also provide a method, which describes to the Plasma factory which the data match its plasma data structure.

Classes

<table>
<thead>
<tr>
<th>BasePlasma</th>
<th>Registration class for GenericPlasma and declares some abstract methods for data common in different kinds of plasmas.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenericPlasma(**kwargs)</td>
<td>A Generic Plasma class.</td>
</tr>
</tbody>
</table>

BasePlasma

class plasmapy.classes.BasePlasma

Bases: abc.ABC

Registration class for GenericPlasma and declares some abstract methods for data common in different kinds of plasmas.

This class checks for the existence of a method named is_datasource_for when a subclass of GenericPlasma is defined. If it exists it will add that class to the registry.

Attributes Summary

---

<table>
<thead>
<tr>
<th>average_ionization</th>
</tr>
</thead>
<tbody>
<tr>
<td>electron_density</td>
</tr>
<tr>
<td>electron_temperature</td>
</tr>
<tr>
<td>ion_density</td>
</tr>
<tr>
<td>ion_temperature</td>
</tr>
</tbody>
</table>

Attributes Documentation

average_ionization

electron_density

electron_temperature

ion_density

ion_temperature
GenericPlasma

class plasmapy.classes.GenericPlasma(**kwargs):
    Bases: plasmapy.classes.BasePlasma

    A Generic Plasma class. This class contains definitions for abstract methods declared in the BaseClass.

Methods Summary

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>average_ionization()</td>
</tr>
<tr>
<td>electron_density()</td>
</tr>
<tr>
<td>electron_temperature()</td>
</tr>
<tr>
<td>ion_density()</td>
</tr>
<tr>
<td>ion_temperature()</td>
</tr>
</tbody>
</table>

Methods Documentation

average_ionization()

electron_density()

electron_temperature()

ion_density()

ion_temperature()

Class Inheritance Diagram

ABC ----> BasePlasma ----> GenericPlasma

Plasma Subclasses

Plasma Subclasses (plasmapy.classes.sources)

Introduction

The Plasma3D class is a basic structure to contain spatial information about a plasma. To initialize a Plasma3D system, first create an instance of the Plasma3D class and then set the Plasma3D.density, Plasma3D.momentum, Plasma3D.pressure and the Plasma3D.magnetic_field.

This feature is currently under development.
The PlasmaBlob class is a basic structure to contain just plasma parameter information about a plasma with no associated spatial or temporal scales. To initialize a PlasmaBlob system, call it with arguments: electron temperature `PlasmaBlob.T_e`, and electron density `PlasmaBlob.n_e`. You may also optionally define the ionization, `PlasmaBlob.Z`, and relevant plasma particle, `PlasmaBlob.particle`.

This feature is currently under development.

### Reference/API

#### Classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HDF5Reader</strong></td>
<td><code>hdf5, **kwargs</code></td>
</tr>
<tr>
<td><strong>Plasma3D</strong></td>
<td><code>domain_x, domain_y, domain_z</code></td>
</tr>
<tr>
<td><strong>PlasmaBlob</strong></td>
<td><code>T_e, n_e[, Z, particle]</code></td>
</tr>
</tbody>
</table>

**HDF5Reader**

```python
class plasmapy.classes.sources.HDF5Reader(hdf5, **kwargs)
    Bases: plasmapy.classes.GenericPlasma
```

**Attributes Summary**

- `charge_density`
- `electric_current`
- `electric_field`
- `magnetic_field`

**Methods Summary**

- `close()`
- `is_datasource_for(**kwargs)`

**Attributes Documentation**

- `charge_density`
- `electric_current`
- `electric_field`
- `magnetic_field`

**Methods Documentation**

- `close()`
- `classmethod is_datasource_for(**kwargs)```
Plasma3D

class plasmapy.classes.sources.Plasma3D(domain_x, domain_y, domain_z)
Bases: plasmapy.classes.GenericPlasma

Core class for describing and calculating plasma parameters with spatial dimensions.

x
x-coordinates within the plasma domain. Equal to the domain_x input parameter.
Type astropy.units.Quantity

y
y-coordinates within the plasma domain. Equal to the domain_y input parameter.
Type astropy.units.Quantity

z
z-coordinates within the plasma domain. Equal to the domain_z input parameter.
Type astropy.units.Quantity

grid
(3, x, y, z) array containing the values of each coordinate at every point in the domain.
Type astropy.units.Quantity

domain_shape
Shape of the plasma domain.
Type tuple

density
(x, y, z) array of mass density at every point in the domain.
Type astropy.units.Quantity

momentum
(3, x, y, z) array of the momentum vector at every point in the domain.
Type astropy.units.Quantity

pressure
(x, y, z) array of pressure at every point in the domain.
Type astropy.units.Quantity

magnetic_field
(3, x, y, z) array of the magnetic field vector at every point in the domain.
Type astropy.units.Quantity

Parameters

• domain_x (astropy.units.Quantity) – 1D array of x-coordinates for the plasma domain. Must have units convertible to length.

• domain_y (astropy.units.Quantity) – 1D array of y-coordinates for the plasma domain. Must have units convertible to length.

• domain_z (astropy.units.Quantity) – 1D array of z-coordinates for the plasma domain. Must have units convertible to length.
Attributes Summary

- `alfven_speed`
- `electric_field_strength`
- `magnetic_field_strength`
- `velocity`

Methods Summary

- `add_magnetostatic(*mstats)`
- `is_datasource_for(**kwargs)`

Attributes Documentation

- `alfven_speed`
- `electric_field_strength`
- `magnetic_field_strength`
- `velocity`

Methods Documentation

- `add_magnetostatic(*mstats)`
- `classmethod is_datasource_for(**kwargs)`

PlasmaBlob

class plasmapy.classes.sources.PlasmaBlob(T_e, n_e, Z=None, particle='p')
Bases: plasmapy.classes.GenericPlasma

Class for describing and calculating plasma parameters without spatial/temporal description.

Attributes Summary

- `composition`
- `electron_density`
- `electron_temperature`
- `ionization`

Methods Summary

- `coupling()` Ion-ion coupling parameter to determine if quantum/coupling effects are important.
- `is_datasource_for(**kwargs)`

Continued on next page
Table 33 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantum_theta()</td>
<td>Quantum theta parameter, which compares Fermi kinetic energy to thermal kinetic energy to check if quantum effects are important.</td>
</tr>
<tr>
<td>regimes()</td>
<td>Generate a comprehensive description of the plasma regimes based on plasma properties and consequent plasma parameters.</td>
</tr>
</tbody>
</table>

**Attributes Documentation**

- composition
- electron_density
- electron_temperature
- ionization

**Methods Documentation**

- coupling()
  Ion-ion coupling parameter to determine if quantum/coupling effects are important. This compares Coulomb potential energy to thermal kinetic energy.

- classmethod is_datasource_for(**kwargs)

- quantum_theta()
  Quantum theta parameter, which compares Fermi kinetic energy to thermal kinetic energy to check if quantum effects are important.

- regimes()
  Generate a comprehensive description of the plasma regimes based on plasma properties and consequent plasma parameters.

**Class Inheritance Diagram**

```
ABC → BasePlasma → GenericPlasma → Plasma3D → HDF5Reader
```

2.3. Data Structures and Simulation 107
Writing a new Plasma subclass

Any subclass of `GenericPlasma` which defines a method named `is_datasource_for` will automatically be registered with the `Plasma` factory. The `is_datasource_for` method describes the form of the data for which the `GenericPlasma` subclass is valid. For example, it might check the number and types of keyword arguments. This makes it straightforward to define your own `GenericPlasma` subclass for a new data structure or a custom data source like simulated data. These classes only have to be imported for this to work, as demonstrated by the following example.

```python
import plasmapy.classes
import astropy.units as u

class FuturePlasma(plasmapy.classes.GenericPlasma):
    def __init__(self, **kwargs):
        super(FuturePlasma, self).__init__(**kwargs)

        # Specify a classmethod that determines if the input data matches
        # this new subclass
        @classmethod
        def is_datasource_for(cls, **kwargs):
            """
            Determines if any of keyword arguments have a dimensionless value.
            """
            for _, value in kwargs.items():
                try:
                    if value.unit == u.dimensionless_unscaled:
                        return True
                except AttributeError:
                    pass

            return False

This class will now be available through the `Plasma` factory as long as this class has been defined, i.e. imported into the current session.

If you do not want to create a method named `is_datasource_for` you can manually register your class and matching method using the following method.

```python
import plasmapy.classes

plasmapy.classes.Plasma.register(FuturePlasma, FuturePlasma.some_matching_method)
```

### 2.3.2 Particle Tracker (plasmapy.simulation.tracker)

**Introduction**

This module contains the `ParticleTracker` class, which is a simple particle stepper implementing the Boris algorithm.

This module is highly unstable and is expected to change a lot in the future.

**Examples:**

- Particle stepper
## Reference/API

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParticleTracker</td>
<td>Object representing a species of particles: ions, electrons, or simply a group of particles with a particular initial velocity distribution.</td>
</tr>
</tbody>
</table>

```python
class plasmapy.simulation.particletracker.ParticleTracker(plasma, particle_type='p', n_particles=1, scaling=1, dt=<Quantity 'inf s'>, nt=<inf>)
```

### Parameters

- **plasma** *(Plasma)* – plasma from which fields can be pulled
- **type** *(str)* – particle type. See `plasmapy.atomic.atomic` for suitable arguments. The default is a proton.
- **n_particles** *(int)* – number of macroparticles. The default is a single particle.
- **scaling** *(float)* – number of particles represented by each macroparticle. The default is 1, which means a 1:1 correspondence between particles and macroparticles.
- **dt** *(astropy.units.Quantity)* – length of timestep
- **nt** *(int)* – number of timesteps

### Attributes

- **x** *(astropy.units.Quantity)* – Current position and velocity, respectively. Shape (n, 3).
- **v** *(astropy.units.Quantity)* – Current position and velocity, respectively. Shape (n, 3).
- **position_history** *(astropy.units.Quantity)* – History of position and velocity. Shape (nt, n, 3).
- **velocity_history** *(astropy.units.Quantity)* – History of position and velocity. Shape (nt, n, 3).
- **q** *(astropy.units.Quantity)* – Charge and mass of particle.
- **m** *(astropy.units.Quantity)* – Charge and mass of particle.
**PlasmaPy Documentation, Release 0.2.0**

**eff_q**
Type astropy.units.Quantity

**eff_m**
Total charge and mass of macroparticle.
Type astropy.units.Quantity

**kinetic_energy**
calculated from \( v \), as in, current velocity.
Type astropy.units.Quantity

**kinetic_energy_history**
calculated from velocity_history.

**Examples**
See plasmapy/examples/particle-stepper.ipynb.

**Attributes Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>kinetic_energy_history</strong></td>
<td>Calculates the kinetic energy history for each particle.</td>
</tr>
</tbody>
</table>

**Methods Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boris_push</strong> ([init])</td>
<td>Implements the Boris algorithm for moving particles and updating their velocities.</td>
</tr>
<tr>
<td><strong>plot_time_trajectories</strong> ([plot])</td>
<td>Draws position history versus time.</td>
</tr>
<tr>
<td><strong>plot_trajectories</strong> ()</td>
<td>Draws trajectory history.</td>
</tr>
<tr>
<td><strong>run</strong> ()</td>
<td>Runs a simulation instance.</td>
</tr>
<tr>
<td><strong>test_kinetic_energy</strong> ()</td>
<td>Test conservation of kinetic energy.</td>
</tr>
</tbody>
</table>

**Attributes Documentation**

**kinetic_energy_history**
Calculates the kinetic energy history for each particle.

- **Returns** Array of kinetic energies, shape (nt, n).
- **Return type** Quantity

**Methods Documentation**

**boris_push** (init=False)
Implements the Boris algorithm for moving particles and updating their velocities.

- **Parameters**
  - **init** (bool (optional)) – If True, does not change the particle positions and sets dt to -dt/2.
Notes

The Boris algorithm is the standard energy conserving algorithm for particle movement in plasma physics. See\(^1\) for more details.

Conceptually, the algorithm has three phases:

1. Add half the impulse from electric field.
2. Rotate the particle velocity about the direction of the magnetic field.
3. Add the second half of the impulse from the electric field.

This ends up causing the magnetic field action to be properly “centered” in time, and the algorithm conserves energy.

References

\begin{verbatim}
plot_time_trajectories (plot='xyz')
Draws position history versus time.

Parameters plot (str (optional)) -- Enable plotting of position component x, y, z for each of these letters included in plot.

plot_trajectories()
Draws trajectory history.

run()
Runs a simulation instance.

test_kinetic_energy()
Test conservation of kinetic energy.
\end{verbatim}

Class Inheritance Diagram

2.4 Physical Data

2.4.1 Atomic (plasmapy.atomic)

Introduction

The \texttt{atomic} subpackage provides access to information about atoms, ions, isotopes, and other particles.

Particle Class

The `Particle` class provides an object-oriented interface for particle information.

Creating a Particle Instance

The simplest way to create an instance of the `Particle` class is to pass it a `str` representing a particle.

```python
>>> from plasmapy.atomic import Particle
>>> electron = Particle('e-')
```

The `Particle` class accepts a variety of different `str` formats to represent particles. Atomic symbols are case-sensitive, but element names and many aliases are not.

```python
>>> alpha = Particle('alpha')
>>> deuteron = Particle('D+')
>>> iron56 = Particle('Fe-56')
>>> helium = Particle('helium')
>>> muon = Particle('mu-')
>>> antimuon = Particle('antimuon')
>>> hydride = Particle('H-')
```

An `int` may be used as the first positional argument to `Particle` to represent an atomic number. For isotopes and ions, the mass number may be represented with the `mass_number` keyword and the integer charge may be represented with the `Z` keyword.

```python
>>> proton = Particle(1, mass_number=1, Z=1)
```

The most frequently used `Particle` instances may be imported directly from the atomic subpackage.

```python
>>> from plasmapy.atomic import proton, electron
```

The `Particle` instances that may be imported directly are: `proton`, `electron`, `neutron`, `positron`, `deuteron`, `triton`, and `alpha`.

Accessing Particle Properties

The properties of each particle may be accessed using the attributes of the `Particle` instance.

```python
>>> proton.atomic_number
1
>>> electron.integer_charge
-1
>>> triton.mass_number
3
```

Some of these properties are returned as a `Quantity` in SI units.

```python
>>> alpha.charge
<Quantity 3.20435324e-19 C>
>>> deuteron.mass
```

(continues on next page)
Strings representing particles may be accessed using the `particle`, `element`, `isotope`, and `ionic_symbol` attributes.

```python
>>> antimuon.particle
'mu+'
>>> triton.element
'H'
>>> alpha.isotope
'He-4'
>>> deuteron.ionic_symbol
'D 1+
```

## Categories

The `categories` attribute returns a set with the classification categories corresponding to the particle.

```python
>>> sorted(electron.categories)
['charged', 'electron', 'fermion', 'lepton', 'matter', 'stable']
```

Membership of a particle within a category may be checked using the `is_category` method.

```python
>>> alpha.is_category('lepton')
False
>>> electron.is_category('fermion', 'lepton', 'charged')
True
>>> iron56.is_category(['element', 'isotope'])
True
```

The particle must be in all of the categories in the `require` keyword, at least one of the categories in the `any_of` keyword, and none of the categories in the `exclude` in order for it to return `True`.

```python
>>> deuteron.is_category(require={'element', 'isotope', 'ion'})
True
>>> iron56.is_category(any_of=['charged', 'uncharged'])
False
>>> alpha.is_category(exclude='lepton')
True
```

Calling the `is_category` method with no arguments returns a set containing all of the valid categories for any particle. Valid categories include: 'actinide', 'alkali metal', 'alkaline earth metal', 'antibaryon', 'antilepton', 'antimatter', 'antineutrino', 'baryon', 'boson', 'charged', 'electron', 'element', 'fermion', 'halogen', 'ion', 'isotope', 'lanthanide', 'lepton', 'matter', 'metal', 'metalloid', 'neutrino', 'neutron', 'noble gas', 'nonmetal', 'positron', 'post-transition metal', 'proton', 'stable', 'transition metal', 'uncharged', and 'unstable'.

2.4. Physical Data
PlasmaPy Documentation, Release 0.2.0

Conditionals and Equality Properties
Equality between particles may be tested either between two Particle instances, or between a Particle instance
and a str.
>>> Particle('H-1') == Particle('protium 1+')
False
>>> alpha == 'He-4 2+'
True

The is_electron and is_ion attributes provide a quick way to check whether or not a particle is an electron or
ion, respectively.
>>> electron.is_electron
True
>>> hydride.is_electron
False
>>> deuteron.is_ion
True

The element and isotope attributes return None when the particle does not correspond to an element or isotope. Because non-empty strings evaluate to True and None evaluates to False when converted to a bool, these
attributes may be used in conditional statements to test whether or not a particle is in one of these categories.
particles = [Particle('e-'), Particle('Fe-56'), Particle('alpha')]
for particle in particles:
if particle.element:
print(f"{particle} corresponds to element {particle.element}")
if particle.isotope:
print(f"{particle} corresponds to isotope {particle.isotope}")

Returning Antiparticles
The antiparticle of an elementary particle or antiparticle may be found by either using Python’s unary invert operator
(~) or the antiparticle attribute of a Particle instance.
>>> ~electron
Particle("e+")
>>> antimuon.antiparticle
Particle("mu-")

Functions
In addition to the Particle class, the atomic subpackage has a functional interface.
Symbols and Names
Several functions in atomic return string representations of particles,
isotope_symbol, ionic_symbol, and element_name.

114

including atomic_symbol,

Chapter 2. User Documentation


The full symbol of the particle can be found using `particle_symbol`.

```
>>> particle_symbol('electron')
'e-'
```

**Particle Properties**

The `atomic_number` and `mass_number` functions are analogous to the corresponding attributes in the `Particle` class.

```
>>> atomic_number('iron')
26
>>> mass_number('T+')
3
```

Charge information may be found using `integer_charge` and `electric_charge`.

```
>>> integer_charge('H-')
-1
>>> electric_charge('muon antineutrino')
<Quantity 0. C>
```

These functions will raise a `ChargeError` for elements and isotopes that lack explicit charge information.

```
>>> electric_charge('H')
Traceback (most recent call last):
  ... plasmapy.atomic.exceptions.ChargeError: Charge information is required for electric_charge.
```

The standard atomic weight for the terrestrial environment may be accessed using `standard_atomic_weight`.

```
>>> standard_atomic_weight('Pb').to('u')
<Quantity 207.2 u>
```

The mass of a particle may be accessed through the `particle_mass` function.

```
>>> particle_mass('deuteron')
<Quantity 3.34358372e-27 kg>
```
Isotopes

The relative isotopic abundance of each isotope in the terrestrial environment may be found using `isotopic_abundance`.

```python
>>> isotopic_abundance('H-1')
0.999885
>>> isotopic_abundance('D')
0.000115
```

A list of all discovered isotopes in order of increasing mass number can be found with `known_isotopes`.

```python
>>> known_isotopes('H')
['H-1', 'D', 'T', 'H-4', 'H-5', 'H-6', 'H-7']
```

The isotopes of an element with a non-zero isotopic abundance may be found with `common_isotopes`.

```python
>>> common_isotopes('Fe')
['Fe-56', 'Fe-54', 'Fe-57', 'Fe-58']
```

All stable isotopes of an element may be found with `stable_isotopes`.

```python
>>> stable_isotopes('Pb')
['Pb-204', 'Pb-206', 'Pb-207', 'Pb-208']
```

Stability

The `is_stable` function returns `True` for stable particles and `False` for unstable particles.

```python
>>> is_stable('e-')
True
>>> is_stable('T')
False
```

The `half_life` function returns the particle’s half-life as a `Quantity` in units of seconds, if known.

```python
>>> half_life('n')
<Quantity 881.5 s>
```

For stable particles (or particles that have not been discovered to be unstable), `half_life` returns infinity seconds.

```python
>>> half_life('p+')
<Quantity inf s>
```

If the particle’s half-life is not known to sufficient precision, then `half_life` returns a `str` with the estimated value while issuing a `MissingAtomicDataWarning`.

Additional Properties

The `reduced_mass` function is useful in cases of two-body collisions.

```python
>>> reduced_mass('e-', 'p+)
<Quantity 9.10442514e-31 kg>
```

(continues on next page)
>>> reduced_mass('D+', 'T+')
<Quantity 2.00486597e-27 kg>

Nuclear Reactions

Binding Energy

The binding energy of a nuclide may be accessed either as an attribute of a `Particle` instance, or by using the `nuclear_binding_energy` function.

```python
>>> from plasmapy.atomic import Particle, nuclear_binding_energy

>>> D = Particle('deuterium')
>>> D.binding_energy
<Quantity 3.56414847e-13 J>
>>> nuclear_binding_energy('D').to('GeV')
<Quantity 0.00222457 GeV>
```

Nuclear Reactions

The energy released from a nuclear reaction may be found using the `nuclear_reaction_energy` function. The input may be a `str` representing the reaction.

```python
>>> from plasmapy.atomic import nuclear_reaction_energy

>>> nuclear_reaction_energy('Be-8 + alpha --> carbon-12')
<Quantity 1.18025735e-12 J>
```

The reaction may also be inputted using the `reactants` and `products` keywords.

```python
>>> nuclear_reaction_energy(reactants=['D', 'T'], products=['alpha', 'n'])
<Quantity 2.81812097e-12 J>
```

Decorators

Passing `Particle` Instances to Functions and Methods

When calculating plasma parameters, we very frequently need to access the properties of the particles that make up that plasma. The `particle_input` decorator allows functions and methods to easily the properties of a particle.

The `particle_input` decorator takes valid representations of particles given in arguments to functions and passes through the corresponding instance of the `Particle` class. The arguments must be annotated with `Particle` so that the decorator knows to create the `Particle` instance. The function can then access particle properties by using `Particle` attributes. This decorator will raise an `InvalidParticleError` if the input does not correspond to a valid particle.

```python
from plasmapy.atomic import Particle, particle_input

@particle_input
def particle_mass(particle: Particle):
    return particle.mass
```
If only one positional or keyword argument is annotated with `Particle`, then the keywords `mass_numb` and `Z` may be used when the decorated function is called.

```python
@particle_input
def integer_charge(particle: Particle, Z: int = None, mass_numb: int = None) -> int:
    return particle.integer_charge
```

The above example includes optional type hint annotations for `Z` and `mass_numb` and the returned value. The `particle_input` decorator may be used in methods in classes as well:

```python
class ExampleClass:
    @particle_input
    def particle_symbol(self, particle: Particle) -> str:
        return particle.particle
```

On occasion it is necessary for a function to accept only certain categories of particles. The `particle_input` decorator enables several ways to allow this.

If an annotated keyword is named `element`, `isotope`, or `ion`; then `particle_input` will raise an `InvalidElementError`, `InvalidIsotopeError`, or `InvalidIonError` if the particle is not associated with an element, isotope, or ion; respectively.

```python
@particle_input
def capitalized_element_name(element: Particle):
    return element.element_name
```

```python
@particle_input
def number_of_neutrons(isotope: Particle):
    return isotope.mass_number - isotope.atomic_number
```

```python
@particle_input
def number_of_bound_electrons(ion: Particle):
    return ion.atomic_number - ion.integer_charge
```

The keywords `require`, `any_of`, and `exclude` to the decorator allow further customization of the particle categories allowed as inputs. These keywords are used as in `is_category`.

```python
@particle_input(require='charged')
def sign_of_charge(charged_particle: Particle):
    """Require a charged particle."""
    return '+' if charged_particle.integer_charge > 0 else '-'
```

```python
@particle_input(any_of=['charged', 'uncharged'])
def integer_charge(particle: Particle) -> int:
    """Accept only particles with charge information."""
    return particle.integer_charge
```

```python
@particle_input(exclude={'antineutrino', 'neutrino'})
def particle_mass(particle: Particle):
    """Exclude neutrinos/antineutrinos because these particles have weakly constrained masses."
    return particle.mass
```
See Also

- The mendeleev Python package provides access to properties of elements, isotopes, and ions in the periodic table of elements.

Reference/API

Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_number(element)</td>
<td>Return the number of protons in an atom, isotope, or ion.</td>
</tr>
<tr>
<td>atomic_symbol(element)</td>
<td>Return the atomic symbol.</td>
</tr>
<tr>
<td>common_isotopes(argument, ...)</td>
<td>Return a list of isotopes of an element with an isotopic abundances greater than zero, or if no input is provided, a list of all such isotopes for every element.</td>
</tr>
<tr>
<td>electric_charge(particle)</td>
<td>Return the electric charge (in coulombs) of a particle.</td>
</tr>
<tr>
<td>element_name(element)</td>
<td>Return the name of an element.</td>
</tr>
<tr>
<td>half_life(particle, mass_numb)</td>
<td>Return the half-life in seconds for unstable isotopes and particles, and numpy.inf in seconds for stable isotopes and particles.</td>
</tr>
<tr>
<td>integer_charge(particle)</td>
<td>Return the integer charge of a particle.</td>
</tr>
<tr>
<td>ionic_symbol(particle, mass_numb, Z)</td>
<td>Return the ionic symbol of an ion or neutral atom.</td>
</tr>
<tr>
<td>is_stable(particle, mass_numb)</td>
<td>Return True for stable isotopes and particles and False for unstable isotopes.</td>
</tr>
<tr>
<td>isotope_symbol(isotope, mass_numb)</td>
<td>Return the symbol representing an isotope.</td>
</tr>
<tr>
<td>isotopic_abundance(isotope, mass_numb)</td>
<td>Return the isotopic abundances if known, and otherwise zero.</td>
</tr>
<tr>
<td>known_isotopes(argument, ...)</td>
<td>Return a list of all known isotopes of an element, or a list of all known isotopes of every element if no input is provided.</td>
</tr>
<tr>
<td>mass_number(isotope)</td>
<td>Get the mass number (the number of protons and neutrons) of an isotope.</td>
</tr>
<tr>
<td>nuclear_binding_energy(particle, mass_numb)</td>
<td>Return the nuclear binding energy associated with an isotope.</td>
</tr>
<tr>
<td>nuclear_reaction_energy(*args, **kwargs)</td>
<td>Return the released energy from a nuclear reaction.</td>
</tr>
<tr>
<td>particle_input(wrapped_function, require, ...)</td>
<td>Convert arguments to methods and functions to Particle objects.</td>
</tr>
<tr>
<td>particle_mass(particle, *, Z, mass_numb)</td>
<td>Return the mass of a particle.</td>
</tr>
<tr>
<td>particle_symbol(particle, mass_numb, Z)</td>
<td>Return the symbol of a particle.</td>
</tr>
<tr>
<td>reduced_mass(test_particle, target_particle)</td>
<td>Find the reduced mass between two particles.</td>
</tr>
<tr>
<td>stable_isotopes(argument, ...)</td>
<td>Return a list of all stable isotopes of an element, or if no input is provided, a list of all such isotopes for every element.</td>
</tr>
<tr>
<td>standard_atomic_weight(element)</td>
<td>Return the standard (conventional) atomic weight of an element based on the relative abundances of isotopes in terrestrial environments.</td>
</tr>
</tbody>
</table>
atomic_number

```
plasmapy.atomic.atomic_number(  
    element: plasmapy.atomic.particle_class.Particle  
) → numbers.Integral
```

Return the number of protons in an atom, isotope, or ion.

**Parameters**

- `element` *(str or Particle)*: A string representing an element, isotope, or ion; or an instance of the `Particle` class.

**Returns**

- `atomic_number`: The atomic number of an element.

**Return type**

`int`

**Raises**

- `InvalidElementError`: If the argument is a valid particle but not a valid element.
- `InvalidParticleError`: If the argument does not correspond to a valid particle.
- `TypeError`: If the argument is not a `str`.

**See also:**

- `mass_number` [returns the mass number (the total number of protons and neutrons) of an isotope.]

**Examples**

```
>>> atomic_number("H")
1
>>> atomic_number("tritium")
1
>>> atomic_number("alpha")
2
>>> atomic_number("oganesson")
118
```

atomic_symbol

```
plasmapy.atomic.atomic_symbol(  
    element: plasmapy.atomic.particle_class.Particle  
) → str
```

Return the atomic symbol.

**Parameters**

- `element` *(str, int, or Particle)*: A `str` representing an element, isotope, or ion; or an `int` or `str` representing an atomic number.

**Returns**

- `symbol`: The atomic symbol of the element, isotope, or ion.

**Return type**

`str`

**Raises**

- `InvalidElementError`: If the argument is a valid particle but not a valid element.
- `InvalidParticleError`: If the argument does not correspond to a valid particle.
- `TypeError`: If the argument is not a `str` or `int`.

**See also:**

- `element_name()`, `isotope_symbol()`, `ionic_symbol()`, `particle_symbol()`
Notes

This function returns the symbol of the element rather than the symbol of an isotope or ion. For example, 'deuterium', 'T', or 'hydrogen-2' will yield 'H'; 'alpha' will yield 'He'; and 'iron-56' or 'Fe-56' will yield 'Fe'.

This function is case insensitive when there is no potential for ambiguity associated with case. However, this function will return 'H' for hydrogen for lower case 'p' but capital 'P' if the argument is 'P' for phosphorus. This function will return 'N' for nitrogen if the argument is capital 'N', but will not accept lower case 'n' for neutrons.

Examples

```python
>>> atomic_symbol('helium')
'He'
>>> atomic_symbol(42)
'Mo'
>>> atomic_symbol('D')
'H'
>>> atomic_symbol('C-13')
'C'
>>> atomic_symbol('alpha')
'He'
>>> atomic_symbol('79')
'Au'
>>> atomic_symbol('N')  # Nitrogen
'N'
>>> atomic_symbol('P'), atomic_symbol('p')  # Phosphorus, proton
('P', 'H')
```

common_isotopes

```python
plasmapy.atomic.common_isotopes(argument: Union[str, numbers.Integral] = None, most_common_only: bool = False) → List[str]
```

Return a list of isotopes of an element with an isotopic abundances greater than zero, or if no input is provided, a list of all such isotopes for every element.

Parameters

- **argument** (int or str, optional) – A string or integer representing an atomic number or element, or a string representing an isotope.
- **most_common_only** (bool) – If set to True, return only the most common isotope.

Returns isotopes_list – List of all isotopes of an element with isotopic abundances greater than zero, sorted from most abundant to least abundant. If no isotopes have isotopic abundances greater than zero, this function will return an empty list. If no arguments are provided, then a list of all common isotopes of all elements will be provided that is sorted by atomic number, with entries for each element sorted from most abundant to least abundant.

Return type list of str or empty list

Raises

- **InvalidElementError** – If the argument is a valid particle but not a valid element.
- **InvalidParticleError** – If the argument does not correspond to a valid particle.
• **TypeError** – If the argument is not a string or integer.

**Notes**

The isotopic abundances are based on the terrestrial environment and may not be appropriate for space and astrophysical applications.

**See also:**

- **known_isotopes** [returns a list of isotopes that] have been discovered.
- **stable_isotopes** [returns isotopes that are stable] against radioactive decay.
- **isotopic_abundance** [returns the relative isotopic] abundance.

**Examples**

```python
>>> common_isotopes('H')
['H-1', 'D']
>>> common_isotopes(44)
['Ru-102', 'Ru-104', 'Ru-101', 'Ru-99', 'Ru-100', 'Ru-96', 'Ru-98']
>>> common_isotopes('beryllium 2+')
['Be-9']
>>> common_isotopes('Fe')
['Fe-56', 'Fe-54', 'Fe-57', 'Fe-58']
>>> common_isotopes('Fe', most_common_only=True)
['Fe-56']
>>> common_isotopes()[0:7]
['H-1', 'D', 'He-4', 'He-3', 'Li-7', 'Li-6', 'Be-9']
```

**electric_charge**

```python
plasmapy.atomic.electric_charge(particle: plasmapy.atomic.particle_class.Particle) → astropy.units.quantity.Quantity
```

Return the electric charge (in coulombs) of a particle.

**Parameters** **particle** *(str)* – String representing an element or isotope followed by integer charge information.

**Returns** **charge** – The electric charge in coulombs.

**Return type** **Quantity**

**Raises**

- **InvalidParticleError** – If the argument does not correspond to a valid particle or contradictory information is provided.
- **ChargeError** – If charge information for the particle is not available.
- **AtomicWarning** – If the input represents an ion with an integer charge that is below -3.

**Notes**

This function supports two formats for integer charge information.
The first format is a string that has information for the element or isotope at the beginning, a space in between, and the integer charge information in the form of an integer followed by a plus or minus sign, or a plus or minus sign followed by an integer.

The second format is a string containing element information at the beginning, following by one or more plus or minus signs.

This function returns $-1.6021766208 \times 10^{-19}$ C for electrons and $1.6021766208 \times 10^{-19}$ C for positrons.

**Examples**

```python
>>> electric_charge('p+')
<Quantity 1.60217662e-19 C>
>>> electric_charge('H-')
<Quantity -1.60217662e-19 C>
```

**element_name**

plasmapy.atomic.element_name(element: plasmapy.atomic.particle_class.Particle) → str

Return the name of an element.

Parameters **argument** (str, int, or Particle) – A str representing an element, isotope, or ion or an int representing an atomic number

Returns **name** – The name of the element.

Return type **str**

Raises

- InvalidElementError – If the argument is a valid particle but not a valid element.
- InvalidParticleError – If the argument does not correspond to a valid particle.
- TypeError – If the argument is not a str or int.

See also:

atomic_symbol(), isotope_symbol(), ionic_symbol(), particle_symbol()

**Examples**

```python
>>> element_name("H")
'hydrogen'
>>> element_name("T")
'hydrogen'
>>> element_name("alpha")
'helium'
>>> element_name(42)
'molybdenum'
>>> element_name("C-12")
'carbon'
```
**half_life**

```python
plasmapy.atomic.half_life(particle: plasmapy.atomic.particle_class.Particle, mass_numb: Optional[numbers.Integral] = None) → astropy.units.quantity.Quantity
```

Return the half-life in seconds for unstable isotopes and particles, and `numpy.inf` in seconds for stable isotopes and particles.

**Parameters**

- `particle` *(int, str, or Particle)* – A string representing an isotope or particle, an integer representing an atomic number, or an instance of the Particle class.
- `mass_numb` *(int, optional)* – The mass number of an isotope.

**Returns** `half_life_sec` – The half-life of the isotope or particle in units of seconds.

**Return type** `Quantity`

**Raises**

- `InvalidParticleError` – If the argument does not correspond to a valid particle or contradictory information is provided.
- `MissingAtomicDataError` – If no half-life data is available for the isotope.
- `TypeError` – The argument is not an `int` or `str` or the mass number is not an `int`.

**Notes**

Accurate half-life data is not known for all isotopes. Some isotopes may have upper or lower limits on the half-life, in which case this function will return a string with that information and issue a `MissingAtomicDataWarning`. When no isotope information is available, then this function raises a `MissingAtomicDataError`.

**Examples**

```python
>>> half_life('T')
<Quantity 3.888e+08 s>
>>> half_life('n')
<Quantity 881.5 s>
>>> half_life('H-1')
<Quantity inf s>
```

**integer_charge**

```python
plasmapy.atomic.integer_charge(particle: plasmapy.atomic.particle_class.Particle) → numbers.Integral
```

Return the integer charge of a particle.

**Parameters** `particle` *(str)* – String representing a particle.

**Returns** `Z` – The charge as a multiple of the elementary charge.

**Return type** `int`

**Raises**
• InvalidParticleError – If the argument does not correspond to a valid particle or contradictory information is provided.
• ChargeError – If charge information for the particle is not available.
• AtomicWarning – If the input represents an ion with an integer charge that is less than or equal to \(-3\), which is unlikely to occur in nature.

Notes

This function supports two formats for integer charge information.

The first format is a string that has information for the element or isotope at the beginning, a space in between, and the integer charge information in the form of an integer followed by a plus or minus sign, or a plus or minus sign followed by an integer.

The second format is a string containing element information at the beginning, following by one or more plus or minus signs.

Examples

```python
>>> integer_charge('Fe-56 2+')
2
>>> integer_charge('He -2')
-2
>>> integer_charge('H+')
1
>>> integer_charge('N-14++')
2
```

ionic_symbol

plasmapy.atomic.ionic_symbol (particle: plasmapy.atomic.particle_class.Particle, mass_numb: numbers.Integral = None, Z: numbers.Integral = None) \(\rightarrow\) str

Return the ionic symbol of an ion or neutral atom.

Parameters

• particle (int, str, or Particle) – A str representing an element, isotope, or ion; or an int representing an atomic number.
• mass_numb (int or str, optional) – The mass number of the ion or neutral atom.
• Z (int or str, optional) – The integer charge of the ion or neutral atom.

Returns symbol – The ionic symbol. The result will generally be returned as something like 'He-4 2+', 'D 1+', or 'p+'.

Return type str

Raises

• InvalidIonError – If the arguments correspond to a valid particle but not a valid ion or neutral charged particle.
• InvalidParticleError – If arguments do not correspond to a valid particle or contradictory information is provided.
• **TypeError** – If `particle` is not a `str`, `int`, or `Particle`; or if either of `mass_numb` or `Z` is not an `int` or `str` representing an integer.

**Warns** `-plasmapy.utils.AtomicWarning` – If redundant mass number or charge information is provided.

**See also:**

`atomic_symbol()`, `isotope_symbol()`, `particle_symbol()`

**Examples**

```python
>>> ionic_symbol('alpha')
'He-4 2+'
>>> ionic_symbol(79, mass_numb=197, Z=12)
'Au-197 12+'
>>> ionic_symbol('proton')
'p+'
>>> ionic_symbol('D', Z=1)
'D 1+'
>>> ionic_symbol('H-1', Z=0)
'H-1 0+'
```

**is_stable**

`plasmapy.atomic.is_stable` *(particle: plasmapy.atomic.particle_class.Particle, mass_numb: Optional[numbers.Integral] = None) → bool)*

Return `True` for stable isotopes and particles and `False` for unstable isotopes.

**Parameters**

- `particle`*(int, str, or Particle)* – A string representing an isotope or particle, or an integer representing an atomic number.

- `mass_numb`*(int, optional)* – The mass number of the isotope.

**Returns** `is_stable` *(True, optional)* – The isotope is stable. `False` if it is unstable.

**Return type** `bool`

**Raises**

- `InvalidIsotopeError` – If the arguments correspond to a valid element but not a valid isotope.

- `InvalidParticleError` – If the arguments do not correspond to a valid particle.

- **TypeError** – If the argument is not a `str` or `int`.

- `MissingAtomicDataError` – If stability information is not available.

**Examples**

```python
>>> is_stable("H-1")
True
>>> is_stable("tritium")
False
>>> is_stable("e-")
```

(continues on next page)
True
>>> is_stable("tau")
False

### isotope_symbol

**plasmapy.atomic.isotope_symbol** *(isotope: plasmapy.atomic.particle_class.Particle, mass_num: Optional[numbers.Integral] = None) → str*

Return the symbol representing an isotope.

**Parameters**

- **isotope** *(str, int, or Particle)* – A str representing an element, isotope, or ion or an int representing an atomic number.
- **mass_num** *(int or str, optional)* – The mass number of the isotope.

**Returns** symbol – The isotopic symbol. The result will generally be returned as something like 'He-4' or 'Au-197'. This function will return 'D' for deuterium and 'T' for tritium.

**Return type** str

**Raises**

- **InvalidIsotopeError** – If the argument is a valid particle but not a valid isotope.
- **InvalidParticleError** – If the argument does not correspond to a valid particle or contradictory information is provided.
- **TypeError** – If the argument is not a str, int, or Particle.

** Warns** ‘-plasmapy.utils.AtomicWarning’ – If redundant isotope information is provided.

**See also:**

atomic_symbol(), ionic_symbol(), particle_symbol()

#### Examples

```python
>>> isotope_symbol('He', 4)
'He-4'
>>> isotope_symbol(79, 197)
'Au-197'
>>> isotope_symbol('hydrogen-2')
'D'
>>> isotope_symbol('carbon-13')
'C-13'
>>> isotope_symbol('alpha')
'He-4'
```

### isotopic_abundance

**plasmapy.atomic.isotopic_abundance** *(isotope: plasmapy.atomic.particle_class.Particle, mass_num: Optional[numbers.Integral] = None) → numbers.Real*

Return the isotopic abundances if known, and otherwise zero.
Parameters

• argument (str or int) – A string representing an element or isotope, or an integer representing the atomic number of an element.

• mass_numb (int, optional) – The mass number of an isotope, which is required if and only if the first argument can only be used.

Returns iso_comp – The relative isotopic abundance in the terrestrial environment.

Return type float

Raises

• InvalidIsotopeError – If the argument is a valid particle but not a valid isotope.

• InvalidParticleError – If the argument does not correspond to a valid particle or contradictory information is provided.

• TypeError – If the argument is not a str or int.

Notes

Isotopic composition data are most readily available for the terrestrial environment, so this function may not be wholly appropriate for space and astrophysical applications.

The data retrieved from this routine are those recommended by NIST as of 2017.

Examples

```python
>>> isotopic_abundance('Pb-208')
0.524
>>> isotopic_abundance('hydrogen', 1)
0.999885
```

known_isotopes

plasmapy.atomic.known_isotopes (argument: Union[str, numbers.Integral] = None) → List[str]

Return a list of all known isotopes of an element, or a list of all known isotopes of every element if no input is provided.

Parameters argument (int or str, optional) – A string representing an element, isotope, or ion or an integer representing an atomic number

Returns isotopes_list – List of all of the isotopes of an element that have been discovered, sorted from lowest mass number to highest mass number. If no argument is provided, then a list of all known isotopes of every element will be returned that is sorted by atomic number, with entries for each element sorted by mass number.

Return type list containing str items or an empty list

Raises

• InvalidElementError – If the argument is a valid particle but not a valid element.

• InvalidParticleError – If the argument does not correspond to a valid particle.

• TypeError – If the argument is not a str or int.
Notes

This list returns both natural and artificially produced isotopes.

See also:

*common_isotopes* [returns isotopes with non-zero] isotopic abundances.

*stable_isotopes* [returns isotopes that are] stable against radioactive decay.

Examples

```python
given_isotopes('H')
['H-1', 'D', 'T', 'H-4', 'H-5', 'H-6', 'H-7']
given_isotopes('helium 1+')
['He-3', 'He-4', 'He-5', 'He-6', 'He-7', 'He-8', 'He-9', 'He-10']
given_isotopes()[0:10]
['H-1', 'D', 'T', 'H-4', 'H-5', 'H-6', 'H-7', 'He-3', 'He-4', 'He-5']
len(given_isotopes())  # the number of known isotopes
3352
```

mass_number

plasmapy.atomic.mass_number(isotope: plasmapy.atomic.particle_class.Particle) → numbers.Integral

Get the mass number (the number of protons and neutrons) of an isotope.

Parameters:
- **isotope** (str or Particle) – A string representing an isotope or a neutron; or an instance of the \texttt{plasmapy.atomic.Particle} class.

Returns:
- **mass_number** – The total number of protons plus neutrons in a nuclide.

Return type:
- int

Raises:
- \texttt{InvalidParticleError} – If the argument does not correspond to a valid particle.
- \texttt{InvalidIsotopeError} – If the argument does not correspond to a valid isotope.
- \texttt{TypeError} – The argument is not a str.

See also:

*atomic_number* [returns the number of protons in] an isotope or element

Examples

```python
>>> mass_number("H-1")
1
>>> mass_number("Pb-208")
208
>>> mass_number("tritium")
3
>>> mass_number("alpha")
4
```
nuclear_binding_energy

`plasmapy.atomic.nuclear_binding_energy(particle: plasmapy.atomic.particle_class.Particle, mass_numb: Optional[int] = None) → astropy.units.quantity.Quantity`

Return the nuclear binding energy associated with an isotope.

Parameters

- **particle**(str, int, or `Particle`) – A Particle object, a string representing an element or isotope, or an integer representing the atomic number of an element.
- **mass_numb**(int, optional) – The mass number of an isotope, which is required if and only if the first argument can only be used to determine the element and not the isotope.

Returns **binding_energy** – The binding energy of the nucleus in units of joules.

Return type **Quantity**

Raises

- `InvalidParticleError` – If the inputs do not correspond to a valid particle.
- `AtomicError` – If the inputs do not correspond to a valid isotope or nucleon.
- `TypeError` – If the inputs are not of the correct types.

See also:

- `nuclear_reaction_energy` [Return the change in] binding energy during nuclear fusion or fission reactions.
- `mass_energy` [Return the mass energy of a] nucleon or particle.

Examples

```python
>>> from astropy import units as u
>>> nuclear_binding_energy('Fe-56').to(u.MeV)  # Fe-56
<Quantity 492.25957 MeV>
>>> nuclear_binding_energy(26, 56)  # Fe-56
<Quantity 7.8868678e-11 J>
>>> nuclear_binding_energy('p')  # proton
<Quantity 0. J>
>>> from astropy import units as u
>>> before = nuclear_binding_energy("D") + nuclear_binding_energy("T")
>>> after = nuclear_binding_energy("alpha")
>>> (after - before).to(u.MeV)  # released energy from D + T -> alpha + n
<Quantity 17.589 MeV>
```

nuclear_reaction_energy

`plasmapy.atomic.nuclear_reaction_energy(*args, **kwargs)`

Return the released energy from a nuclear reaction.

Parameters

- **reaction**(str (optional, positional argument only)) – A string representing the reaction, like "D + T -> alpha + n" or "Be-8 -> 2 * He-4".
• **reactants** *(list, tuple, or str, optional, keyword-only)* – A list or tuple containing the reactants of a nuclear reaction (e.g., ['D', 'T']), or a string representing the sole reactant.

• **products** *(list, tuple, or str, optional, keyword-only)* – A list or tuple containing the products of a nuclear reaction (e.g., ['alpha', 'n']), or a string representing the sole product.

**Returns** energy – The difference between the mass energy of the reactants and the mass energy of the products in a nuclear reaction. This quantity will be positive if the reaction is exothermic (releases energy) and negative if the reaction is endothermic (absorbs energy).

**Return type** Quantity

**Raises**

• AtomicError: – If the reaction is not valid, there is insufficient information to determine an isotope, the baryon number is not conserved, or the charge is not conserved.

• TypeError: – If the positional input for the reaction is not a string, or reactants and/or products is not of an appropriate type.

**See also:**

*nuclear_binding_energy* [finds the binding energy] of an isotope

**Notes**

This function requires either a string containing the nuclear reaction, or reactants and products as two keyword-only lists containing strings representing the isotopes and other particles participating in the reaction.

**Examples**

```python
>>> from astropy import units as u

>>> nuclear_reaction_energy("D + T --> alpha + n")
<Quantity 2.8181e-12 J>

>>> triple_alphal = '2*He-4 --> Be-8'
>>> triple_alpha2 = 'Be-8 + alpha --> carbon-12'
>>> energy_triplealpha1 = nuclear_reaction_energy(triple_alphal)
>>> energy_triplealpha2 = nuclear_reaction_energy(triple_alpha2)
>>> print(energy_triplealpha1, energy_triplealpha2)
-1.471430e-14 J 1.1802573e-12 J
>>> energy_triplealpha2.to(u.MeV)
<Quantity 7.3665870 MeV>

>>> nuclear_reaction_energy(reactants=['n'], products=['p+', 'e-'])
<Quantity 1.25343e-13 J>
```
**particle_input**

```python
```

Convert arguments to methods and functions to `Particle` objects.

Take positional and keyword arguments that are annotated with `Particle`, and pass through the `Particle` object corresponding to those arguments to the decorated function or method.

Optionally, raise an exception if the particle does not satisfy the specified categorical criteria.

**Parameters**

- `wrapped_function` *(callable)* – The function or method to be decorated.
- `require` *(str, set, list, or tuple, optional)* – Categories that a particle must be in. If a particle is not in all of these categories, then an `AtomicError` will be raised.
- `any_of` *(str, set, list, or tuple, optional)* – Categories that a particle may be in. If a particle is not in any of these categories, then an `AtomicError` will be raised.
- `exclude` *(str, set, list, or tuple, optional)* – Categories that a particle cannot be in. If a particle is in any of these categories, then an `AtomicError` will be raised.
- `none_shall_pass` *(bool, optional)* – If set to `True`, then the decorated argument may be set to `None` without raising an exception. In such cases, this decorator will pass `None` through to the decorated function or method. If set to `False` and the annotated argument is given a value of `None`, then this decorator will raise a `TypeError`.

**Notes**

If the annotated argument is named `element`, `isotope`, or `ion`, then the decorator will raise an `InvalidElementError`, `InvalidIsotopeError`, or `InvalidIonError` if the particle does not correspond to an element, isotope, or ion, respectively.

If exactly one argument is annotated with `Particle`, then the keywords `Z` and `mass_numb` may be used to specify the integer charge and/or mass number of an ion or isotope. However, the decorated function must allow `Z` and/or `mass_numb` as keywords in order to enable this functionality.

**Raises**

- `TypeError` – If the annotated argument is not a `str`, `int`, `tuple`, `list`, or `Particle`; or if `Z` or `mass_numb` is not an `int`.
- `ValueError` – If the number of input elements in a collection do not match the number of expected elements.
- `plasmapy/utils/InvalidParticleError` – If the annotated argument does not correspond to a valid particle.
- `plasmapy/utils/InvalidElementError` – If an annotated argument is named `element`, and the input does not correspond to an element, isotope, or ion.
- `plasmapy/utils/InvalidIsotopeError` – If an annotated argument is named `isotope`, and the input does not correspond to an isotope or an ion of an isotope.
- `plasmapy/utils/InvalidIonError` – If an annotated argument is named `ion`, and the input does not correspond to an ion.
• plasmapy/utils/ChargeError – If 'charged' is in the require argument and the particle is not explicitly charged, or if any_of = {'charged', 'uncharged'} and the particle does not have charge information associated with it.

• plasmapy/utils/AtomicError – If an annotated argument does not meet the criteria set by the categories in the require, any_of, and exclude keywords; if more than one argument is annotated and Z or mass_numb are used as arguments; or if none of the arguments have been annotated with Particle.

Examples

The following simple decorated function returns the Particle object created from the function’s sole argument:

```python
from plasmapy.atomic import particle_input, Particle
@particle_input
def decorated_function(particle: Particle):
    return particle
```

This decorator may also be used to accept arguments using tuple annotation containing specific number of elements or using list annotation which accepts any number of elements in an iterable. Returns a tuple of Particle:

```python
from plasmapy.atomic import particle_input, Particle
@particle_input
def decorated_tuple_function(particles: (Particle, Particle)):
    return particles
sample_particles = decorated_tuple_function(('He', 'Li'))

@particle_input
def decorated_list_function(particles: [Particle]):
    return particles
sample_particles = decorated_list_function(['Al 3+', 'C'])
sample_particles = decorated_list_function(['He', 'Ne', 'Ar'])
```

This decorator may be used for methods in instances of classes, as in the following example:

```python
from plasmapy.atomic import particle_input, Particle
class SampleClass:
    @particle_input
    def decorated_method(self, particle: Particle):
        return particle
sample_instance = SampleClass()
sample_instance.decorated_method('Fe')
```

Some functions may intended to be used with only certain categories of particles. The require, any_of, and exclude keyword arguments enable this functionality.

```python
from plasmapy.atomic import particle_input, Particle
@particle_input(
    require={'matter'},
    any_of={'charged', 'uncharged'},
    exclude={'neutrino', 'antineutrino'},
)
def selective_function(particle: Particle):
    return particle
```
**particle_mass**

```python
plasmapy.atomic.particle_mass(particle: plasmapy.atomic.particle_class.Particle, *, Z: numbers.Integral = None, mass_numb: numbers.Integral = None) → astropy.units.quantity.Quantity
```

Return the mass of a particle.

**Parameters**

- **particle** *(str, int, or Particle)* – A string representing an element, isotope, ion, or special particle; an integer representing an atomic number; or an instance of the Particle class.
- **Z** *(int, optional, keyword-only)* – The ionization state of the ion.
- **mass_numb** *(int, optional, keyword-only)* – The mass number of an isotope.

**Returns** `mass` – The mass of the particle.

**Return type** `Quantity`

**Raises**

- **TypeError** – The argument is not a string, integer, or Quantity.
- **InvalidParticleError** – If the argument does not correspond to a valid particle.
- **MissingAtomicDataError** – If the standard atomic weight, the isotope mass, or the particle mass is not available.

**See also:**

~plasmapy.atomic.standard_atomic_weight

**Notes**

This function will return the ion mass for ions, the isotope mass for isotopes (when available), the standard atomic weight for elements (when available), or the mass of special particles, as appropriate.

The masses of neutrinos are not available because primarily upper limits are presently known.

**particle_symbol**

```python
plasmapy.atomic.particle_symbol(particle: plasmapy.atomic.particle_class.Particle, mass_numb: numbers.Integral = None, Z: numbers.Integral = None) → str
```

Return the symbol of a particle.

**Parameters**

- **particle** *(int, str, or Particle)* – A str representing a particle, element, isotope, or ion or an int representing an atomic number
- **mass_numb** *(int or str, optional)* – The mass number of an isotope.
- **Z** *(int or str, optional)* – The integer charge of an ion.

**Returns** `symbol` – The particle symbol, containing charge and mass number information when available. The result will generally be returned as something like 'e-', 'Fe', 'He-4 2+', 'D', 'n', 'mu-', or 'p'.

**Return type** `str`
Raises

- `InvalidParticleError` – If arguments do not correspond to a valid particle or contradictory information is provided.
- `TypeError` – If ion is not a `str`, `int`, or `Particle`; or if either of `mass_numb` or `Z` is not an `int` or a `str` representing an integer.

Warns `'plasmapy.utils.AtomicWarning'` – If redundant mass number or charge information is provided.

See also:
`atomic_symbol()`, `isotope_symbol()`, `ionic_symbol()`

Examples

```python
>>> particle_symbol('electron')
'e-
>>> particle_symbol('proton')
'p+
>>> particle_symbol('alpha')
'He-4 2+
>>> particle_symbol('H-1', Z=-1)
'H-1 1-
```

**reduced_mass**

`plasmapy.atomic.reduced_mass(test_particle, target_particle) → astropy.units.quantity.Quantity`

Find the reduced mass between two particles.

**Parameters**

- `target_particle(test_particle,)` -
- `or Constant(Quantity,)` – The test particle as represented by a string, an integer representing atomic number, a `Particle` object, or a `Quantity` or `Constant` with units of mass.

**Returns** `reduced_mass` – The reduced mass between the test particle and target particle.

**Return type** `Quantity`

**Raises**

- `InvalidParticleError` – If either particle is invalid.
- `UnitConversionError` – If an argument is a `Quantity` or `Constant` but does not have units of mass.
- `MissingAtomicDataError` – If the mass of either particle is not known.
- `TypeError` – If either argument is not a `str`, `int`, `Particle`, `Quantity`, or `Constant`.

**Example**
```python
>>> from astropy import units as u
>>> reduced_mass('p+', 'e-')
<Quantity 9.104425e-31 kg>
>>> reduced_mass(5.4e-27 * u.kg, 8.6e-27 * u.kg)
<Quantity 3.31714286e-27 kg>
```

**stable_isotopes**

`plasmapy.atomic.stable_isotopes(\text{argument: Union[\text{str, numbers.Integral}] = None, unstable: bool = False}) \rightarrow \text{List[str]}\`  

Return a list of all stable isotopes of an element, or if no input is provided, a list of all such isotopes for every element.

**Parameters**

- **argument** (int or str) – A string or integer representing an atomic number or element, or a string representing an isotope.
- **unstable** (bool) – If set to True, this function will return a list of the unstable isotopes instead of the stable isotopes.

**Returns**  
`StableIsotopes` – List of all stable isotopes of an element, sorted from lowest mass number.  
If an element has no stable isotopes, this function returns an empty list.

**Return type**  
`list` of strings or empty list

**Raises**

- **InvalidElementError** – If the argument is a valid particle but not a valid element.
- **InvalidParticleError** – If the argument does not correspond to a valid particle.
- **TypeError** – If the argument is not a string or integer.

**Notes**

There are 254 isotopes for which no radioactive decay has been observed. It is possible that some isotopes will be discovered to be unstable but with extremely long half-lives. For example, bismuth-209 was recently discovered to have a half-life of about 1.9e19 years. However, such isotopes can be regarded as virtually stable for most applications.

**See also:**

- `known_isotopes` [returns a list of isotopes that] have been discovered  
- `common_isotopes` [returns isotopes with non-zero] isotopic abundances

**Examples**

```python
>>> stable_isotopes('H')
['H-1', 'D']
>>> stable_isotopes(44)
['Ru-96', 'Ru-98', 'Ru-99', 'Ru-100', 'Ru-101', 'Ru-102', 'Ru-104']
>>> stable_isotopes('beryllium')
['Be-9']
>>> stable_isotopes('Pb-209')
```

(continues on next page)
Find unstable isotopes using the `unstable` keyword.

```python
>>> stable_isotopes('U', unstable=True)[:5]  # only first five
['U-217', 'U-218', 'U-219', 'U-220', 'U-221']
```

---

### standard_atomic_weight

`plasmapy.atomic.standard_atomic_weight(element: plasmapy.atomic.particle_class.Particle) -> astropy.units.quantity.Quantity`

Return the standard (conventional) atomic weight of an element based on the relative abundances of isotopes in terrestrial environments.

**Parameters**

- `element` *(str, int, or Particle)* – A string representing an element or an integer representing an atomic number, or an instance of the Particle class.

**Returns**

- `atomic_weight` – The standard atomic weight of an element based on values from NIST.

**Return type**

Quantity

**Raises**

- `InvalidElementError` – If the argument is a valid particle but not a valid element.
- `InvalidParticleError` – If the argument does not correspond to a valid particle.
- `TypeError` – If the argument is not a str or int.

**See also:**

`particle_mass()`

**Notes**

Standard atomic weight data are most readily available for the terrestrial environment, so this function may not be wholly appropriate for space and astrophysical environments.

The relative abundances of different isotopes of an element sometimes vary naturally in different locations within the terrestrial environment. The CIAAW provides ranges for these element, which include H, Li, B, C, N, O, Mg, Si, S, Cl, Br, Tl. This function provides a single value from the CIAWW 2015 standard values when a single value is given, and the lower accuracy conventional value given by Meija et al. (2013, doi:10.1515/pac-2015-0305) for the elements where a range is given.

**Examples**

```python
>>> standard_atomic_weight("H")
<Quantity 1.6738233e-27 kg>
>>> standard_atomic_weight("lead")
<Quantity 3.440636e-25 kg>
```
## Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IonizationState</strong></td>
<td>Representation of the ionization state distribution of a single element or isotope.</td>
</tr>
<tr>
<td><strong>IonizationStates</strong></td>
<td>Describe the ionization state distributions of multiple elements or isotopes.</td>
</tr>
<tr>
<td><strong>Particle</strong></td>
<td>A class for an individual particle or antiparticle.</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td></td>
</tr>
</tbody>
</table>

### IonizationState

class plasmapy.atomic.IonizationState:

```py
class plasmapy.atomic.IonizationState:
    def __init__(self, particle, ionic_fractions=None, *, T_e: Unit('K') = <Quantity nan K>, kappa: numbers.Real = inf, n_elem: Unit('1 / m3') = <Quantity nan 1 / m3>, tol: Union[float, int] = 1e-15):
```

**Bases**: object

Representation of the ionization state distribution of a single element or isotope.

**Parameters**

- **particle** (str, integer, or Particle) – A str or Particle instance representing an element or isotope, or an integer representing the atomic number of an element.

- **ionic_fractions** (ndarray, list, tuple, or Quantity; optional) – The ionization fractions of an element, where the indices correspond to integer charge. This argument should contain the atomic number plus one items, and must sum to one within an absolute tolerance of tol if dimensionless. Alternatively, this argument may be a Quantity that represents the number densities of each neutral/ion.

- **T_e** (Quantity, keyword-only, optional) – The electron temperature or thermal energy per particle.

- **n_elem** (Quantity, keyword-only, optional) – The number density of the element, including neutrals and all ions.

- **tol** (float or integer, keyword-only, optional) – The absolute tolerance used by isclose when testing normalizations and making comparisons. Defaults to 1e-15.

**Raises**

- AtomicError – If the ionic fractions are not normalized or contain invalid values, or if number density information is provided through both ionic_fractions and n_elem.

- InvalidParticleError – If the particle is invalid.

### Examples

```py
>>> states = IonizationState('H', [0.6, 0.4], n_elem=1*u.cm**-3, T_e=11000*u.K)
>>> states.ionic_fractions[0]  # fraction of hydrogen that is neutral
0.6
>>> states.ionic_fractions[1]  # fraction of hydrogen that is ionized
0.4
>>> states.n_e  # electron number density
```

(continues on next page)
<Quantity 400000. 1 / m³>

```
>>> states.n_elem  # element number density
<Quantity 1000000. 1 / m³>
```

Notes

Calculation of collisional ionization equilibrium has not yet been implemented.

Attributes Summary

- **T_e**: Return the electron temperature.
- **Z_mean**: Return the mean integer charge.
- **Z_most_abundant**: Return a list of the integer charges with the highest ionic fractions.
- **Z_rms**: Return the root mean square integer charge.
- **atomic_number**: Return the atomic number of the element.
- **base_particle**: Return the symbol of the element or isotope.
- **element**: Return the atomic symbol of the element.
- **equil_ionic_fractions**: Return the equilibrium ionic fractions for temperature T_e or the temperature set in the IonizationState instance.
- **integer_charges**: Return an array with the integer charges.
- **ionic_fractions**: Return the ionic fractions, where the index corresponds to the integer charge.
- **ionic_symbols**: Return the ionic symbols for all charge states.
- **isotope**: Return the isotope symbol for an isotope, or None if the particle is not an isotope.
- **kappa**: Return the kappa parameter for a kappa distribution function for electrons.
- **n_e**: Return the electron number density assuming a single species plasma.
- **n_elem**: Return the total number density of neutrals and all ions.
- **number_densities**: Return the number densities for each state.
- **tol**: Return the absolute tolerance for comparisons.

Methods Summary

- **equilibrate(T_e)**: Set the ionic fractions to collisional ionization equilibrium for temperature T_e.
- **info(minimum_ionic_fraction)**: Print quicklook information for an IonizationState instance.
- **normalize()**: Normalize the ionization state distribution (if set) so that the sum becomes equal to one.
Attributes Documentation

**T_e**
Return the electron temperature.

**Z_mean**
Return the mean integer charge

**Z_most_abundant**
Return a list of the integer charges with the highest ionic fractions.

Examples

```python
>>> He = IonizationState('He', [0.2, 0.5, 0.3])
>>> He.Z_most_abundant
[1]
>>> Li = IonizationState('Li', [0.4, 0.4, 0.2, 0.0])
>>> Li.Z_most_abundant
[0, 1]
```

**Z_rms**
Return the root mean square integer charge.

**atomic_number**
Return the atomic number of the element.

**base_particle**
Return the symbol of the element or isotope.

**element**
Return the atomic symbol of the element.

**equil_ionic_fractions**
Return the equilibrium ionic fractions for temperature T_e or the temperature set in the IonizationState instance. Not implemented.

**integer_charges**
Return an array with the integer charges.

**ionic_fractions**
Return the ionic fractions, where the index corresponds to the integer charge.

Examples

```python
>>> hydrogen_states = IonizationState('H', [0.9, 0.1])
>>> hydrogen_states.ionic_fractions
array([0.9, 0.1])
```

**ionic_symbols**
Return the ionic symbols for all charge states.

**isotope**
Return the isotope symbol for an isotope, or None if the particle is not an isotope.

**kappa**
Return the kappa parameter for a kappa distribution function for electrons.
The value of \( \kappa \) must be greater than 1.5 in order to have a valid distribution function. If \( \kappa \) equals {\textit{inf}}\%, then the distribution function reduces to a Maxwellian.

**n\_e**
- Return the electron number density assuming a single species plasma.

**n\_elem**
- Return the total number density of neutrals and all ions.

**number\_densities**
- Return the number densities for each state.

**tol**
- Return the absolute tolerance for comparisons.

### Methods Documentation

**equilibrate** \((T\_e: \text{Unit}("K") = <\text{Quantity} \text{nan} K>)\)
- Set the ionic fractions to collisional ionization equilibrium for temperature \( T\_e \). Not implemented.

**info** \((\text{minimum\_ionic\_fraction: numbers.Real} = 0.01) \rightarrow \text{None}\)
- Print quicklook information for an `IonizationState` instance.

**Parameters**
- **minimum\_ionic\_fraction\( (\text{Real}) \)\): If the ionic fraction for a particular ionization state is below this level, then information for it will not be printed. Defaults to 0.01.

### Example

```python
>>> He_states = IonizationState(
...     'He',
...     [0.941, 0.058, 0.001],
...     T\_e = 5.34 * u.K,
...     kappa = 4.05,
...     n\_elem = 5.51e19 * u.m ** -3,
... )
>>> He_states.info()
IonizationState instance for He with Z\_mean = 0.06
------------------------------
He 0+: 0.941  n\_i = 5.18e+19 m**-3
He 1+: 0.058  n\_i = 3.20e+18 m**-3
------------------------------

n\_elem = 5.51e19 m**-3
n\_e = 3.31e18 m**-3
T\_e = 5.34e+00 K
kappa = 4.05
------------------------------
```

**normalize** \( \rightarrow \text{None} \)
- Normalize the ionization state distribution (if set) so that the sum becomes equal to one.
IonizationStates

class plasmapy.atomic.IonizationStates(inputs: Union[Dict[str, numpy.ndarray], List[T], Tuple], *, T_e: Unit("K") = <Quantity nan K>, equilibrate: Optional[bool] = None, abundances: Optional[Dict[str, numbers.Real]] = None, log_abundances: Optional[Dict[str, numbers.Real]] = None, n: Unit("1 / m³") = <Quantity nan 1 / m³>, tol: numbers.Real = 1e-15, kappa: numbers.Real = inf)

Bases: object

Describe the ionization state distributions of multiple elements or isotopes.

Parameters

- **inputs** (list, tuple, or dict) – A list or tuple of elements or isotopes (if T_e is provided); a list of IonizationState instances; a dict with elements or isotopes as keys and a ndarray of ionic fractions as the values; or a dict with elements or isotopes as keys and Quantity instances with units of number density.

- **abundances** (dict or str, optional, keyword-only) – The relative abundances of each element in the plasma.

- **log_abundances** (dict, optional, keyword-only) – The base 10 logarithm of the relative abundances of each element in the plasma.

- **n** (Quantity, optional, keyword-only) – The number density scaling factor. The number density of an element will be the product of its abundance and n.

- **T_e** (Quantity, optional, keyword-only) – The electron temperature in units of temperature or thermal energy per base particle.

- **kappa** (float, optional, keyword-only) – The value of kappa for a kappa distribution function.

- **tol** (float or integer, keyword-only, optional) – The absolute tolerance used by isclose when testing normalizations and making comparisons. Defaults to 1e-15.

- **equilibrate** (bool, optional, keyword-only) – Set the ionic fractions to the estimated collisional ionization equilibrium. Not implemented.

Raises AtomicError – If IonizationStates cannot be instantiated.

Examples

```python
>>> from astropy import units as u
>>> from plasmapy.atomic import IonizationStates
>>> states = IonizationStates(
...     {'H': [0.5, 0.5], 'He': [0.95, 0.05, 0]},
...     T_e = 1.2e4 * u.K,
...     n = 1e15 * u.m ** -3,
...     abundances = {'H': 1, 'He': 0.08},
... )
>>> states.ionic_fractions
({'H': array([0.5, 0.5]), 'He': array([0.95, 0.05, 0.])})
```

The number densities are given by the ionic fractions multiplied by the abundance and the
To change the ionic fractions for a single element, use item assignment.

```
>>> states = IonizationStates(['H', 'He'])
>>> states['H'] = [0.1, 0.9]
```

Item assignment will also work if you supply number densities.

```
>>> states['He'] = [0.4, 0.6, 0.0] * u.m ** -3
>>> states.ionic_fractions['He']
array([0.4, 0.6, 0.])
>>> states.number_densities['He']
<Quantity [0.4, 0.6, 0. ] 1 / m3>
```

**Notes**

No more than one of `abundances`, `log_abundances`, and `number_densities` may be specified.

If the value provided during item assignment is a `Quantity` with units of number density that retains the total element density, then the ionic fractions will be set proportionately.

When making comparisons between `IonizationStates` instances, `nan` values are treated as equal. Equality tests are performed to within a tolerance of `tol`.

Collisional ionization equilibrium is based on atomic data that has relative errors of order 20%.

**Attributes Summary**

- `T_e` Return the electron temperature.
- `abundances` Return the elemental abundances.
- `base_particles` Return a list of the elements and isotopes whose ionization states are being kept track of.
- `ionic_fractions` Return a `dict` containing the ionic fractions for each element and isotope.
- `kappa` Return the kappa parameter for a kappa distribution function for electrons.
- `log_abundances` Return a `dict` with atomic or isotope symbols as keys and the base 10 logarithms of the relative abundances as the corresponding values.
- `n` Return the number density scaling factor.
- `n_e` Return the electron number density under the assumption of quasineutrality.
- `number_densities` Return a `dict` containing the number densities for element or isotope.
- `tol` Return the absolute tolerance for comparisons.

**Methods Summary**
equilibrate\((T_e, particles, \text{kappa})\)

Set the ionic fractions to collisional ionization equilibrium.

info\((\text{minimum} \text{ionic fraction})\)

Print quicklook information for an IonizationStates instance.

normalize\()\)

Normalize the ionic fractions so that the sum for each element equals one.

Attributes Documentation

\text{T_e}

Return the electron temperature.

\text{abundances}

Return the elemental abundances.

\text{base_particles}

Return a list of the elements and isotopes whose ionization states are being kept track of.

\text{ionic_fractions}

Return a \text{dict} containing the ionic fractions for each element and isotope.

The keys of this \text{dict} are the symbols for each element or isotope. The values will be \text{ndarray} objects containing the ionic fractions for each ionization level corresponding to each element or isotope.

\text{kappa}

Return the kappa parameter for a kappa distribution function for electrons.

The value of \text{kappa} must be greater than 1.5 in order to have a valid distribution function. If \text{kappa} equals \text{inf}, then the distribution function reduces to a Maxwellian.

\text{log_abundances}

Return a \text{dict} with atomic or isotope symbols as keys and the base 10 logarithms of the relative abundances as the corresponding values.

\text{n}

Return the number density scaling factor.

\text{n_e}

Return the electron number density under the assumption of quasineutrality.

\text{number_densities}

Return a \text{dict} containing the number densities for element or isotope.

\text{tol}

Return the absolute tolerance for comparisons.

Methods Documentation

equilibration\((T_e: \text{Unit} ("K") = \text{<Quantity nan K>}, \text{particles: str = ‘all’, kappa: numbers.Real = inf})\)

Set the ionic fractions to collisional ionization equilibrium. Not implemented.

The electron temperature used to calculate the new equilibrium ionic fractions will be the argument \text{T_e} to this method if given, and otherwise the attribute \text{T_e} if no electron temperature is provided to this method.

Parameters

• \text{T_e (Quantity, optional)} – The electron temperature.
• **particles** *(list, tuple, or str, optional)* – The elements and isotopes to be equi-
librated. If particles is 'all' (default), then all elements and isotopes will be equi-
librated.

• **kappa** *(Real)* – The value of kappa for a kappa distribution for electrons.

```python
info (minimum_ionic_fraction: numbers.Real = 0.01) → None
Print quicklook information for an IonizationStates instance.

Parameters  
minimum_ionic_fraction (Real) – If the ionic fraction for a particular ion-
ization state is below this level, then information for it will not be printed. Defaults to 0.01.
```

**Examples**

```python
>>> states = IonizationStates({  
...     'H': [0.1, 0.9], 'He': [0.95, 0.05, 0.0],  
...     T_e = 12000 * u.K,  
...     n = 3e9 * u.cm ** -3,  
...     abundances = {'H': 1.0, 'He': 0.1},  
...     kappa = 3.4,  
... })
>>> states.info()
IonizationStates instance for: H, He
-----------------------------------------
H 0+: 0.100  n_i = 3.00e+14 m**-3
H 1+: 0.900  n_i = 2.70e+15 m**-3
-----------------------------------------
He 0+: 0.950  n_i = 2.85e+14 m**-3
He 1+: 0.050  n_i = 1.50e+13 m**-3
-----------------------------------------
n_e = 2.71e+15 m**-3
T_e = 1.20e+04 K
kappa = 3.40
-----------------------------------------
```

**normalize** () → None
Normalizes the ionic fractions so that the sum for each element equals one.

**Particle**

```python
Particle (argument: Union[str, numbers.Integral], mass_numb: numbers.Integral = None, Z: numbers.Integral = None)
```

**Bases:** object

A class for an individual particle or antiparticle.

**Parameters**

- **argument** *(str, int, or Particle)* – A string representing a particle, element, isotope, or ion; an integer representing the atomic number of an element; or a Particle instance.

- **mass_numb** *(int, optional)* – The mass number of an isotope or nuclide.

- **Z** *(int, optional)* – The integer charge of the particle.

**Raises**

- **TypeError** – For when any of the arguments or keywords is not of the required type.
InvalidParticleError – Raised when the particle input does not correspond to a valid particle or is contradictory.

InvalidElementError – For when an attribute is being accessed that requires information about an element, but the particle is not an element, isotope, or ion.

InvalidIsotopeError – For when an attribute is being accessed that requires information about an isotope or nuclide, but the particle is not an isotope (or an ion of an isotope).

ChargeError – For when either the charge or integer_charge attributes is being accessed but the charge information for the particle is not available.

AtomicError – Raised for attempts at converting a Particle object to a bool.

Examples

Particles may be defined using a wide variety of aliases:

```python
>>> proton = Particle('p+')
>>> electron = Particle('e-')
>>> neutron = Particle('neutron')
>>> deuteron = Particle('D', Z=1)
>>> alpha = Particle('He', mass_num=4, Z=2)
>>> positron = Particle('positron')
>>> hydrogen = Particle(1)  # atomic number
```

The `particle` attribute returns the particle’s symbol in the standard form.

```python
>>> positron.particle
'e+'
```

The `element, isotope, and ionic_symbol` attributes return the symbols for each of these different types of particles.

```python
>>> proton.element
'H'
>>> alpha.isotope
'He-4'
>>> deuteron.ionic_symbol
'D 1+'
```

The `ionic_symbol` attribute works for neutral atoms if charge information is available.

```python
>>> deuterium = Particle("D", Z=0)
>>> deuterium.ionic_symbol
'D 0+'
```

If the particle doesn’t belong to one of those categories, then these attributes return `None`.

```python
>>> positron.element is None
True
```

The attributes of a `Particle` instance may be used to test whether or not a particle is an element, isotope, or ion.

```python
>>> True if positron.element else False
False
```
Many of the attributes return physical properties of a particle.

```python
>>> electron.integer_charge
-1
>>> proton.spin
0.5
>>> alpha.atomic_number
2
>>> deuteron.mass_number
2
>>> deuteron.binding_energy.to('MeV')
<Quantity 2.224... MeV>
>>> alpha.charge
<Quantity 3.20435...e-19 C>
>>> neutron.half_life
<Quantity 881.5 s>
>>> Particle('C-14').half_life.to(u.year)
<Quantity 5730. yr>
>>> deuteron.electron_number
0
>>> alpha.neutron_number
2
```

If a `Particle` instance represents an elementary particle, then the unary ~ (invert) operator may be used to return the particle's antiparticle.

```python
>>> ~electron
Particle("e+")
>>> ~proton
Particle("p-")
>>> ~positron
Particle("e-")
```

A `Particle` instance may be used as the first argument to `Particle`.

```python
>>> iron = Particle('Fe')
>>> iron == Particle(iron)
True
>>> Particle(iron, mass_numb=56, Z=6)
Particle("Fe-56 6+")
```

If the previously constructed `Particle` instance represents an element, then the `Z` and `mass_numb` arguments may be used to specify an ion or isotope.

```python
>>> iron = Particle('Fe')
>>> Particle(iron, Z=1)
Particle("Fe 1+")
>>> Particle(iron, mass_numb=56)
Particle("Fe-56")
```

The `categories` attribute and `is_category` method may be used to find and test particle membership in categories.

**Attributes Summary**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>antiparticle</td>
<td>Return the corresponding antiparticle, or raise an <code>AtomicError</code> if the particle is not an elementary particle.</td>
</tr>
<tr>
<td>atomic_number</td>
<td>Return the number of protons in an element, isotope, or ion.</td>
</tr>
<tr>
<td>baryon_number</td>
<td>Return the number of baryons in a particle.</td>
</tr>
<tr>
<td>binding_energy</td>
<td>Return the nuclear binding energy in joules.</td>
</tr>
<tr>
<td>categories</td>
<td>Return the particle’s categories.</td>
</tr>
<tr>
<td>charge</td>
<td>Return the particle’s electron charge in coulombs.</td>
</tr>
<tr>
<td>electron_number</td>
<td>Return the number of electrons in an ion.</td>
</tr>
<tr>
<td>element</td>
<td>Return the atomic symbol if the particle corresponds to an element, and <code>None</code> otherwise.</td>
</tr>
<tr>
<td>element_name</td>
<td>Return the name of the element corresponding to this particle, or raise an <code>InvalidElementError</code> if the particle does not correspond to an element.</td>
</tr>
<tr>
<td>half_life</td>
<td>Return the particle’s half-life in seconds, or a <code>str</code> with half-life information.</td>
</tr>
<tr>
<td>integer_charge</td>
<td>Return the particle’s integer charge.</td>
</tr>
<tr>
<td>ionic_symbol</td>
<td>Return the ionic symbol if the particle corresponds to an ion or neutral atom, and <code>None</code> otherwise.</td>
</tr>
<tr>
<td>is_electron</td>
<td>Return <code>True</code> if the particle is an electron, and <code>False</code> otherwise.</td>
</tr>
<tr>
<td>is_ion</td>
<td>Return <code>True</code> if the particle is an ion, and <code>False</code> otherwise.</td>
</tr>
<tr>
<td>isotope</td>
<td>Return the isotope symbol if the particle corresponds to an isotope, and <code>None</code> otherwise.</td>
</tr>
<tr>
<td>isotope_name</td>
<td>Return the name of the element along with the isotope symbol if the particle corresponds to an isotope, and <code>None</code> otherwise.</td>
</tr>
<tr>
<td>isotopic_abundance</td>
<td>Return the isotopic abundance of an isotope.</td>
</tr>
<tr>
<td>lepton_number</td>
<td>Return 1 for leptons, −1 for antileptons, and 0 otherwise.</td>
</tr>
<tr>
<td>mass</td>
<td>Return the mass of the particle in kilograms.</td>
</tr>
<tr>
<td>mass_energy</td>
<td>Return the mass energy of the particle in joules.</td>
</tr>
<tr>
<td>mass_number</td>
<td>Return the number of nucleons in an isotope.</td>
</tr>
<tr>
<td>neutron_number</td>
<td>Return the number of neutrons in an isotope or nucleon.</td>
</tr>
<tr>
<td>nuclide_mass</td>
<td>Return the mass of the bare nucleus of an isotope or a neutron.</td>
</tr>
<tr>
<td>particle</td>
<td>Return the particle’s symbol.</td>
</tr>
<tr>
<td>periodic_table</td>
<td>Return a <code>namedtuple</code> to access category, period, group, and block information about an element.</td>
</tr>
</tbody>
</table>
Table 43 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roman_symbol</td>
<td>Return the spectral name of the particle (i.e. spin)</td>
</tr>
<tr>
<td>spin</td>
<td>Return the spin of the particle.</td>
</tr>
<tr>
<td>standard_atomic_weight</td>
<td>Return an element’s standard atomic weight in kg.</td>
</tr>
</tbody>
</table>

### Methods Summary

- **ionize(n, inplace)**
  - Create a new `Particle` instance corresponding to the current `Particle` after being ionized n times.

- **is_category(*category_tuple, require, ...)**
  - Determine if the particle meets categorization criteria.

- **recombine(n[, inplace])**
  - Create a new `Particle` instance corresponding to the current `Particle` after undergoing recombination n times.

### Attributes Documentation

- **antiparticle**
  - Return the corresponding antiparticle, or raise an `AtomicError` if the particle is not an elementary particle.

  This attribute may be accessed by using the unary operator ~ acting on a `Particle` instance.

  **Examples**

  ```python
  >>> electron = Particle('e-')
  >>> electron.antiparticle
  Particle("e+")
  
  >>> antineutron = Particle('antineutron')
  >>> ~antineutron
  Particle("n")
  ```

- **atomic_number**
  - Return the number of protons in an element, isotope, or ion.

  If the particle is not an element, then this attribute will raise an `InvalidElementError`.

  **Examples**

  ```python
  >>> proton = Particle('p+')
  >>> proton.atomic_number
  1
  >>> curium = Particle('Cm')
  >>> curium.atomic_number
  96
  ```

- **baryon_number**
  - Return the number of baryons in a particle.

  This attribute will return the number of protons and neutrons minus the number of antiprotons and antineutrons. The baryon number is equivalent to the mass number for isotopes.

  **Examples**

  ```python
  ```
If the baryon number is unavailable, then this attribute will raise a `Missing Atomic Data Error`.

**Examples**

```python
>>> alpha = Particle('alpha')
>>> alpha.baryon_number
4
```

**binding_energy**

Return the nuclear binding energy in joules.

This attribute will raise an `Invalid Isotope Error` if the particle is not a nucleon or isotope.

**Examples**

```python
>>> alpha = Particle('alpha')
>>> alpha.binding_energy
<Quantity 4.53346...e-12 J>
>>> Particle('T').binding_energy.to('MeV')
<Quantity 8.481... MeV>
```

The binding energy of a nucleon equals 0 joules.

```python
>>> neutron = Particle('n')
>>> proton = Particle('p+')
>>> neutron.binding_energy
<Quantity 0. J>
>>> proton.binding_energy
<Quantity 0. J>
```

**categories**

Return the particle’s categories.

**Examples**

```python
>>> gold = Particle('Au')
>>> 'transition metal' in gold.categories
True
>>> 'antilepton' in gold.categories
False
```

**charge**

Return the particle’s electron charge in coulombs.

This attribute will raise a `Charge Error` if the charge has not been specified.

**Examples**

```python
>>> electron = Particle('e-')
>>> electron.charge
<Quantity -1.60217662e-19 C>
```
**electron_number**
Return the number of electrons in an ion.
This attribute will return the number of bound electrons in an ion, or 1 for an electron.
If this particle is not an ion or electron, then this attribute will raise an InvalidIonError.

**Examples**

```python
>>> Particle('Li 0+').electron_number
3
>>> Particle('e-').electron_number
1
```

**element**
Return the atomic symbol if the particle corresponds to an element, and None otherwise.

**Examples**

```python
>>> alpha = Particle('alpha')
>>> alpha.element
'He'
```

**element_name**
Return the name of the element corresponding to this particle, or raise an InvalidElementError if the particle does not correspond to an element.

**Examples**

```python
>>> tritium = Particle('T')
>>> tritium.element_name
'hydrogen'
```

**half_life**
Return the particle’s half-life in seconds, or a str with half-life information.
Particles that do not have sufficiently well-constrained half-lives will return a str containing the information that is available about the half-life and issue a MissingAtomicDataWarning.

**Examples**

```python
>>> neutron = Particle('n')
>>> neutron.half_life
<Quantity 881.5 s>
```

**integer_charge**
Return the particle’s integer charge.
This attribute will raise a ChargeError if the charge has not been specified.
Examples

```python
>>> muon = Particle('mu-')
>>> muon.integer_charge
-1
```

**ionic_symbol**

Return the ionic symbol if the particle corresponds to an ion or neutral atom, and `None` otherwise.

Examples

```python
>>> deuteron = Particle('deuteron')
>>> deuteron.ionic_symbol
'D 1+'
>>> hydrogen_atom = Particle('H', Z=0)
>>> hydrogen_atom.ionic_symbol
'H 0+'
```

**is_electron**

Return `True` if the particle is an electron, and `False` otherwise.

Examples

```python
>>> Particle('e-').is_electron
True
>>> Particle('e+').is_electron
False
```

**is_ion**

Return `True` if the particle is an ion, and `False` otherwise.

Examples

```python
>>> Particle('D+').is_ion
True
>>> Particle('H-1 0+').is_ion
False
>>> Particle('e+').is_ion
False
```

**isotope**

Return the isotope symbol if the particle corresponds to an isotope, and `None` otherwise.

Examples

```python
>>> alpha = Particle('alpha')
>>> alpha.isotope
'He-4'
```

**isotope_name**

Return the name of the element along with the isotope symbol if the particle corresponds to an isotope, and `None` otherwise.
If the particle is not a valid element, then this attribute will raise an `InvalidElementError`. If it is not an isotope, then this attribute will raise an `InvalidIsotopeError`.

**Examples**

```python
>>> deuterium = Particle("D")
>>> deuterium.isotope_name
'deuterium'
>>> iron_isotope = Particle("Fe-56", Z=16)
>>> iron_isotope.isotope_name
'iron-56'
```

**isotopic_abundance**

Return the isotopic abundance of an isotope.

If the isotopic abundance is not available, this attribute will raise a `MissingAtomicDataError`. If the particle is not an isotope or is an ion of an isotope, then this attribute will raise an `InvalidIsotopeError`.

**Examples**

```python
>>> D = Particle('deuterium')
>>> D.isotopic_abundance
0.000115
```

**lepton_number**

Return 1 for leptons, −1 for antileptons, and 0 otherwise.

This attribute returns the number of leptons minus the number of antileptons, excluding bound electrons in an atom or ion.

If the lepton number is unavailable, then this attribute will raise a `MissingAtomicDataError`.

**Examples**

```python
>>> Particle('e-').lepton_number
1
>>> Particle('mu+').lepton_number
-1
>>> Particle('He-4 0+').lepton_number
0
```

**mass**

Return the mass of the particle in kilograms.

If the particle is an element and not an isotope or ion, then this attribute will return the standard atomic weight, if available.

If the particle is an isotope but not an ion, then this attribute will return the isotopic mass, including bound electrons.

If the particle is an ion, then this attribute will return the mass of the element or isotope (as just described) minus the product of the integer charge and the electron mass.

For special particles, this attribute will return the standard value for the particle’s mass.
If the mass is unavailable (e.g., for neutrinos or elements with no standard atomic weight), then this attribute will raise a `MissingAtomicDataError`.

**Examples**

```python
>>> Particle('He').mass
<Quantity 6.64647...e-27 kg>
>>> Particle('He+').mass
<Quantity 6.64556...e-27 kg>
>>> Particle('He-4 +1').mass
<Quantity 6.64556...e-27 kg>
>>> Particle('alpha').mass
<Quantity 6.64465...e-27 kg>
```

**mass_energy**

Return the mass energy of the particle in joules.

If the particle is an isotope or nuclide, return the mass energy of the nucleus only.

If the mass of the particle is not known, then raise a `MissingAtomicDataError`.

**Examples**

```python
>>> proton = Particle('p+')
>>> proton.mass_energy
<Quantity 1.503277...e-10 J>

>>> protium = Particle('H-1 0+')
>>> protium.mass_energy
<Quantity 1.503277...e-10 J>

>>> electron = Particle('electron')
>>> electron.mass_energy.to('MeV')
<Quantity 0.510998... MeV>
```

**mass_number**

Return the number of nucleons in an isotope.

This attribute will return the number of protons plus the number of neutrons in an isotope or nuclide.

If the particle is not an isotope, then this attribute will raise an `InvalidIsotopeError`.

**Examples**

```python
>>> alpha = Particle('helium-4 2+')
>>> alpha.mass_number
4
```

**neutron_number**

Return the number of neutrons in an isotope or nucleon.

This attribute will return the number of neutrons in an isotope, or 1 for a neutron.

If this particle is not an isotope or neutron, then this attribute will raise an `InvalidIsotopeError`.
Examples

```python
>>> alpha = Particle('He-4++
>>> alpha.neutron_number
2
>>> Particle('n').neutron_number
1
```

**nuclide_mass**

Return the mass of the bare nucleus of an isotope or a neutron.

This attribute will raise a `InvalidIsotopeError` if the particle is not an isotope or neutron, or a `MissingAtomicDataError` if the isotope mass is not available.

Examples

```python
>>> deuterium = Particle('D')
>>> deuterium.nuclide_mass
<Quantity 3.34358372e-27 kg>
```

**particle**

Return the particle’s symbol.

Examples

```python
>>> electron = Particle('electron')
>>> electron.particle
'e-
```

**periodic_table**

Return a `namedtuple` to access category, period, group, and block information about an element.

If the particle is not an element, isotope, or ion, then this attribute will raise an `InvalidElementError`.

Examples

```python
>>> gold = Particle('Au')
>>> gold.periodic_table.category
'transition metal'
>>> gold.periodic_table.period
6
>>> gold.periodic_table.group
11
>>> gold.periodic_table.block
'd'
```

**roman_symbol**

Return the spectral name of the particle (i.e. the ionic symbol in Roman numeral notation). If the particle is not an ion or neutral atom, return `None`. The roman numeral represents one plus the integer charge. Raise `ChargeError` if no charge has been specified and `OutOfRangeError` if the charge is negative.
Examples

```python
>>> proton = Particle('proton')
>>> proton.roman_symbol
'H-1 II'
>>> hydrogen_atom = Particle('H', Z=0)
>>> hydrogen_atom.roman_symbol
'H I'
```

**spin**

Return the spin of the particle.

If the spin is unavailable, then a `MissingAtomicDataError` will be raised.

**Examples**

```python
>>> positron = Particle('e+')
>>> positron.spin
0.5
```

**standard_atomic_weight**

Return an element's standard atomic weight in kg.

If the particle is isotope or ion or not an element, this attribute will raise an `InvalidElementError`.

If the element does not have a defined standard atomic weight, this attribute will raise a `MissingAtomicDataError`.

**Examples**

```python
>>> oxygen = Particle('O')
>>> oxygen.standard_atomic_weight
<Quantity 2.656696...e-26 kg>
```

Methods Documentation

**ionize** *(n: numbers.Integral = 1, inplace: bool = False)*

Create a new `Particle` instance corresponding to the current `Particle` after being ionized n times.

If `inplace` is `False` (default), then return the ionized `Particle`.

If `inplace` is `True`, then replace the current `Particle` with the newly ionized `Particle`.

**Parameters**

- **n** *(positive integer)* - The number of bound electrons to remove from the `Particle` object. Defaults to 1.
- **inplace** *(bool, optional)* - If `True`, then replace the current `Particle` instance with the newly ionized `Particle`.

**Returns** `particle` – A new `Particle` object that has been ionized n times relative to the original `Particle`. If `inplace` is `False`, instead return `None`.

**Return type** `Particle`

**Raises**
• **InvalidElementError** – If the `Particle` is not an element.
• **ChargeError** – If no charge information for the `Particle` object is specified.
• **InvalidIonError** – If there are less than \( n \) remaining bound electrons.
• **ValueError** – If \( n \) is not positive.

### Examples

```python
given_particle = Particle("Fe 6+").ionize()
given_particle  # Particle("Fe 7+")
```

```python
helium_particle = Particle("He-4 0+")
helium_particle.ionize(n=2, inplace=True)
helium_particle  # Particle("He-4 2+")
```

### `is_category` (**category_tuple**, **require**: `Union[str, Set[T], Tuple[L[T]]] = {}`, **any_of**: `Union[str, Set[T], Tuple[L[T]]] = {}`, **exclude**: `Union[str, Set[T], Tuple[L[T]]] = {}`) ➔ `bool`

Determine if the particle meets categorization criteria. Return `True` if the particle is in all of the inputted categories, and `False` the particle is not.

Required categories may be entered as positional arguments, including as a `list`, `set`, or `tuple` of required categories. These may also be included using the `require` keyword argument. This method will return `False` if the particle is not in all of the required categories.

If categories are inputted using the `any_of` keyword argument, then this method will return `False` if the particle is not of any of the categories in `any_of`.

If the `exclude` keyword is set, then this method will return `False` if the particle is in any of the excluded categories, whether or not the particle matches the other criteria.

### Examples

```python
Particle('e-').is_category('lepton')  # True
Particle('p+').is_category('baryon', exclude='charged')  # False
Particle('n').is_category({'matter', 'baryon'}, exclude={'charged'})  # True
Particle('mu+').is_category('antilepton', exclude='baryon')  # True
```

### `recombine` (**n**: `numbers.Integral = 1`, **inplace**: `False`)

Create a new `Particle` instance corresponding to the current `Particle` after undergoing recombination \( n \) times.

If `inplace` is `False` (default), then return the `Particle` that just underwent recombination.

If `inplace` is `True`, then replace the current `Particle` with the `Particle` that just underwent recombination.

#### Parameters

- **n** (*positive integer*) – The number of electrons to recombine into the `Particle` object.
• *inplace*(bool, optional) – If True, then replace the current *Particle* instance with the *Particle* that just underwent recombination.

**Returns** particle – A new *Particle* object that has undergone recombination n times relative to the original *Particle*. If inplace is False, instead return None.

**Return type** *Particle*

**Raises**

• *InvalidElementError* – If the *Particle* is not an element.
• *ChargeError* – If no charge information for the *Particle* object is specified.
• *ValueError* – If n is not positive.

**Examples**

```python
>>> Particle("Fe 6+").recombine()
Particle("Fe 5+")
>>> helium_particle = Particle("He-4 2+")
>>> helium_particle.recombine(n=2, inplace=True)
>>> helium_particle
Particle("He-4 0+")
```

**State**

```python
class plasmapy.atomic.State(integer_charge, ionic_fraction, ionic_symbol, number_density)
Bases: tuple
```

**Class Inheritance Diagram**

```
State
|-----------------
|                  |
|                  |
|-----------------
| Particle
|                  |
|                  |
|-----------------
| IonizationStates
|                  |
|                  |
|-----------------
| IonizationState
```
2.5 Utilities

2.5.1 Core package utilities (**plasmapy.utils**)

Introduction

**Plasmapy.utils** is where we store functionality that helps us write (what we try to think of as) clean, readable and informative code. This means:

- the many kinds of warnings and exceptions you may (hopefully not!) encounter while working with Plasmapy, such as `plasmapy.utils.RelativityWarning` or `plasmapy.utils.PhysicsError`.
- decorators we use for reusable physical quantity computation and checking, such as `plasmapy.utils.decorators.validate_quantities` and `plasmapy.utils.check_relativistic`.
- Some helper utilities for importing and testing packages such as `plasmapy.utils.call_string`.

Reference/API

**plasmapy.utils.decorators Package**

A module to contain various decorators used to build readable and useful code.

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>angular_freq_to_hz(fn)</code></td>
<td>A decorator that adds to a function the ability to convert the function’s return from angular frequency (rad/s) to frequency (Hz).</td>
</tr>
<tr>
<td><code>check_relativistic([func, betafrac])</code></td>
<td>Warns or raises an exception when the output of the decorated function is greater than <code>betafrac</code> times the speed of light.</td>
</tr>
<tr>
<td><code>check_values([func])</code></td>
<td>A decorator to ‘check’ – limit/control – the values of input and return arguments to a function or method.</td>
</tr>
<tr>
<td><code>check_units([func])</code></td>
<td>A decorator to ‘check’ – limit/control – the units of input and return arguments to a function or method.</td>
</tr>
<tr>
<td><code>preserve_signature(f)</code></td>
<td>A decorator for decorators, which preserves the signature of the function being wrapped.</td>
</tr>
<tr>
<td><code>validate_quantity([func, ...])</code></td>
<td>A decorator to ‘validate’ – control and convert – the units and values of input and return arguments to a function or method.</td>
</tr>
</tbody>
</table>

**angular_freq_to_hz**

`plasmapy.utils.decorators.angular_freq_to_hz(fn)`

A decorator that adds to a function the ability to convert the function’s return from angular frequency (rad/s) to frequency (Hz).

A kwarg `to_hz` is added to the function’s signature, with a default value of `False`. The keyword is also added to the function’s docstring under the “Other Parameters” heading.

**Parameters**

- `fn (function)` – The function to be decorated
PlasmaPy Documentation, Release 0.2.0

Raises `ValueError` – If `fn` has already defined a kwarg `to_hz`

Returns The decorated function

Return type `callable`

Notes

- If `angular_freq_to_hz` is used with decorator `validate_quantities()`, then `angular_freq_to_hz` should be used inside `validate_quantities()` but special consideration is needed for setup. The following is an example of an appropriate setup:

```python
import astropy.units as u
from plasmapy.utils.decorators.converter import angular_freq_to_hz
from plasmapy.utils.decorators.validators import validate_quantities

@validate_quantities(validations_on_return={'units': [u.rad / u.s, u.Hz]})
@angular_freq_to_hz
def foo(x: u.rad / u.s) -> u.rad / u.s:
    return x
```

Adding `u.Hz` to the allowed units allows the converted quantity to pass the validations.

Examples

```python
>>> import astropy.units as u
>>> from plasmapy.utils.decorators.converter import angular_freq_to_hz

>>> @angular_freq_to_hz
... def foo(x):
...     return x

>>> foo(5 * u.rad / u.s, to_hz=True)
<Quantity 0.79577472 Hz>

>>> foo(-1 * u.rad / u.s, to_hz=True)
<Quantity -0.15915494 Hz>
```

Decoration also works with methods

```python
>>> class Foo:
...     def __init__(self, x):
...         self.x = x
...     @angular_freq_to_hz
...     def bar(self):
...         return self.x

>>> foo = Foo(0.5 * u.rad / u.s)
>>> foo.bar(to_hz=True)
<Quantity 0.07957747 Hz>
```
check_relativistic

plasmapy.utils.decorators.check_relativistic(func=None, betafrac=0.05)

Warns or raises an exception when the output of the decorated function is greater than \( betafrac \) times the speed of light.

Parameters

- **func** (function, optional) – The function to decorate.
- **betafrac** (float, optional) – The minimum fraction of the speed of light that will raise a RelativityWarning. Defaults to 5%.

Returns

Decorated function.

Return type

function

Raises

- TypeError – If \( V \) is not a Quantity.
- UnitConversionError – If \( V \) is not in units of velocity.
- ValueError – If \( V \) contains any nan values.
- RelativityError – If \( V \) is greater than or equal to the speed of light.

Warns –plasmapy.utils.RelativityWarning – If \( V \) is greater than or equal to \( betafrac \) times the speed of light, but less than the speed of light.

Examples

```python
>>> from astropy import units as u
>>> @check_relativistic
... def speed():
...     return 1 * u.m / u.s
```

Passing in a custom betafrac:

```python
>>> @check_relativistic(betafrac=0.01)
... def speed():
...     return 1 * u.m / u.s
```

cHECK VALUES

plasmapy.utils.decorators.check_values(func=None, checks_on_return: Dict[str, bool] = None, **checks)

A decorator to ‘check’ – limit/control – the values of input and return arguments to a function or method.

Parameters

- **func** – The function to be decorated
- **checks_on_return** (Dict[str, bool]) – Specifications for value checks on the return of the function being wrapped. (see check values for valid specifications)
- ****checks (Dict[str, Dict[str, bool]]) – Specifications for value checks on the input arguments of the function being wrapped. Each keyword argument in checks is the name of a function argument to be checked and the keyword value contains the value check specifications.

2.5. Utilities
The value check specifications are defined within a dictionary containing the keys defined below. If the dictionary is empty or omitting keys, then the default value will be assumed for the missing keys.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>can_be_negative</td>
<td>bool</td>
<td>[DEFAULT True] values can be negative</td>
</tr>
<tr>
<td>can_be_complex</td>
<td>bool</td>
<td>[DEFAULT False] values can be complex numbers</td>
</tr>
<tr>
<td>can_be_inf</td>
<td>bool</td>
<td>[DEFAULT True] values can be inf</td>
</tr>
<tr>
<td>can_be_nan</td>
<td>bool</td>
<td>[DEFAULT True] values can be nan</td>
</tr>
<tr>
<td>none_shall_pass</td>
<td>bool</td>
<td>[DEFAULT False] values can be a python None</td>
</tr>
</tbody>
</table>

Notes

- Checking of function arguments `*args` and `**kwargs` is not supported.
- Full functionality is defined by the class `CheckValues`.

Examples

```python
from plasmapy.utils.decorators import check_values

@check_values(arg1={'can_be_negative': False, 'can_be_nan': False},
              arg2={'can_be_inf': False},
              checks_on_return={'none_shall_pass': True})
def foo(arg1, arg2):
    return None

# on a method
class Foo:
    @check_values(arg1={'can_be_negative': False, 'can_be_nan': False},
                  arg2={'can_be_inf': False},
                  checks_on_return={'none_shall_pass': True})
    def bar(self, arg1, arg2):
        return None
```

`check_units`

`plasmapy.utils.decorators.check_units(func=None, checks_on_return: Dict[str, Any] = None, **checks)`

A decorator to ‘check’ – limit/control – the units of input and return arguments to a function or method.

Parameters

- `func` – The function to be decorated
- `checks_on_return` (list of astropy `units` or dict of unit specifications) – Specifications for unit checks on the return of the function being wrapped. (see `check_units` for valid specifications)
- `**checks` (list of astropy `units` or dict of unit specifications) – Specifications for unit checks on the input arguments of the function being wrapped. Each keyword argument in `checks` is the name of a function argument to be checked and the keyword value contains the unit check specifications.
Unit checks can be defined by passing one of the astropy `units`, a list of astropy units, or a dictionary containing the keys defined below. Units can also be defined with function annotations, but must be consistent with decorator `**checks` arguments if used concurrently. If a key is omitted, then the default value will be assumed.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units</td>
<td>list of desired astropy units</td>
<td></td>
</tr>
<tr>
<td>equivalencies</td>
<td>[DEFAULT None] A list of equivalent pairs to try if the units are not directly convertible. (see equivalencies, and/or astropy equivalencies)</td>
<td></td>
</tr>
<tr>
<td>pass_equivalent_units</td>
<td>bool</td>
<td>[DEFAULT False] allow equivalent units to pass</td>
</tr>
</tbody>
</table>

**Notes**

- Checking of function arguments `*args` and `**kwargs` is not supported.
- Decorator does NOT perform any unit conversions, look to `validate_quantities()` if that functionality is desired.
- If it is desired that `None` values do not raise errors or warnings, then include `None` in the list of units or as a default value for the function argument.
- If units are not specified in `checks`, then the decorator will attempt to identify desired units by examining the function annotations.
- Full functionality is defined by the class `CheckUnits`.

**Examples**

Define units with decorator parameters:

```python
import astropy.units as u
from plasmapy.utils.decorators import check_units

@check_units(arg1=('units': u.cm),
             arg2=u.cm,
             checks_on_return=[u.cm, u.km])
def foo(arg1, arg2):
    return arg1 + arg2
```

(continues on next page)
Define units with function annotations:

```python
import astropy.units as u
from plasmapy.utils.decorators import check_units

@check_units
def foo(arg1: u.cm, arg2: u.cm) -> u.cm:
    return arg1 + arg2
```

Allow `None` values to pass:

```python
import astropy.units as u
from plasmapy.utils.decorators import check_units

@check_units(chcks_on_return=[u.cm, u.km])
def bar(self, arg1, arg2):
    return arg1 + arg2
```

Allow return values to have equivalent units:

```python
import astropy.units as u
from plasmapy.utils.decorators import check_units

@check_units
def foo(arg1:
    return arg1
```

Allow equivalent units to pass with specified equivalencies:

```python
import astropy.units as u
from plasmapy.utils.decorators import check_units

@check_units
def foo(arg):
    return arg1
```

`preserve_signature`

`plasmapy.utils.decorators.preserve_signature(f)`

A decorator for decorators, which preserves the signature of the function being wrapped. This preservation
allows IDE function parameter hints to work on the wrapped function. To do this, the __signature__ dunder is defined, or inherited, from the function being wrapped to the resulting wrapped function.

**Parameters**

- `f (callable)` – The function being wrapped.

**Returns**

Wrapped version of the function.

**Return type**

callable

### Examples

```python
>>> def a_decorator(f):
...     @preserve_signature
...     @functools.wraps(f)
...     def wrapper(*args, **kwargs):
...         return wrapper(*args, **kwargs)
...     return wrapper
```

#### validate_quantities

Plasmapy.utils.decorators.validate_quantities(func=None, validations_on_return=None, **validations)

A decorator to ‘validate’ – control and convert – the units and values of input and return arguments to a function or method. Arguments are expected to be astropy Quantity objects.

**Parameters**

- `func` – The function to be decorated
- `validations_on_return` (dictionary of validation specifications) – Specifications for unit and value validations on the return of the function being wrapped. (see quantity validations for valid specifications.
- `**validations` (dictionary of validation specifications) – Specifications for unit and value validations on the input arguments of the function being wrapped. Each keyword argument in validations is the name of a function argument to be validated and the keyword value contains the unit and value validation specifications.

Unit and value validations can be defined by passing one of the astropy units, a list of astropy units, or a dictionary containing the keys defined below. Units can also be defined with function annotations, but must be consistent with decorator **validations arguments if used concurrently. If a key is omitted, then the default value will be assumed.

2.5. Utilities 165
<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units</td>
<td>list of desired astropy units</td>
<td></td>
</tr>
<tr>
<td>equivalencies</td>
<td>[DEFAULT None] A list of equivalent pairs to try if the units are not directly convertible. (see equivalencies, and/or astropy equivalencies)</td>
<td></td>
</tr>
<tr>
<td>pass_equivalent_units</td>
<td>bool</td>
<td>[DEFAULT False] allow equivalent units to pass</td>
</tr>
<tr>
<td>can_be_negative</td>
<td>bool</td>
<td>[DEFAULT True] values can be negative</td>
</tr>
<tr>
<td>can_be_complex</td>
<td>bool</td>
<td>[DEFAULT False] values can be complex numbers</td>
</tr>
<tr>
<td>can_be_inf</td>
<td>bool</td>
<td>[DEFAULT True] values can be inf</td>
</tr>
<tr>
<td>can_be_nan</td>
<td>bool</td>
<td>[DEFAULT True] values can be nan</td>
</tr>
<tr>
<td>none_shall_pass</td>
<td>bool</td>
<td>[DEFAULT False] values can be a python None</td>
</tr>
</tbody>
</table>

**Notes**

- Validation of function arguments *args and **kwargs is not supported.
- **None** values will pass when **None is included in the list of specified units, is set as a default value for the function argument, or **none_shall_pass** is set to **True**. If **none_shall_pass** is doubly/triply defined through the mentioned options, then they all must be consistent with each other.
- If units are not specified in **validations**, then the decorator will attempt to identify desired units by examining the function annotations.
- Full functionality is defined by the class **ValidateQuantities**.

**Examples**

Define unit and value validations with decorator parameters:

```python
import astropy.units as u
from plasmapy.utils.decorators import validate_quantities

@validate_quantities(mass={'units': u.g, 'can_be_negative': False},
    vel=u.cm / u.s,)
```

(continues on next page)
def foo(mass, vel):
    return mass * vel

# on a method
class Foo:
    @validate_quantities
    def bar(self, mass, vel):
        return mass * vel

Define units with function annotations:

```python
import astropy.units as u
from plasmapy.utils.decorators import validate_quantities

@validate_quantities
def foo(mass: u.g, vel: u.cm / u.s) -> u.g * u.cm / u.s:
    return mass * vel

# rely only on annotations
@validate_quantities
def foo(x: u.cm, time: u.s) -> u.cm / u.s:
    return x / time

# on a method
class Foo:
    @validate_quantities
    def bar(self, mass: u.g, vel: u.cm / u.s) -> u.g * u.cm / u.s:
        return mass * vel
```

Allow None values to pass:

```python
import astropy.units as u
from plasmapy.utils.decorators import validate_quantities

@validate_quantities(arg2={'none_shall_pass': True},
                     checks_on_return=[u.cm, None])
def foo(arg1: u.cm = None, arg2: u.cm):
    return None
```

Allow return values to have equivalent units:

```python
import astropy.units as u
from plasmapy.utilsdecorators import validate_quantities

@validate_quantities(arg1={'units': u.cm},
                     checks_on_return={'units': u.km,
                                       'pass_equivalent_units': True})
def foo(arg1):
    return arg1
```

Allow equivalent units to pass with specified equivalencies:
```python
import astropy.units as u
from plasmapy.utils.decorators import validate_quantities

@validate_quantities(arg1={'units': u.K,
                           'equivalencies': u.temperature(),
                           'pass_equivalent_units': True})
def foo(arg1):
    return arg1
```

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CheckBase</strong></td>
<td>Base class for ‘Check’ decorator classes.</td>
</tr>
<tr>
<td><strong>CheckUnits</strong></td>
<td>A decorator class to ‘check’ – limit/control – the units of input and return arguments to a function or method.</td>
</tr>
<tr>
<td><strong>CheckValues</strong></td>
<td>A decorator class to ‘check’ – limit/control – the values of input and return arguments to a function or method.</td>
</tr>
<tr>
<td><strong>ValidateQuantities</strong></td>
<td>A decorator class to ‘validate’ – control and convert – the units and values of input and return arguments to a function or method.</td>
</tr>
</tbody>
</table>

#### CheckBase

```python
class plasmapy.utils.decorators.CheckBase(checs_on_return=None, **checks):
    Bases: object
    Base class for ‘Check’ decorator classes.

    Parameters
    ----------
    * checks_on_return – specified checks on the return of the wrapped function
    * **checks – specified checks on the input arguments of the wrapped function
```

#### Attributes Summary

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checks</td>
<td>Requested checks on the decorated function’s input arguments and/or return.</td>
</tr>
</tbody>
</table>

#### Attributes Documentation

**checks**

Requested checks on the decorated function’s input arguments and/or return.

#### CheckUnits

```python
class plasmapy.utils.decorators.CheckUnits(checs_on_return: Union[astropy.units.core.Unit,
                                                                 List[astropy.units.core.Unit], Dict[str, Any]] = None, **checks):
    Bases: plasmapy.utils.decorators.CheckBase
```

Chapter 2. User Documentation
A decorator class to ‘check’ – limit/control – the units of input and return arguments to a function or method.

Parameters

- **checks_on_return** (list of astropy **units** or dict of unit specifications) – Specifications for unit checks on the return of the function being wrapped. (see **check units** for valid specifications)

- **checks** (list of astropy **units** or dict of unit specifications) – Specifications for unit checks on the input arguments of the function being wrapped. Each keyword argument in **checks** is the name of a function argument to be checked and the keyword value contains the unit check specifications.

Unit checks can be defined by passing one of the astropy **units**, a list of astropy units, or a dictionary containing the keys defined below. Units can also be defined with function annotations, but must be consistent with decorator **checks** arguments if used concurrently. If a key is omitted, then the default value will be assumed.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units</td>
<td></td>
<td>list of desired astropy <strong>units</strong></td>
</tr>
<tr>
<td>equivalencies</td>
<td></td>
<td>[DEFAULT None] A list of equivalent pairs to try if the units are not directly convertible. (see <strong>equivalencies</strong>, and/or astropy equivalencies)</td>
</tr>
<tr>
<td>pass_equivalent_units</td>
<td>bool</td>
<td>[DEFAULT False] allow equivalent units to pass</td>
</tr>
</tbody>
</table>

Notes

- Checking of function arguments ***args** and ***kwargs** is not supported.
- Decorator does NOT perform any unit conversions.
- If it is desired that **None** values do not raise errors or warnings, then include **None** in the list of units or as a default value for the function argument.
- If units are not specified in **checks**, then the decorator will attempt to identify desired units by examining the function annotations.

Examples

Define units with decorator parameters:
import astropy.units as u
from plasmapy.utils.decorators import CheckUnits

@CheckUnits(arg1={'units': u.cm},
            arg2=u.cm,
            checks_on_return=[u.cm, u.km])
def foo(arg1, arg2):
    return arg1 + arg2

# or on a method
class Foo:
    @CheckUnits(arg1={'units': u.cm},
                arg2=u.cm,
                checks_on_return=[u.cm, u.km])
def bar(self, arg1, arg2):
    return arg1 + arg2

Define units with function annotations:

import astropy.units as u
from plasmapy.utils.decorators import CheckUnits

@CheckUnits()

def foo(arg1: u.cm, arg2: u.cm) -> u.cm:
    return arg1 + arg2

# or on a method
class Foo:
    @CheckUnits()
    def bar(self, arg1: u.cm, arg2: u.cm) -> u.cm:
        return arg1 + arg2

Allow None values to pass, on input and output:

import astropy.units as u
from plasmapy.utils.decorators import CheckUnits

@CheckUnits(checks_on_return=[u.cm, None])
def foo(arg1: u.cm = None):
    return arg1

Allow return values to have equivalent units:

import astropy.units as u
from plasmapy.utils.decorators import CheckUnits

@CheckUnits(arg1={'units': u.cm},
            checks_on_return={'units': u.km,
                              'pass_equivalent_units': True})
def foo(arg1):
    return arg1

Allow equivalent units to pass with specified equivalencies:
@CheckUnits(arg1={'units': u.K,  
'equivalencies': u.temperature_energy(),  
'pass_equivalent_units': True})

def foo(arg1):
    return arg1

Methods Summary

```
__call__(f)
```

*param f* Function to be wrapped

Methods Documentation

```
__call__(f)
```

- **Parameters**
  - `f` – Function to be wrapped

- **Returns**
  - wrapped function of `f`

- **Return type**
  - function

CheckValues

class plasmapy.utils.decorators.CheckValues(checks_on_return: Dict[str, bool] = None, **checks)

Bases: plasmapy.utils.decorators.CheckBase

A decorator class to ‘check’ – limit/control – the values of input and return arguments to a function or method.

- **Parameters**
  - `checks_on_return` *(Dict[str, bool]*) – Specifications for value checks on the return of the function being wrapped. (see check values for valid specifications)

  - `**checks` *(Dict[str, Dict[str, bool]]*) – Specifications for value checks on the input arguments of the function being wrapped. Each keyword argument in `checks` is the name of a function argument to be checked and the keyword value contains the value check specifications.

The value check specifications are defined within a dictionary containing the keys defined below. If the dictionary is empty or omitting keys, then the default value will be assumed for the missing keys.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>can_be_negative</td>
<td>bool</td>
<td>[DEFAULT True] values can be negative</td>
</tr>
<tr>
<td>can_be_complex</td>
<td>bool</td>
<td>[DEFAULT False] values can be complex numbers</td>
</tr>
<tr>
<td>can_be_inf</td>
<td>bool</td>
<td>[DEFAULT True] values can be <code>inf</code></td>
</tr>
<tr>
<td>can_be_nan</td>
<td>bool</td>
<td>[DEFAULT True] values can be <code>nan</code></td>
</tr>
<tr>
<td>none_shall_pass</td>
<td>bool</td>
<td>[DEFAULT False] values can be a python <code>None</code></td>
</tr>
</tbody>
</table>
Notes

- Checking of function arguments `*args` and `**kwargs` is not supported.

Examples

```python
from plasmapy.utils.decorators.checks import CheckValues

@CheckValues(arg1={'can_be_negative': False, 'can_be_nan': False},
             arg2={'can_be_inf': False},
             checks_on_return={'none_shall_pass': True})
def foo(arg1, arg2):
    return None

# on a method
class Foo:
    @CheckValues(arg1={'can_be_negative': False, 'can_be_nan': False},
                 arg2={'can_be_inf': False},
                 checks_on_return={'none_shall_pass': True})
def bar(self, arg1, arg2):
    return None
```

Methods Summary

- `_call_`(f)
  
  **param f**  Function to be wrapped

Methods Documentation

 `_call_`(f)

  Parameters  
  
  f – Function to be wrapped

  Returns  
  
  wrapped function of f

  Return type  
  
  function

ValidateQuantities

```python
class plasmapy.utils.decorators.ValidateQuantities(validations_on_return=None, **validations):
    pass
```

A decorator class to ‘validate’ – control and convert – the units and values of input and return arguments to a function or method. Arguments are expected to be astropy `Quantity` objects.

Parameters

- `validations_on_return`  
  
  (dictionary of validation specifications) – Specifications for unit and value validations on the return of the function being wrapped. (see `quantity validations` for valid specifications.)
**validations** *(dictionary of validation specifications)* – Specifications for unit and value validations on the input arguments of the function being wrapped. Each keyword argument in `validations` is the name of a function argument to be validated and the keyword value contains the unit and value validation specifications.

Unit and value validations can be defined by passing one of the astropy `units`, a list of astropy units, or a dictionary containing the keys defined below. Units can also be defined with function annotations, but must be consistent with decorator `**validations` arguments if used concurrently. If a key is omitted, then the default value will be assumed.

<table>
<thead>
<tr>
<th>Key</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>units</td>
<td>list of desired astropy units</td>
<td></td>
</tr>
<tr>
<td>equivalencies</td>
<td>[DEFAULT None] A list of equivalent pairs to try if units are not directly convertible. (see equivalencies, and/or astropy equivalencies)</td>
<td></td>
</tr>
<tr>
<td>pass_equivalent_units</td>
<td>bool</td>
<td>[DEFAULT False] allow equivalent units to pass</td>
</tr>
<tr>
<td>can_be_negative</td>
<td>bool</td>
<td>[DEFAULT True] values can be negative</td>
</tr>
<tr>
<td>can_be_complex</td>
<td>bool</td>
<td>[DEFAULT False] values can be complex numbers</td>
</tr>
<tr>
<td>can_be_inf</td>
<td>bool</td>
<td>[DEFAULT True] values can be inf</td>
</tr>
<tr>
<td>can_be_nan</td>
<td>bool</td>
<td>[DEFAULT True] values can be nan</td>
</tr>
<tr>
<td>none_shall_pass</td>
<td>bool</td>
<td>[DEFAULT False] values can be a python None</td>
</tr>
</tbody>
</table>

**Notes**

- Validation of function arguments `*args` and `**kwargs` is not supported.
- `None` values will pass when `None` is included in the list of specified units, is set as a default value for the function argument, or `none_shall_pass` is set to `True`. If `none_shall_pass` is doubly/triply defined through the mentioned options, then they all must be consistent with each other.
- If units are not specified in `validations`, then the decorator will attempt to identify desired units by examining the function annotations.

**Examples**

Define unit and value validations with decorator parameters:
import astropy.units as u
from plasmapy.utils.decorators import ValidateQuantities

@ValidateQuantities(mass={'units': u.g, 'can_be_negative': False},
                    vel=u.cm / u.s,
                    validations_on_return=[u.g * u.cm / u.s, u.kg * u.m / u.s])
def foo(mass, vel):
    return mass * vel

# on a method
class Foo:
    @ValidateQuantities(mass={'units': u.g, 'can_be_negative': False},
                        vel=u.cm / u.s,
                        validations_on_return=[u.g * u.cm / u.s, u.kg * u.m / u.s])
    def bar(self, mass, vel):
        return mass * vel

Define units with function annotations:

import astropy.units as u
from plasmapy.utils.decorators import ValidateQuantities

@ValidateQuantities(mass={'can_be_negative': False})
def foo(mass: u.g, vel: u.cm / u.s) -> u.g * u.cm / u.s:
    return mass * vel

# on a method
class Foo:
    @ValidateQuantities(mass={'can_be_negative': False})
    def bar(self, mass: u.g, vel: u.cm / u.s) -> u.g * u.cm / u.s:
        return mass * vel

Allow None values to pass:

import astropy.units as u
from plasmapy.utils.decorators import ValidateQuantities

@ValidateQuantities(checks_on_return=[u.cm, None])
def foo(arg1: u.cm = None):
    return arg1

Allow return values to have equivalent units:

import astropy.units as u
from plasmapy.utils.decorators import ValidateQuantities

@ValidateQuantities(arg1={'units': u.cm},
                    checks_on_return={'units': u.km, 'pass_equivalent_units': True})
def foo(arg1):
    return arg1

Allow equivalent units to pass with specified equivalencies:
import astropy.units as u
from plasmapy.utils.decorators import ValidateQuantities

@ValidateQuantities(arg1={'units': u.K, 'equivalencies': u.temperature(), 'pass_equivalent_units': True})
def foo(arg1):
    return arg1

Attributes Summary

validations Requested validations on the decorated function’s input arguments and return variable.

Methods Summary

__call__(f) param f Function to be wrapped

Attributes Documentation

validations Requested validations on the decorated function’s input arguments and return variable.

Methods Documentation

__call__(f)
Parameters f – Function to be wrapped
Returns wrapped function of f
Return type function

Class Inheritance Diagram
plasmapy.utils.exceptions Module

Exceptions and warnings specific to PlasmaPy.

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PlasmaPyError</td>
<td>Base class of PlasmaPy custom errors.</td>
</tr>
<tr>
<td>PhysicsError</td>
<td>The base exception for physics-related errors.</td>
</tr>
<tr>
<td>RelativityError</td>
<td>An exception for speeds greater than the speed of light.</td>
</tr>
<tr>
<td>PlasmaPyWarning</td>
<td>Base class of PlasmaPy custom warnings.</td>
</tr>
<tr>
<td>CouplingWarning</td>
<td>A warning for functions that rely on a particular coupling regime to be valid.</td>
</tr>
<tr>
<td>ImplicitUnitConversionWarning</td>
<td>A warning for an implicit conversion between equivalent astropy units.</td>
</tr>
<tr>
<td>PhysicsWarning</td>
<td>The base warning for warnings related to non-physical situations.</td>
</tr>
<tr>
<td>RelativityWarning</td>
<td>A warning for when relativistic velocities are being used in or are returned by non-relativistic functionality.</td>
</tr>
</tbody>
</table>

PlasmaPyError

```python
exception plasmapy.utils.exceptions.PlasmaPyError
    Base class of PlasmaPy custom errors.
```

All custom exceptions raised by PlasmaPy should inherit from this class and be defined in this module.

PhysicsError

```python
exception plasmapy.utils.exceptions.PhysicsError
    The base exception for physics-related errors.
```

RelativityError

```python
exception plasmapy.utils.exceptions.RelativityError
    An exception for speeds greater than the speed of light.
```

PlasmaPyWarning

```python
exception plasmapy.utils.exceptions.PlasmaPyWarning
    Base class of PlasmaPy custom warnings.
```

All PlasmaPy custom warnings should inherit from this class and be defined in this module.

Warnings should be issued using `warn`, which will not break execution if unhandled.

CouplingWarning

```python
exception plasmapy.utils.exceptions.CouplingWarning
    A warning for functions that rely on a particular coupling regime to be valid.
```
**ImplicitUnitConversionWarning**

```python
exception plasmapy.utils.exceptions.ImplicitUnitConversionWarning
A warning for an implicit conversion between equivalent astropy units.
```

**PhysicsWarning**

```python
exception plasmapy.utils.exceptions.PhysicsWarning
The base warning for warnings related to non-physical situations.
```

**RelativityWarning**

```python
exception plasmapy.utils.exceptions.RelativityWarning
A warning for when relativistic velocities are being used in or are returned by non-relativistic functionality.
```

---

**Class Inheritance Diagram**

![Class Inheritance Diagram]

---

**plasmapy.utils.pytest_helpers Package**

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert_can_handle_nparray(function_to_test)</code></td>
<td>Test for ability to handle numpy array quantities.</td>
</tr>
<tr>
<td><code>run_test(func, args, kwargs, VT = {}, ...)</code></td>
<td>Test that a function or class returns the expected result, raises the expected exception, or issues an expected warning for the supplied positional and keyword arguments.</td>
</tr>
<tr>
<td><code>run_test_equivalent_calls(*test_inputs, ...)</code></td>
<td>Test that different functions/inputs return equivalent results.</td>
</tr>
</tbody>
</table>
assert_can_handle_nparray

```python
plasmapy.utils.pytest_helpers.assert_can_handle_nparray(function_to_test, insert_some_nans=[], insert_all_nans=[], kwars={})
```

Test for ability to handle numpy array quantities.

**Parameters**

- **function_to_test** – The function to be tested for ability to handle numpy array quantities. Arguments are automatically given a vector input based on their variable name. Current args that are interpreted as vectors are: ["T", "T_i", "T_e", "temperature"] ["n", "n_i", "n_e", "density"] ["B"] ["V", "Vperp"] ["coulomb_log"] ["characteristic_length"]

- **insert_some_nans (list)** – List of argument names in which to insert some np.nan values. These must be arguments that will be tested as vectors as listed above.

- **insert_all_nans (list)** – List of argument names to fill entirely with np.nan values.

- **kwars (dict)** – Arguments to pass directly to the function in under test, in the normal kwars python dictionary format.

**Raises** `ValueError` – If this function cannot interpret a parameter of function_to_test,

**Examples**

```python
>>> from plasmapy.formulary.parameters import Alfven_speed, gyrofrequency
>>> assert_can_handle_nparray(Alfven_speed)
>>> assert_can_handle_nparray(gyrofrequency, kwars={"signed": True})
>>> assert_can_handle_nparray(gyrofrequency, kwars={"signed": False})
```

run_test

```python
plasmapy.utils.pytest_helpers.run_test(func, args: Any = (), kwars: Dict[KT, VT] = {}, expected_outcome: Any = None, rtol: float = 0.0, atol: float = 0.0)
```

Test that a function or class returns the expected result, raises the expected exception, or issues an expected warning for the supplied positional and keyword arguments.

**Parameters**

- **func (callable, list, or tuple)** – The callable to be tested. The first (and sole) argument to run_test may alternatively be a list or tuple containing these arguments (optionally omitting kwars if the len returns 3).

- **args (tuple or object)** – The positional arguments to func.

- **kwars (dict)** – The keyword arguments to func.

- **expected_outcome (object)** – The expected result, exception, or warning from func(+args, **kwars). This may also be a tuple of length two that contains the expected result as the first item and the expected warning as the second item.

- **rtol (float)** – The relative tolerance to be used by allclose in an element-wise comparison, defaulting to 0.
• **atol** *(float)* – The absolute tolerance to be used by `allclose` in an element-wise comparison, defaulting to 0.

**Returns**

**Return type** *None*

**Raises**

*UnexpectedResultError* – If the test returns a result that is different from the expected result.

*InconsistentTypeError* – If the actual result is of a different type than the expected result.

*UnexpectedExceptionError* – If an exception occurs when no exception or a different exception is expected.

*MissingExceptionError* – If no exception is raised when an exception is expected.

*MissingWarningError* – An expected warning is not issued.

*UnitsError* – If the result has different units than expected.

*TypeError* – If the equality of the actual result and expected result cannot be determined (e.g., for a class lacking an `__eq__` method).

**Examples**

The simplest way to use `run_test` is with inputs for the function to be tests, the positional arguments in a *tuple* or *list*, the keyword arguments in a *dict*, and then finally the expected result or outcome.

```python
>>> args = tuple()
>>> kwargs = dict()
>>> run_test(lambda: 0, args, kwargs, 0)
```

If `expected` is a an exception or warning, then `run_test` will raise an exception if the expected exception is not raised or the expected warning is not issued.

```python
>>> from warnings import warn

>>> issue_warning = lambda: warn("Electrons are weird!", UserWarning)
>>> run_test(issue_warning, args, kwargs, UserWarning)
```

```python
>>> def raise_exception():
...     raise RuntimeError
... >>> run_test(raise_exception, args, kwargs, RuntimeError)
```

For warnings, `run_test` can accept a *tuple* of two items where the first item is the expected result and the second item is the expected warning.

```python
>>> def return_arg_and_warn(x):
...     warn("", UserWarning)
...     return x
... run_test(return_arg_and_warn, 1, {}, (1, UserWarning))
```

This function is also flexible enough that it can accept a *tuple* or *list* as its sole argument, with the arguments in the same order as in the function signature.
>>> return_arg = lambda x: x
>>> inputs = (return_arg, 42, {}, 42)
>>> run_test(inputs)

If the tuple or list has a length of 3, then run_test assumes that kwargs is missing.

>>> inputs_without_kwargs = [return_arg, 42, 42]
>>> run_test(inputs_without_kwargs)

import pytest
def func(x, raise_exception=False, issue_warning=False):
    if raise_exception:
        raise ValueError("I'm sorry, Dave. I'm afraid I can't do that.")
    elif issue_warning:
        warn("Open the pod bay doors, HAL.", UserWarning)
    return x

inputs_table = [
    (func, 1, 1),
    (func, (2,), {}, 2),
    (func, 3, {'raise_exception': True}, ValueError),
    (func, 4, {'issue_warning': True}, UserWarning),
    (func, 5, {'issue_warning': True}, (5, UserWarning)),
]

@pytest.mark.parametrize('inputs', inputs_table)
def test_func(inputs):
    run_test(inputs)

run_test_equivalent_calls

plasmapy.utils.pytest_helpers.run_test_equivalent_calls(*test_inputs, require_same_type: bool = True)

Test that different functions/inputs return equivalent results.

Parameters

• **test_inputs** – The functions and inputs to the tests in an allowed format, as described below.

• **require_same_type** (bool) – If True (the default), then all of the results are required to be of the same type. If False, results do not need to be of the same type (e.g., cases like 1.0 == 1 will not raise an exception).

Raises

• UnexpectedResultError – If not all of the results are equivalent, or not all of the results are of the same type and require_same_type evaluates to True.

• UnexpectedExceptionError – If an exception is raised whilst attempting to run one of the test cases.

• InvalidTestError – If there is an error associated with the inputs or the test is set up incorrectly.
Examples

There are several possible formats that can be accepted by this `run_test_equivalent_calls` to test that different combinations of functions (or other `callable` objects), positional arguments, and keyword arguments return equivalent results.

To test a single function that takes a single positional argument, then `test_inputs` may be the function followed by an arbitrary number of positional arguments to be included into the function.

```python
>>> def f(x):
    return x ** 2
>>> run_test_equivalent_calls(f, -1, 1)
```

To test a single function with an arbitrary number of positional and keyword arguments, the first argument should be the function, followed by an arbitrary number of `tuple` or `list` objects that contain a `tuple` or `list` containing the positional arguments, and a `dict` containing the keyword arguments.

```python
>>> def g(x, y, z):
    return x + y + z
>>> run_test_equivalent_calls(g, ((1, 2, 3), {}), ((3, 2), {'z': 1}))
```

If there is only one positional argument, then it is not necessary to include it in a `tuple` or `list`.

```python
>>> run_test_equivalent_calls(f, (1,), ({},))
>>> run_test_equivalent_calls(f, (1,), (1, {}))
```

To test multiple functions with an arbitrary number of positional and keyword arguments, use a series of `tuple` or `list` objects that contain the function for each test, a `tuple` or `list` with the positional arguments, and a `dict` with the keyword arguments.

```python
>>> def p(x, y=None): return x + y if y else x
>>> def q(x, y=None): return x + 1 if y else x
>>> run_test_equivalent_calls([p, (1,), {'y': 1}], [q, (2,), {'y': False}])
```

The inputs may also be passed in as a whole as a `tuple` or `list`.

```python
>>> run_test_equivalent_calls(f, -1, 1)
>>> run_test_equivalent_calls([f, -1, 1])
```

If `require_same_type` is `False`, then an exception will not be raised if the results are of different types.

```python
>>> run_test_equivalent_calls(f, -1, 1.0, require_same_type=False)
```

Classes

- **InconsistentTypeError**: Exception for when the type of the actual result differs from the type of the expected result.
- **IncorrectResultError**: Exception for when the actual result differs from the expected result by more than the allowed tolerance.
- **InvalidTestError**: Exception for when the inputs to a test are not valid.
- **MissingExceptionError**: Exception for when an expected exception is not raised.
- **MissingWarningError**: Exception for when a warning is expected to be issued, but isn’t.
- **RunTestError**: Base exception for test failures.
InconsistencyTypeError

exception plasmapy.utils.pytest_helpers.InconsistencyTypeError
    Exception for when the type of the actual result differs from the type of the expected result. Derived from RunTestError.

IncorrectResultError

exception plasmapy.utils.pytest_helpers.IncorrectResultError
    Exception for when the actual result differs from the expected result by more than the allowed tolerance. Derived from RunTestError.

InvalidTestError

exception plasmapy.utils.pytest_helpers.InvalidTestError
    Exception for when the inputs to a test are not valid.

MissingExceptionError

exception plasmapy.utils.pytest_helpers.MissingExceptionError
    Exception for when an expected exception is not raised. Derived from RunTestError.

MissingWarningError

exception plasmapy.utils.pytest_helpers.MissingWarningError
    Exception for when a warning is expected to be issued, but isn’t. Derived from RunTestError.

RunTestError

exception plasmapy.utils.pytest_helpers.RunTestError
    Base exception for test failures. Derived from Exception.

UnexpectedExceptionError

exception plasmapy.utils.pytest_helpers.UnexpectedExceptionError
    Exception for when an exception is expected, but a different exception is raised instead. Derived from RunTestError.
UnexpectedResultError

```python
exception plasmapy.utils.pytest_helpers.UnexpectedResultError
```

Exception for when the actual result differs from the expected result. Derived from RunTestError.

Class Inheritance Diagram

```
 InconsistentTypeError
   
   IncorrectResultError
   
   MissingExceptionError
   
   MissingWarningError
   
   UnexpectedExceptionError
   
   UnexpectedResultError
```

2.6 Examples

2.6.1 Examples

General examples

General-purpose and introductory examples from PlasmaPy.

You may also run these online via Binder (link uses the current master version).

Note: Click here to download the full example code or to run this example in your browser via Binder

1D Maxwellian distribution function

We import the usual modules, and the hero of this notebook, the Maxwellian 1D distribution:
import numpy as np
from astropy import units as u
import matplotlib.pyplot as plt
from astropy.constants import (m_e, k_B)
from plasmapy.formulary import Maxwellian_1D

Given we’ll be plotting, import astropy’s quantity support:

from astropy.visualization import quantity_support
quantity_support()

Out:
<astropy.visualization.units.quantity_support.<locals>.MplQuantityConverter object at 0x7f77ffe20320>

As a first example, let’s get the probability density of finding an electron with a speed of 1 m/s if we have a plasma at a temperature of 30 000 K:

p_dens = Maxwellian_1D(v=1 * u.m / u.s,
                       T=30000 * u.K,
                       particle='e',
                       v_drift=0 * u.m / u.s)
print(p_dens)

Out:
5.916328704912824e-07 s / m

Note the units! Integrated over speed, this will give us a probability. Let’s test that for a bunch of particles:

T = 3e4 * u.K
dv = 10 * u.m / u.s
v = np.arange(-5e6, 5e6, 10) * u.m / u.s

Check that the integral over all speeds is 1 (the particle has to be somewhere):

for particle in ['p', 'e']:
    pdf = Maxwellian_1D(v, T=T, particle=particle)
    integral = (pdf).sum() * dv
    print(f"Integral value for {particle}: {integral}"")
plt.plot(v, pdf, label=particle)
plt.legend()
Out:

Integral value for p: 1.0000000000000002
Integral value for e: 0.9999999999998787
<matplotlib.legend.Legend object at 0x7f77fec1fc50>

The standard deviation of this distribution should give us back the temperature:

```python
std = np.sqrt((Maxwellian_1D(v, T=T, particle='e') * v ** 2 * dv).sum())
T_theo = (std ** 2 / k_B * m_e).to(u.K)
```

```bash
print('T from standard deviation:', T_theo)
print('Initial T:', T)
```

Out:

```
T from standard deviation: 29999.999999792235 K
Initial T: 30000.0 K
```

Total running time of the script: ( 0 minutes 1.494 seconds)

Note: Click here to download the full example code or to run this example in your browser via Binder
Particle stepper

An example of PlasmaPy’s particle stepper class, currently in need of a rewrite for speed.

```python
import numpy as np
from astropy import units as u
from plasmapy.classes import Plasma
from plasmapy.simulation import ParticleTracker
from plasmapy.formulary import gyrofrequency

plasma = Plasma(domain_x=np.linspace(-1, 1, 10) * u.m,
                 domain_y=np.linspace(-1, 1, 10) * u.m,
                 domain_z=np.linspace(-1, 1, 10) * u.m)

B0 = 4 * u.T
plasma.magnetic_field[0, :, :, :] = np.ones((10, 10, 10)) * B0
E0 = 2 * u.V / u.m
plasma.electric_field[1, :, :, :] = np.ones((10, 10, 10)) * E0

freq = gyrofrequency(B0, 'p').to(u.Hz, equivalencies=u.dimensionless_angles())
gyroperiod = (1/freq).to(u.s)
steps_to_gyroperiod = 10
timestep = gyroperiod / steps_to_gyroperiod

number_steps = steps_to_gyroperiod * int(2 * np.pi)
trajectory = ParticleTracker(plasma, 'p', 1, 1, timestep, number_steps)

trajectory.v[0][0] = 1 * (u.m / u.s)

trajectory.run()
trajectory.plot_time_trajectories()
```
Plot the shape of the trajectory in 3D.

```
trajectory.plot_trajectories()
```
As a test, we calculate the mean velocity in the z direction from the velocity and position:

```python
vmean = trajectory.velocity_history[:, :, 2].mean()
print(f"The calculated drift velocity is \(vmean=\{vmean:.4f\}\) to compare with the theoretical \(E_0/B_0=\{E0/B0:.4f\}\)"
```

Out:

```
The calculated drift velocity is 0.5233 m / s to compare with the theoretical \(E_0/B_0=0.5000\) V / (m T)
```

and from position:

```python
Vdrift = trajectory.position_history[-1, 0, 2] / (trajectory.NT * trajectory.dt)
print(f"The calculated drift velocity from position is \(Vdrift=\{Vdrift:.4f\}\")
```

Out:

```
The calculated drift velocity from position is 0.5232 m / s
```

**Total running time of the script:** (0 minutes 0.471 seconds)

**Note:** Click [here](#) to download the full example code or to run this example in your browser via Binder.
The plasma dispersion function

Let's import some basics (and PlasmaPy!)

```python
import numpy as np
import matplotlib.pyplot as plt

import plasmapy.formulary.dispersionfunction
help(plasmapy.formulary.dispersionfunction.plasma_dispersion_func)
```

```
Help on function plasma_dispersion_func in module plasmapy.formulary.

plasma_dispersion_func(zeta: Union[complex, int, float, numpy.ndarray, astropy.units.Quantity]) -> Union[complex, float, numpy.ndarray, astropy.units.Quantity]
    Calculate the plasma dispersion function.
    Parameters
    ----------
    zeta : complex, int, float, ~numpy.ndarray, or ~astropy.units.Quantity
        Argument of plasma dispersion function.
    Returns
    -------
    Z : complex, float, or ~numpy.ndarray
        Value of plasma dispersion function.
    Raises
    ------
    TypeError
        If the argument is of an invalid type.
    ~astropy.units.UnitsError
        If the argument is a `~astropy.units.Quantity` but is not dimensionless.
    ValueError
        If the argument is not entirely finite.
    See Also
    --------
    plasma_dispersion_func_deriv
    Notes
    -----
    The plasma dispersion function is defined as:
    .. math:
        \mathcal{Z}(\zeta) = \pi^{-0.5} \int_{-\infty}^{+\infty} \frac{e^{-x^2}}{x-\zeta} \, dx
    where the argument is a complex number [fried.conte-1961].
    ```

(continues on next page)

2.6. Examples
In plasma wave theory, the plasma dispersion function appears frequently when the background medium has a Maxwellian distribution function. The argument of this function then refers to the ratio of a wave's phase velocity to a thermal velocity.

References
----------

Examples
--------
>>> plasma_dispersion_func(0)
1.7724538509055159j
>>> plasma_dispersion_func(1j)
0.757872156141312j
>>> plasma_dispersion_func(-1.52+0.47j)
(0.608888957234254+0.33494583882874024j)

We'll now make some sample data to visualize the dispersion function:

```python
x = np.linspace(-1, 1, 1000)
X, Y = np.meshgrid(x, x)
Z = X + 1j * Y
print(Z.shape)
Out:
(1000, 1000)

Before we start plotting, let's make a visualization function first:

def plot_complex(X, Y, Z, N=50):
    fig, (real_axis, imag_axis) = plt.subplots(1, 2)
    real_axis.contourf(X, Y, Z.real, N)
    imag_axis.contourf(X, Y, Z.imag, N)
    real_axis.set_title("Real values")
    imag_axis.set_title("Imaginary values")
    for ax in [real_axis, imag_axis]:
        ax.set_xlabel("Real values")
        ax.set_ylabel("Imaginary values")
    fig.tight_layout()
plot_complex(X, Y, Z)
```
We can now apply our visualization function to our simple dispersion relation

```python
# sphinx_gallery_thumbnail_number = 2
F = plasmapy.formulary.dispersionfunction.plasma_dispersion_func(Z)
plot_complex(X, Y, F)
```
So this is going to be a hack and I’m not 100% sure the dispersion function is quite what I think it is, but let’s find the area where the dispersion function has a lesser than zero real part because I think it may be important (brb reading Fried and Conte):

```
plot_complex(X, Y, F.real < 0)
```
We can also visualize the derivative:

```python
F = plasmapy.formulary.dispersionfunction.plasma_dispersion_func_deriv(Z)
plot_complex(X, Y, F)
```
Plotting the same function on a larger area:

```python
x = np.linspace(-2, 2, 2000)
X, Y = np.meshgrid(x, x)
Z = X + 1j * Y
print(Z.shape)
```

Out:

```
(2000, 2000)
```

```python
F = plasmapy.formulary.dispersionfunction.plasma_dispersion_func(Z)
plot_complex(X, Y, F, 100)
```
Now we examine the derivative of the dispersion function as a function of the phase velocity of an electromagnetic wave propagating through the plasma. This is recreating figure 5.1 in: J. Sheffield, D. Froula, S. H. Glenzer, and N. C. Luhmann Jr. Plasma scattering of electromagnetic radiation: theory and measurement techniques. Chapter 5 Pg 106 (Academic press, 2010).

```python
xs = np.linspace(0, 4, 100)
ws = (-1 / 2) * plasmapy.formulary.dispersionfunction.plasma_dispersion_func_deriv(xs)
wRe = np.real(ws)
wIm = np.imag(ws)

plt.plot(xs, wRe, label="Re")
plt.plot(xs, wIm, label="Im")
plt.axis([0, 4, -0.3, 1])
plt.legend(loc='upper right', frameon=False, labelspacing=0.001, fontsize=14, borderaxespad=0.1)
plt.show()
```
Braginskii coefficients

A short example of how to calculate classical transport coefficients from Braginski’s theory.

```python
from astropy import units as u
from plasmapy.formulary import ClassicalTransport
```

We’ll use some sample ITER data, without much regard for whether the regime is even fit for classical transport theory:

```python
thermal_energy_per_electron = 8.8 * u.keV
electron_concentration = 10.1e19 / u.m**3

thermal_energy_per_ion = 8.0 * u.keV
ion_concentration = electron_concentration
ion_particle = 'D+' # a crude approximation
```

We now make the default ClassicalTransport object:
These variables are calculated during initialization and can be referred to straight away:

```python
print(braginskii.coulomb_log_ei)
print(braginskii.coulomb_log_ii)
print(braginskii.hall_e)
print(braginskii.hall_i)
```

Out:

```
18.015542112815666
20.41557520752423
0.0
0.0
```

These quantities are not calculated during initialization and can be referred to via methods. To signify the need to calculate them, we call them via ()

2.6. Examples
```
print(braginskii.resistivity)
print(braginskii.thermoelectric_conductivity)
print(braginskii.electron_thermal_conductivity)
print(braginskii.ion_thermal_conductivity)
```

```
They also change with magnetization:

```
mag_braginskii = ClassicalTransport(thermal_energy_per_electron,
    electron_concentration,
    thermal_energy_per_ion,
    ion_concentration,
    ion_particle,
    B = 0.1 * u.T)
```

```
print(mag_braginskii.resistivity)
print(mag_braginskii.thermoelectric_conductivity)
print(mag_braginskii.electron_thermal_conductivity)
print(mag_braginskii.ion_thermal_conductivity)
```

```
They also change with direction with respect to the magnetic field. Here, we choose to print out, as arrays, the (parallel, perpendicular, and cross) directions. Take a look at the docs to ClassicalTransport for more information on these.
```

```python
classical_all_direction = ClassicalTransport(thermal_energy_per_electron,
    electron_concentration,
    thermal_energy_per_ion,
    ion_concentration,
    ion_particle,
    B = 0.1 * u.T,
    field_orientation = 'all')
```
The viscosities return arrays:

```python
print(braginskii.electron_viscosity)
print(mag_braginskii.electron_viscosity)
print(braginskii.ion_viscosity)
print(mag_braginskii.ion_viscosity)
```

Out:

```python
[16.29411376 16.28874805 16.28874805 0. 0. ] Pa s
[1.62941138e+01 2.00480711e-25 8.01922844e-25 1.47442522e-12
2.94885044e-12] Pa s
[1271.38945503 1267.52435833 1267.52435833 0. 0. ] Pa s
[1.27138946e+03 5.99222933e-17 2.39689173e-16 2.57162285e-07
5.14324570e-07] Pa s
```

Total running time of the script: ( 0 minutes 0.416 seconds)

Note: Click [here](#) to download the full example code or to run this example in your browser via Binder

### Analysing ITER parameters

Let’s try to look at ITER plasma conditions using the `physics` subpackage.

```python
from astropy import units as u
from plasmapy import formulary
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
```

The radius of electric field shielding clouds, also known as the Debye length, would be

```python
electron_temperature = 8.8 * u.keV
electron_concentration = 10.1e19 / u.m**3
print(formulary.Debye_length(electron_temperature, electron_concentration))
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/plasmapy/checkouts/stable/plasmapy/utils/decorators/validators.py:357: ImplicitUnitConversionWarning: The argument 'T_e' to function Debye_length() has a non-standard unit conversion...converting keV to K
  f"{err_msg} has a non-standard unit conversion..."
6.939046810942984e-05 m
```

Note that we can also neglect the unit for the concentration, as 1/m^3 is the a standard unit for this kind of Quantity:

```python
print(formulary.Debye_length(electron_temperature, 10.1e19))
```

Out:
Assuming the magnetic field as 5.3 Teslas (which is the value at the major radius):

```python
B = 5.3 * u.T
print(formulary.gyrofrequency(B, particle='e'))
print(formulary.gyroradius(B, T_i=electron_temperature, particle='e'))
```

Out:

```plaintext
932174605709.2465 rad / s
5.968562765183547e-05 m
```

The electron inertial length would be

```python
print(formulary.inertial_length(electron_concentration, particle='e'))
```

Out:

```plaintext
0.0005287720427518426 m
```

In these conditions, they should reach thermal velocities of about

```python
print(formulary.thermal_speed(T=electron_temperature, particle='e'))
```

Out:

```plaintext
(continues on next page)```
And the Langmuir wave plasma frequency should be on the order of

```python
print(formulary.plasma_frequency(electron_concentration))
```

Out:

```
566959736448.652 rad / s
```

Let's try to recreate some plots and get a feel for some of these quantities. We will also compare our data to real-world plasma situations.

```python
n_e = np.logspace(4, 30, 100) / u.m**3
plt.plot(n_e, formulary.plasma_frequency(n_e))
plt.scatter(electron_concentration, formulary.plasma_frequency(electron_concentration), label="Our Data")

# IRT1 Tokamak Data
# http://article.sapub.org/pdf/10.5923.j.jnpp.20110101.03.pdf
n_e = 1.2e19 / u.m**3
T_e = 136.8323 * u.eV
B = 0.82 * u.T
plt.scatter(n_e, formulary.plasma_frequency(n_e), label="IRT1 Tokamak")

# Wendelstein 7-X Stellerator Data
n_e = 3e19 / u.m**3
T_e = 6 * u.keV
B = 3 * u.T
plt.scatter(n_e, formulary.plasma_frequency(n_e), label="W7-X Stellerator")

# Solar Corona
# Estimated by Nick Murphy
n_e = 1e15 / u.m**3
T_e = 1e6 * u.K
B = 0.005 * u.T
T_e.to(u.eV, equivalencies=u.temperature_energy())
plt.scatter(n_e, formulary.plasma_frequency(n_e), label="Solar Corona")

# Interstellar (warm neutral) Medium
# Estimated by Nick Murphy
n_e = 1e6 / u.m**3
T_e = 5e3 * u.K
B = 0.005 * u.T
T_e.to(u.eV, equivalencies=u.temperature_energy())
plt.scatter(n_e, formulary.plasma_frequency(n_e), label="Interstellar Medium")

# Solar Wind at 1 AU
# Estimated by Nick Murphy
n_e = 7e6 / u.m**3
T_e = 1e5 * u.K
```

(continues on next page)
B = 5e-9 * u.T
T_e.to(u.eV, equivalencies=u.temperature_energy())
plt.scatter(n_e, formulary.plasma_frequency(n_e), label="Solar Wind (1AU)"
plt.xlabel("Electron Concentration (m^-3)"
plt.ylabel("Langmuir Wave Plasma Frequency (rad/s)"
plt.grid()
plt.xscale("log")
plt.yscale("log")
plt.legend()
plt.title("Log-scale plot of plasma frequencies")
plt.show()

Total running time of the script: ( 0 minutes 0.337 seconds)

Note: Click here to download the full example code or to run this example in your browser via Binder

Langmuir probe data analysis

Let’s analyze a few Langmuir probe characteristics using the diagnostics.langmuir subpackage. First we need to import the module and some basics.
The first characteristic we analyze is a simple single-probe measurement in a low (ion) temperature, low density plasma with a cylindrical probe. This allows us to utilize OML theory implemented in `swept_probe_analysis`. The data has been preprocessed with some smoothing, which allows us to obtain an Electron Energy Distribution Function (EEDF) as well.

```python
from plasmapy.diagnostics.langmuir import Characteristic, swept_probe_analysis
import matplotlib.pyplot as plt
import astropy.units as u
import numpy as np
import os
from pprint import pprint

# Load the bias and current values stored in the .p pickle file.
path = os.path.join("langmuir_samples", "Beckers2017.npy")
bias, current = np.load(path)

# Create the Characteristic object, taking into account the correct units
characteristic = Characteristic(u.Quantity(bias, u.V),
                                u.Quantity(current, u.A))

# Calculate the cylindrical probe surface area
probe_length = 1.145 * u.mm
probe_diameter = 1.57 * u.mm
probe_area = (probe_length * np.pi * probe_diameter +
              np.pi * 0.25 * probe_diameter**2)

print(swept_probe_analysis(characteristic,
                             probe_area, 'He-4+',
                             visualize=True,
                             plot_EEDF=True))
```

Now we can actually perform the analysis. Since the plasma is in Helium an ion mass number of 4 is entered. The results are visualized and the obtained EEDF is also shown.

```python
# Load the bias and current values stored in the .p pickle file.
path = os.path.join("langmuir_samples", "Beckers2017.npy")
bias, current = np.load(path)

# Create the Characteristic object, taking into account the correct units
characteristic = Characteristic(u.Quantity(bias, u.V),
                                u.Quantity(current, u.A))

# Calculate the cylindrical probe surface area
probe_length = 1.145 * u.mm
probe_diameter = 1.57 * u.mm
probe_area = (probe_length * np.pi * probe_diameter +
              np.pi * 0.25 * probe_diameter**2)

print(swept_probe_analysis(characteristic,
                             probe_area, 'He-4+',
                             visualize=True,
                             plot_EEDF=True))
```
The cyan and yellow lines indicate the fitted electron and ion currents, respectively. The green line is the sum of these and agrees nicely with the data. This indicates a successful analysis.

The next sample probe data is provided by David Pace. It is also obtained from a low relatively ion temperature and density plasma, in Argon.

Initially the electrons are assumed to be Maxwellian. To check this the fit of the electron growth region will be plotted.
It can be seen that this plasma is slightly bi-Maxwellian, as there are two distinct slopes in the exponential section. The analysis is now performed with bimaxwellian set to True, which yields improved results.
The probe current resolution of the raw data is relatively poor, but the analysis still performs well in the ion current region. The bi-Maxwellian properties are not significant but do make a difference. Check this analysis without setting `bimaxwellian` to True! This is reflected in the results, which indicate that the temperatures of the cold and hot electron population are indeed different, but relatively close.

This Helium plasma is fully bi-Maxwellian.

```python
# Import probe data and calculate probe surface area.
path = os.path.join("langmuir_samples", "Beckers2017b.npy")
bias, current = np.load(path)
characteristic = Characteristic(u.Quantity(bias, u.V),
                        u.Quantity(current, u.A))
probe_length = 1.145 * u.mm
probe_diameter = 1.57 * u.mm
probe_area = (probe_length * np.pi * probe_diameter +
```

(continues on next page)
\[ \pi \times 0.25 \times \text{probe_diameter}^2 \]

plot\_electron\_fit is set to True to check the bi-Maxwellian properties. The fit converges nicely to the two slopes of the electron growth region.

```python
pprint(swept_probe_analysis(characteristic,
                            probe_area,
                            'He-4+',
                            bimaxwellian=True,
                            plot\_electron\_fit=True,
                            visualize=True))
```

### Exponential fit

- Bimaxwellian exponential section fit
- Exponential fit
- Exponential section
Cold Magnetized Plasma Waves Tensor Elements (S, D, P in Stix’s notation)

This example shows how to calculate the values of the cold plasma tensor elements for various electromagnetic wave frequencies.

```python
# First, import some basics (and `PlasmaPy`)!
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)
Let's define some parameters, such as the magnetic field magnitude, the plasma species and densities and the frequency band of interest

\[
B = 2 \times \text{u.T} \\
\text{species} = ['e', 'D+'] \\
n = [1 \times 10^{18} \times \text{u.m}^{-3}, 1 \times 10^{18} \times \text{u.m}^{-3}] \\
f = \text{np.logspace(start=6, stop=11.3, num=3001)} \quad \# \text{1 MHz to 200 GHz} \\
\omega_{\text{RF}} = f \times (2 \times \text{np.pi}) \times (\text{u.rad} / \text{u.s})
\]

```python
help(cold_plasma_permittivity_SDP)
```

Out:

Help on function cold_plasma_permittivity_SDP in module plasmapy.formulary.dielectric:

cold_plasma_permittivity_SDP(B: ~astropy.units.Quantity, species, n, omega: ~astropy.units.Quantity)  
  Magnetized Cold Plasma Dielectric Permittivity Tensor Elements.

  Elements (S, D, P) are given in the "Stix" frame, ie. with B // z.

  The :math:`\exp(-i \omega t)` time-harmonic convention is assumed.

  Parameters
  ----------
  B : ~astropy.units.Quantity
      Magnetic field magnitude in units convertible to tesla.

  species : list of str
      List of the plasma particle species
      e.g.: ['e', 'D+] or ['e', 'D+', 'He+'].

  n : list of ~astropy.units.Quantity
      `list` of species density in units convertible to per cubic meter
      The order of the species densities should follow species.

  omega : ~astropy.units.Quantity
      Electromagnetic wave frequency in rad/s.

  Returns
  -------
  sum : ~astropy.units.Quantity
      S ("Sum") dielectric tensor element.

  difference : ~astropy.units.Quantity
      D ("Difference") dielectric tensor element.

  plasma : ~astropy.units.Quantity
      P ("Plasma") dielectric tensor element.

  Notes
  -----
The dielectric permittivity tensor is expressed in the Stix frame with the \( \exp(-i \omega t) \) time-harmonic convention as
\[
\mathbf{\varepsilon} = \mathbf{\varepsilon}_0 \begin{pmatrix} S & -i D & 0 \\ +i D & S & 0 \\ 0 & 0 & P \end{pmatrix}
\]

where:
\[
S = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega^2 - \Omega_{c,s}^2}
\]
\[
D = \sum_s \frac{\Omega_{c,s}}{\omega} \frac{\omega_{p,s}^2}{\omega^2 - \Omega_{c,s}^2}
\]
\[
P = 1 - \sum_s \frac{\omega_{p,s}^2}{\omega^2}
\]
where \( \omega_{p,s} \) is the plasma frequency and \( \Omega_{c,s} \) is the signed version of the cyclotron frequency for the species \( s \).

References
--------

Examples
-------
```python
>>> from astropy import units as u
>>> from numpy import pi
>>> B = 2*u.T
>>> species = ['e', 'D+']
>>> n = [1e18*u.m**-3, 1e18*u.m**-3]
>>> omega = 3.7e9*(2*pi)*(u.rad/u.s)
>>> permittivity = S, D, P = cold_plasma_permittivity_SDP(B, species, n, omega)
>>> S
<Quantity 1.02422...>
>>> permittivity.sum # namedtuple-style access
<Quantity 1.02422...>
>>> D
<Quantity 0.39089...>
>>> P
<Quantity -4.8903...>
```

Filter positive and negative values, for display purposes only. Still for display purposes, replace 0 by NaN to NOT plot 0 values
```python
S_pos = S * (S > 0)
D_pos = D * (D > 0)
P_pos = P * (P > 0)
S_neg = S * (S < 0)
D_neg = D * (D < 0)
```
\[ P_{\text{neg}} = P * (P < 0) \]
\[ S_{\text{pos}}[S_{\text{pos}} == 0] = \text{np.NaN} \]
\[ D_{\text{pos}}[D_{\text{pos}} == 0] = \text{np.NaN} \]
\[ P_{\text{pos}}[P_{\text{pos}} == 0] = \text{np.NaN} \]
\[ S_{\text{neg}}[S_{\text{neg}} == 0] = \text{np.NaN} \]
\[ D_{\text{neg}}[D_{\text{neg}} == 0] = \text{np.NaN} \]
\[ P_{\text{neg}}[P_{\text{neg}} == 0] = \text{np.NaN} \]

```python
plt.figure(figsize=(12, 6))
plt.semilogx(f, abs(S_pos),
            f, abs(D_pos),
            f, abs(P_pos), lw=2)
plt.semilogx(f, abs(S_neg), '#1f77b4',
            f, abs(D_neg), '#ff7f0e',
            f, abs(P_neg), '#2ca02c', lw=2, ls='--')
plt.yscale('log')
plt.grid(True, which='major')
plt.grid(True, which='minor')
plt.ylim(1e-4, 1e8)
plt.xlim(1e6, 200e9)
plt.legend(('S > 0', 'D > 0', 'P > 0', 'S < 0', 'D < 0', 'P < 0'),
            fontsize=16, ncol=2)
plt.xlabel('RF Frequency [Hz]', size=16)
plt.ylabel('Absolute value', size=16)
plt.tick_params(labelsize=14)
```

Cold Plasma tensor elements in the rotating basis

\[ L, R, P = \text{cold_plasma_permittivity}_\text{LRP}(B, \text{species}, n, \omega_\text{RF}) \]

\[ L_{\text{pos}} = L * (L > 0) \]
\[ R_{\text{pos}} = R * (R > 0) \]
\[ L_{\text{neg}} = L * (L < 0) \]
\[ R_{\text{neg}} = R * (R < 0) \]
L_pos[L_pos == 0] = np.NaN
R_pos[R_pos == 0] = np.NaN
L_neg[L_neg == 0] = np.NaN
R_neg[R_neg == 0] = np.NaN

plt.figure(figsize=(12, 6))
plt.semilogx(f, abs(L_pos),
    f, abs(R_pos),
    f, abs(P_pos), lw=2)
plt.semilogx(f, abs(L_neg), '#1f77b4',
    f, abs(R_neg), '#ff7f0e',
    f, abs(P_neg), '#2ca02c', lw=2, ls='--')
plt.yscale('log')
plt.grid(True, which='major')
plt.grid(True, which='minor')
plt.xlim(1e6, 200e9)
plt.legend(('L > 0', 'R > 0', 'P > 0', 'L < 0', 'R < 0', 'P < 0'),
    fontsize=16, ncol=2)
plt.xlabel('RF Frequency [Hz]', size=16)
plt.ylabel('Absolute value', size=16)
plt.tick_params(labelsize=14)

Checks if the values obtained are coherent. They should satisfy $S = (R+L)/2$ and $D = (R-L)/2$

```python
try:
    np.testing.assert_allclose(S, (R + L) / 2)
    np.testing.assert_allclose(D, (R - L) / 2)
except AssertionError as e:
    print(e)

# Checks for R=S+D and L=S-D
try:
    np.testing.assert_allclose(R, S + D)
    np.testing.assert_allclose(L, S - D)
except AssertionError as e:
    print(e)
```
Magnetostatic Fields

An example of using PlasmaPy's `Magnetostatic` class in the `physics` subpackage.

```python
from plasmapy.formulary import magnetostatics
from plasmapy.classes.sources import Plasma3D
import numpy as np
import astropy.units as u
import matplotlib.pyplot as plt

Some common magnetostatic fields can be generated and added to a plasma object. A dipole

```python
dipole = magnetostatics.MagneticDipole(np.array([0, 0, 1])*u.A*u.m*u.m, np.array([0, _→0, 0])*u.m)
print(dipole)
```

Out:

```
MagneticDipole(moment=[0. 0. 1.], p0=[0. 0. 0.])
```

initialize a a plasma, where the magnetic field will be calculated on

```python
plasma = Plasma3D(domain_x=np.linspace(-2, 2, 30) * u.m,
                  domain_y=np.linspace(0, 0, 1) * u.m,
                  domain_z=np.linspace(-2, 2, 20) * u.m)
add the dipole field to it

```python
plasma.add_magnetostatic(dipole)
```

```python
X, Z = plasma.grid[0, :, 0, :], plasma.grid[2, :, 0, :]
U = plasma.magnetic_field[0, :, 0, :].value.T  # because grid uses 'ij' indexing
W = plasma.magnetic_field[2, :, 0, :].value.T  # because grid uses 'ij' indexing
```

```python
plt.figure()
plt.axis('square')
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('Dipole field in x-z plane, generated by a dipole pointing in the z→direction')
plt.streamplot(plasma.x.value, plasma.z.value, U, W)
```
A circular current-carrying wire

```python
cw = magnetostatics.CircularWire(np.array([0, 0, 1]), np.array([0, 0, 0])*u.m, 1*u.m, -1*u.A)
print(cw)
```

Out:

```
CircularWire(normal=[0. 0. 1.], center=[0. 0. 0.], radius=1.0, current=1.0)
```

initialize a plasma, where the magnetic field will be calculated on

```python
plasma = Plasma3D(domain_x=np.linspace(-2, 2, 30) * u.m,
                   domain_y=np.linspace(0, 0, 1) * u.m,
                   domain_z=np.linspace(-2, 2, 20) * u.m)
```

add the circular coil field to it

```python
plasma.add_magnetostatic(cw)
```

X, Z = plasma.grid[0, :, 0, :], plasma.grid[2, :, 0, :]
U = plasma.magnetic_field[0, :, 0, :].value.T # because grid uses 'ij' indexing

(continues on next page)
W = plasma.magnetic_field[2, :, 0, :].value.T  # because grid uses 'ij' indexing

plt.figure()
plt.axis('square')
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.title('Circular coil field in x-z plane, generated by a circular coil in the x-y plane')
plt.streamplot(plasma.x.value, plasma.z.value, U, W)

Out:
<matplotlib.streamplot.StreamplotSet object at 0x7f77fec05240>

A circular wire can be described as parametric equation and converted to GeneralWire

gw_cw = cw.to_GeneralWire()

# the calculated magnetic field is close
print(gw_cw.magnetic_field([0, 0, 0]) - cw.magnetic_field([0, 0, 0]))

Out:
[ 0.00000000e+00  0.00000000e+00 -4.13416205e-12] T
A infinite straight wire

```python
iw = magnetostatics.InfiniteStraightWire(np.array([0, 1, 0]), np.array([0, 0, 0])*u.m, 1*u.A)
print(iw)
```

Out:

```
InfiniteStraightWire(direction=[0. 1. 0.], p0=[0. 0. 0.], current=1.0)
```

initialize a a plasma, where the magnetic field will be calculated on

```python
plasma = Plasma3D(domain_x=np.linspace(-2, 2, 30) * u.m,  
                 domain_y=np.linspace(0, 0, 1) * u.m,  
                 domain_z=np.linspace(-2, 2, 20) * u.m)

# add the infinite straight wire field to it
plasma.add_magnetostatic(iw)

X, Z = plasma.grid[0, :, 0, :], plasma.grid[2, :, 0, :]
U = plasma.magnetic_field[0, :, 0, :].value.T  # because grid uses 'ij' indexing
W = plasma.magnetic_field[2, :, 0, :].value.T  # because grid uses 'ij' indexing

plt.figure()
plt.title('Dipole field in x-z plane, generated by a infinite straight wire '  
         'pointing in the y direction')
plt.axis('square')
plt.xlim(-2, 2)
plt.ylim(-2, 2)
plt.streamplot(plasma.x.value, plasma.z.value, U, W)
```
Pole field in x-z plane, generated by an infinite straight wire pointing in the y dir

Out:

<matplotlib.streamplot.StreamplotSet object at 0x7f77fe8320b8>

Total running time of the script: 0 minutes 1.880 seconds
The PlasmaPy Development Guide contains information on how to contribute to PlasmaPy, along with guidelines for code, testing, and documentation.

3.1 PlasmaPy Development Guide

3.1.1 Installing PlasmaPy for Development

Obtaining PlasmaPy source code

After creating your GitHub account, go to the PlasmaPy repository on GitHub and fork a copy of PlasmaPy to your account.

To access Git commands on Windows, try Git Bash.

Next you must clone your fork to your computer. Go to the directory that will host your PlasmaPy directory, and run one of the following commands (after changing your-username to your username). If you would like to use HTTPS (which is the default and easier to set up), then run:

```
git clone https://github.com/your-username/PlasmaPy.git
```

SSH is a more secure option, but requires you to set up an SSH key beforehand. The equivalent SSH command is:

```
git clone git@github.com:your-username/PlasmaPy.git
```

After cloning, we must tell git where the development version of PlasmaPy is by running:

```
cd PlasmaPy
git remote add upstream git://github.com/PlasmaPy/PlasmaPy.git
```

To check on which remotes exist, run `git remote -v`. You should get something like this:
Setting up an environment for development

Setup procedures for the two most popular virtual environments, conda and virtualenv, are listed below.

**Conda**

To set up a development environment for PlasmaPy, we strongly recommend the Anaconda distribution.

**Activate Anaconda**

After installing Anaconda, launch any conda environment. By default, conda installs a root environment, which you should be able to activate via

```
source /home/user/anaconda3/bin/activate root
```

where /home/user/anaconda3/ can be swapped to wherever your anaconda installation resides.

On newer versions of Anaconda the recommended activation process has changed to:

```
. /home/user/anaconda3/etc/profile.d/conda.sh
conda activate
```

**Note:** On Windows, the way to do this is via running Anaconda Prompt from the Start Menu. Git Bash may also work if you have added Anaconda to PATH.

**Create your environment**

Having activated Anaconda, enter PlasmaPy’s repository root directory and create an environment with our suggested packages by executing the following:

```
conda env create -f requirements/environment.yml
```

You may now enter the environment via

```
source activate plasmapy
```

**Note:** On Windows, skip the source part of the previous command.

In newer Conda versions, the command to run is

```
conda activate plasmapy
```
Virtualenv

Create a directory for holding the PlasmaPy repository, move into it and create the virtual environment

```
virtualenv -p python3 .
```

You may need to make sure that this directory’s path doesn’t contain any spaces, otherwise virtualenv may throw an error.

Your virtual environment should now be created. If you run `ls` you will notice that virtualenv has created a number of subdirectories: `bin/`, `lib/`, and `include/`. This is why we’re not creating the virtualenv within the repository itself - so as to not pollute it. To activate the virtualenv you will run:

```
source ./bin/activate
```

You should now see that your shell session is prepended with (plasmapy), like so:

```
(plasmapy) user@name:~/programming/plasmapy$
```

This indicates that the virtualenv is running. Congratulations! When you’re done working on PlasmaPy, you can deactivate the virtualenv by running:

```
source deactivate
```

Now that you have plasmapy on your local computer and you have a virtual environment, you will want to “install” this development version of PlasmaPy along with its dependencies. Start by activating your virtual environment. Next you want install the PlasmaPy dependencies. One way to do this is to do

```
(plasmapy) user@name:~/programming/plasmapy$ pip install -r requirements/environment.txt
```

Next, setup the development version of PlasmaPy which you just cloned by moving into the root directory of the cloned repo and running the setup.py script there:

```
(plasmapy) user@name:~/programming/plasmapy/PlasmaPy$ pip install -e .
```

You should now be all set to run development versions of PlasmaPy modules via `import PlasmaPy` in your test scripts!

Running anaconda with virtualenv

If you are running the Anaconda suite and want to use virtualenv to setup your virtual environment, you will have to let the system know where the Python interpreter can be found. On Linux this is done with (for example, assuming having installed Anaconda into `~/anaconda3`):

```
export LD_LIBRARY_PATH="$HOME/anaconda3/lib/
```

Exporting the library path to the dynamic linker will only last for the duration of the current shell session.

You will have to add the python library directory to `LD_LIBRARY_PATH`, as described in a previous step, prior to activating the virtualenv for every new shell session.

Installing your own dev version

To be able to import PlasmaPy from your source version, enter the repository root and use one of
python setup.py develop
pip install -e .

Note: If you are not working within a virtual environment, this may end in a permission error - this can be avoided via also adding the --user flag. But seriously, use a virtual environment and spare yourself the trouble.

Either one of these commands will create a soft link to your cloned repository. Any changes in Python code you make there will be there when you import plasmapy from an interactive session.

### 3.1.2 Code Development Guidelines

This document describes the coding requirements and guidelines to be followed during the development of PlasmaPy and affiliated packages.

Code written for PlasmaPy must be compatible with Python 3.6 and later.

#### Coding Style

- PlasmaPy follows the PEP8 Style Guide for Python Code. This style choice helps ensure that the code will be consistent and readable.
  - The PEP 8 Speaks integration on GitHub will comment when there are any departures from the PEP 8 style guide.
  - PEP 8 compliance may be checked locally using pycodestyle.
  - Line lengths should be chosen to maximize the readability and elegance of the code. The maximum line length for Python code in PlasmaPy is 99 characters.
  - Docstrings and comments should generally be limited to 72 characters.
- Follow the existing coding style within a subpackage.
- Use standard abbreviations for imported packages when possible, such as import numpy as np, import matplotlib as mpl, import matplotlib.pyplot as plt, and import astropy.units as u.
- __init__.py files for modules should not contain any significant implementation code, but it can contain a docstring describing the module and code related to importing the module. Any substantial functionality should be put into a separate file.
- Use absolute imports, such as from plasmapy.mathematics import Fermi_integral, rather than relative imports such as from ..mathematics import Fermi_integral.
- For multiline imports, instead of using \ to break lines, wrap the imported names in () parentheses and use consistent whitespace.
- Use Optional[type] for type hinted keyword arguments with a default value of None.
- There should be at most one pun per 1284 lines of code.

#### Branches, commits, and pull requests

Before making any changes, it is prudent to update your local repository with the most recent changes from the development repository:
git fetch upstream

Changes to PlasmaPy should be made using branches. It is usually best to avoid making changes on your master branch so that it can be kept consistent with the upstream repository. Instead we can create a new branch for the specific feature that you would like to work on:

git branch *your-new-feature*

Descriptive branch names such as grad-shafranov or adding-eigenfunction-poetry are helpful, while vague names like edits are considered harmful. After creating your branch locally, let your fork of PlasmaPy know about it by running:

git push --set-upstream origin *your-new-feature*

It is also useful to configure git so that only the branch you are working on gets pushed to GitHub:

git config --global push.default simple

Once you have set up your fork and created a branch, you are ready to make edits to PlasmaPy. Switch to your new branch by running:

git checkout *your-new-feature*

Go ahead and modify files with your favorite text editor. Be sure to include tests and documentation with any new functionality. We recommend reading about best practices for scientific computing. PlasmaPy uses the PEP 8 style guide for Python code and the numpydoc format for docstrings to maintain consistency and readability. New contributors should not worry too much about precisely matching these styles when first submitting a pull request, as the PEP8 Speaks GitHub integration will check pull requests for PEP 8 compatibility, and further changes to the style can be suggested during code review.

You may periodically commit changes to your branch by running

```bash
    git add filename.py
    git commit -m "*brief description of changes*"
```

Committed changes may be pushed to the corresponding branch on your GitHub fork of PlasmaPy using

git push origin *your-new-feature*

or, more simply,

git push

Once you have completed your changes and pushed them to the branch on GitHub, you are ready to make a pull request. Go to your fork of PlasmaPy in GitHub. Select “Compare and pull request”. Add a descriptive title and some details about your changes. Then select “Create pull request”. Other contributors will then have a chance to review the code and offer constructive suggestions. You can continue to edit the pull request by changing the corresponding branch on your PlasmaPy fork on GitHub. After a pull request is merged into the code, you may delete the branch you created for that pull request.

**Commit Messages**

Good commit messages communicate context and intention to other developers and to our future selves. They provide insight into why we chose a particular implementation, and help us avoid past mistakes.

Suggestions on how to write a git commit message:
• Separate subject from body with a blank line
• Limit the subject line to 50 characters
• Capitalize the subject line
• Do not end the subject line with a period
• Use the imperative mood in the subject line
• Wrap the body at 72 characters
• Use the body to explain what and why vs. how

Documentation

• All public classes, methods, and functions should have docstrings using the numpydoc format.
• Docstrings may be checked locally using pydocstyle.
• These docstrings should include usage examples.

Warnings and Exceptions

• Debugging can be intensely frustrating when problems arise and the associated error messages do not provide useful information on the source of the problem. Warnings and error messages must be helpful enough for new users to quickly understand any problems that arise.
• “Errors should never pass silently.” Users should be notified when problems arise by either issuing a warning or raising an exception.
• The exceptions raised by a method should be described in the method’s docstring. Documenting exceptions makes it easier for future developers to plan exception handling.

Units

• Code within PlasmaPy must use SI units to minimize the chance of ambiguity, and for consistency with the recognized international standard. Physical formulae and expressions should be in base SI units.
  – Functions should not accept floats when an Astropy Quantity is expected. In particular, functions should not accept floats and make the assumption that the value will be in SI units.
  – A common convention among plasma physicists is to use electron-volts (eV) as a unit of temperature. Strictly speaking, this unit corresponds not to temperature but is rather a measure of the thermal energy per particle. Code within PlasmaPy must use the kelvin (K) as the unit of temperature to avoid unnecessary ambiguity.
• PlasmaPy uses the astropy.units package to give physical units to values.
  – All units packages available in Python presently have some limitations, including incompatibility with some NumPy and SciPy functions. These limitations are due to issues within NumPy itself. Many of these limitations are being resolved, but require upstream fixes.
• Dimensionless units may be used when appropriate, such as for certain numerical simulations. The conventions and normalizations should be clearly described in docstrings.
Equations and Physical Formulae

- If a quantity has several names, then the function name should be the one that provides the most physical insight into what the quantity represents. For example, *gyrofrequency* indicates gyration, whereas *Larmor_frequency* indicates that this frequency is somehow related to someone named Larmor. Similarly, using *omega_ce* as a function name will make the code less readable to people who are unfamiliar with this particular notation.

- Physical formulae should be inputted without first evaluating all of the physical constants. For example, the following line of code obscures information about the physics being represented:

  ```python
  >>> omega_ce = 1.76e7*(B/u.G)*u.rad/u.s  # doctest: +SKIP
  ```

  In contrast, the following line of code shows the exact formula which makes the code much more readable.

  ```python
  >>> omega_ce = (e * B) / (m_e * c)  # doctest: +SKIP
  ```

  The origins of numerical coefficients in formulae should be documented.

- Docstrings should describe the physics associated with these quantities in ways that are understandable to students who are taking their first course in plasma physics while still being useful to experienced plasma physicists.

- SI units that were named after a person should not be capitalized except at the beginning of a sentence.

- Some plasma parameters depend on more than one quantity with the same units. In the following line, it is difficult to discern which is the electron temperature and which is the ion temperature.

  ```python
  >>> ion_sound_speed(1e6*u.K, 2e6*u.K)  # doctest: +SKIP
  ```

  Remembering that “explicit is better than implicit”, it is more readable and less prone to errors to write:

  ```python
  >>> ion_sound_speed(T_i=1e6*u.K, T_e=2e6*u.K)  # doctest: +SKIP
  ```

- SI units that were named after a person should be lower case except at the beginning of a sentence, even if their symbol is capitalized. For example, kelvin is a unit while Kelvin was a scientist.

Angular Frequencies

Unit conversions involving angles must be treated with care. Angles are dimensionless but do have units. Angular velocity is often given in units of radians per second, though dimensionally this is equivalent to inverse seconds. Astropy will treat radians dimensionlessly when using the *dimensionless_angles* equivalency, but *dimensionless_angles* does not account for the multiplicative factor of 2*pi that is used when converting between frequency (1 / s) and angular frequency (rad / s). An explicit way to do this conversion is to set up an equivalency between cycles/s and Hz:

```python
>>> from astropy import units as u
>>> f_ce = omega_ce.to(u.Hz, equivalencies=[(u.cy/u.s, u.Hz)])  # doctest: +SKIP
```

However, *dimensionless_angles* does work when dividing a velocity by an angular frequency to get a length scale:

```python
>>> d_i = (c/omega_pi).to(u.m, equivalencies=u.dimensionless_angles{})  # doctest: +SKIP
```
3.1.3 Testing Guidelines

Motivation

Tests are vital for software reliability and maintainability. Writing tests requires additional effort now, but saves considerable time in the long run. Tests enable us to modify code and quickly discover when we introduce errors\(^1\). Tests also provide future contributors with examples of how functions and classes were originally intended to be used.

Tests should be readable and maintainable. Well-written tests are easier to understand and modify when the behavior of a function or method is changed. Consequently, tests should be held to the same code quality standards as the rest of the package.

When bugs are discovered, they should be turned into test cases to prevent the bug from emerging again in the future\(^2\).

Overview

Pull requests that create or change functionality must include tests and documentation before being merged.

PlasmaPy uses `pytest <https://docs.pytest.org>`\(^{\dagger}\) for software testing. The test suite may be run locally or automatically via pull requests on GitHub. PlasmaPy undergoes continuous integration testing of the code base by Travis CI and AppVeyor, including code examples in docstrings. Codecov performs test coverage checks and shows whether or not each line of code is run during the test suite. CircleCI tests that the documentation can be successfully built. The results of the documentation test builds are displayed using Giles. PlasmaPy’s test suite is automatically run whenever a pull request to the main repository is made or updated.

Running Tests

Running tests on GitHub

The recommended way to run PlasmaPy’s full test suite when contributing code is to create a pull request from your development branch to PlasmaPy’s GitHub repository. The test suite will be run when the pull request is created and every time your development branch is subsequently updated.

Travis CI and AppVeyor run code tests and check that code examples in docstrings produce the expected output. Travis CI runs the tests in a Linux/MacOS environment whereas AppVeyor runs the tests in a Windows environment.

The results from Travis CI are used to generate test coverage reports which are displayed by Codecov. These reports show which lines of code are covered by tests and which are not, and allow us to write targeted tests to fill in the gaps in test coverage. The results displayed by Codecov will be marked as passing when the code coverage is sufficiently high.

Circle CI performs a test build of the documentation in both HTML and LaTeX formats, and reports any errors that arise.

If any inconsistencies with the PEP 8 style guide are found, then pep8speaks will comment on the pull request and update that comment as the pull request is updated.

---

\(^1\) In *Working Effectively With Legacy Code*, Michael Feathers bluntly writes: “Code without tests is bad code. It doesn’t matter how well written it is; it doesn’t matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don’t know if our code is getting better or worse.”

\(^2\) In the chapter “Bugs Are Missing Tests” in *Beyond Legacy Code*, David Bernstein writes: “Every bug exists because of a missing test in a system. The way to fix bugs using TDD [test-driven development] is first write a failing test that represents the bug and then fix the bug and watch the failing test turn green.”
Running tests from the command line

The recommended method for running the test suite locally on your computer is running

```
python setup.py test
```

in the repository’s root directory. This command will run all of the tests and verify that examples in docstrings produce the expected output. This command (which was enabled by integrating pytest with setuptools) ensures that the package is set up. These tests should be run in a Python environment in which PlasmaPy has not already been installed.

Command line options for pytest may be passed using the `-a` flag. For example, if you want to stop pytest after two test failures, return short traceback reports, and run tests only if the test path contains `plasma` and not `blob`, then run

```
python setup.py test -a "--maxfail=2 --tb=short -k 'plasma and not blob'"
```

One may also run pytest from the command line.

Some tests in the test suite can take a long time to run, which can slow down development. These tests can be identified with the pytest annotation `@pytest.mark.slow`. To skip these tests, execute `pytest -m 'not slow'`. To exclusively test the slow tests, execute `pytest -m slow`.

Running tests within Python

After installing PlasmaPy by running `pip install plasmapy` or `python setup.py install`, then PlasmaPy’s test suite may be run using

```
>>> import plasmapy
>>> plasmapy.test()
```

Writing Tests

Pull requests must include tests of new or changed functionality before being merged.

Best practices for writing tests

The following guidelines are helpful suggestions for writing readable, maintainable, and robust tests.

- Each function and method should have unit tests that check that it returns the expected results, issues the appropriate warnings, and raises the appropriate exceptions.
- Bugs should be turned into test cases.
- Tests are run frequently during code development, and slow tests may interrupt the flow of a contributor. Tests should be minimal, sufficient enough to be complete, and as efficient as possible.
- Slow tests can be annotated with `@pytest.mark.slow` when they cannot be made more efficient.

Test organization and collection

Pytest has certain test discovery conventions that are used to collect the tests to be run.
The tests for each subpackage are contained in a `tests` subfolder. For example, the tests for `atomic` are located in `plasmapy/atomic/tests`. Test files should begin with `test_` and generally contain the name of the module or object that is being tested.

The functions that are to be tested in each test file should likewise be prepended with `test_` (e.g., `test_atomic.py`). Tests may also be grouped into classes. In order for pytest to find tests in classes, the class name should start with `Test` and the methods to be run as tests should start with `test_`. For example, `test_particle_class.py` could define the `TestParticle` class containing the method `test_integer_charge`.

### Assert statements

- Pytest often runs tests by checking `assert` statements.

```python
def test_addition():
    assert 2 + 2 == 4
```

When `assert` statements raise an `AssertionError`, pytest will display the values of the expressions evaluated in the `assert` statement. The automatic output from pytest is sufficient for simple tests as above. For more complex tests, we can add a descriptive error message to provide context that can help us pinpoint the causes of test failures more quickly.

```python
def test_addition():
    assert 2 + 2 == 4, "Addition is broken. Reinstall universe and reboot."
```

To make the error statement easier to read, the values of variables can be included in the error message by using f-strings.

```python
def test_addition():
    result = 2 + 2
    expected = 4
    assert result == expected, f"2 + 2 returns {result} instead of {expected}."
```

### Floating point comparisons

Comparisons between floating point numbers with `==` is fraught with peril because of limited precision and rounding errors. Moreover, the values of fundamental constants in `astropy.constants` are occasionally refined as improvements become available.

Using `numpy.isclose` when comparing floating point numbers and `astropy.units.isclose` for `astropy.units.Quantity` instances lets us avoid these difficulties. The `rtol` keyword for each of these functions allows us to set an acceptable relative tolerance. Ideally, `rtol` should be set to be an order of magnitude or two greater than the expected uncertainty. For mathematical functions, a value of `rtol=1e-14` may be appropriate. For quantities that depend on physical constants, a value between `rtol=1e-8` and `rtol=1e-5` may be required, depending on how much the accepted values for fundamental constants are likely to change. For comparing arrays, `numpy.allclose` and `astropy.units.allclose` should be used instead.

### Testing warnings and exceptions

Robust testing frameworks should test that functions and methods return the expected results, issue the expected warnings, and raise the expected exceptions. Pytest contains functionality to `test warnings` and `test exceptions`.

To test that a function issues an appropriate warning, use `pytest.warns`.
import pytest
import warnings

def issue_warning():
    warnings.warn("Beware the ides of March", UserWarning)

def test_issue_warning():
    with pytest.warns(UserWarning):
        issue_warning()

To test that a function raises an appropriate exception, use pytest.raises.

def raise_exception():
    raise Exception

def test_raise_exception():
    with pytest.raises(Exception):
        raise_exception()
        pytest.fail("Exception not raised.")

Test independence and parametrization

In this section, we’ll discuss the issue of parametrization based on an example of a proof of Gauss’s class number conjecture.

The proof goes along these lines:

- If the generalized Riemann hypothesis is true, the conjecture is true.
- If the generalized Riemann hypothesis is false, the conjecture is also true.
- Therefore, the conjecture is true.

One way to use pytest would be to write sequential test in a single function.

def test_proof_by_riemann_hypothesis():
    assert proof_by_riemann(False)
    assert proof_by_riemann(True)  # only run if previous test passes

If the first test were to fail, then the second test will never be run. We would therefore not know the potentially useful results of the second test. This drawback can be avoided by making independent tests that will both be run.

def test_proof_if_riemann_false():
    assert proof_by_riemann(False)

def test_proof_if_riemann_true():
    assert proof_by_riemann(True)

However, this approach can lead to cumbersome, repeated code if you are calling the same function over and over. If you wish to run multiple tests for the same function, the preferred method is to use pytest’s parametrization capabilities.

@ pytest.mark.parametrize("truth_value", [True, False])
def test_proof_if_riemann(truth_value):
    assert proof_by_riemann(truth_value)

This code snippet will run proof_by_riemann(truth_value) for each truth_value in truth_values_to_test. Both of the above tests will be run regardless of failures. This approach is
much cleaner for long lists of arguments, and has the advantage that you would only need to change the function call
in one place if something changes.

With qualitatively different tests you would use either separate functions or pass in tuples containing inputs and ex-
pected values.

```python
@pytest.mark.parametrize("truth_value, expected", [(True, True), (False, True)])
def test_proof_if_riemann(truth_value, expected):
    assert proof_by_riemann(truth_value) == expected
```

**Pytest helpers**

A robust testing framework should test not just that functions and methods return the expected results, but also that
they issue the expected warnings and raise the expected exceptions. In PlasmaPy, tests often need to compare a float
against a float, an array against an array, and Quantity objects against other Quantity objects to within
a certain tolerance. Occasionally tests will be needed to make sure that a function will return the same value for
different arguments (e.g., due to symmetry properties). PlasmaPy’s utils subpackage contains the run_test and
run_test_equivalent_calls helper functions that can generically perform many of these comparisons and
checks.

The run_test function can be used to check that a callable object returns the expected result, raises the expected
exception, or issues the expected warning for different positional and keyword arguments. This function is particularly
useful when unit testing straightforward functions when you have a bunch of inputs and know the expected result.

Suppose that we want to test the trigonometric property that

\[
\sin(\theta) = \cos(\theta + \frac{\pi}{2}),
\]

We may use run_test as in the following example to check the case of \(\theta \equiv 0\).

```python
from numpy import sin, cos, pi
from plasmapy.utils.pytest_helpers import run_test
def test_trigonometric_properties():
    run_test(func=sin, args=0, expected_outcome=cos(pi/2), atol=1e-16)
```

We may use pytest.mark.parametrize with run_test to check multiple cases. If run_test only receives
one positional argument that is a list or tuple, then it will assume that list or tuple contains the callable,
the positional arguments, the keyword arguments (which may be omitted), and the expected outcome (which may be
the returned object, a warning, or an exception).

```python
@pytest.mark.parametrize("input_tuple", [(sin, 0, cos(pi/2)), (sin, '.', TypeError)])
def test_trigonometry(input_tuple):
    run_test(input_tuple, atol=1e-16)
```

This parametrized function will check that \(\sin(0)\) is within 1e-16 of \(\cos(\pi/2)\) and that \(\sin('.')\) raises a
TypeError.

We may use run_test_equivalent_calls to check symmetry properties such as

\[
\cos(\theta) = \cos(-\theta),
\]

This property can be checked for \(\theta = 1\) with the following code.

```python
def test_cosine_symmetry():
    """Test that \cos(\theta) equals \cos(-\theta).""
    plasmapy.utils.run_test_equivalent_calls(cos, 1, -1)
```
We may also use `pytest.mark.parametrize` with `run_test_equivalent_calls` to sequentially test multiple symmetry properties.

```python
@ pytest.mark.parametrize('input_tuple', [(cos, 1, -1), ([cos, pi/2], [sin, 0])])
def test_symmetry_properties(input_tuple):
    plasmapy.utils.run_test_equivalent_calls(input_tuple, atol=1e-16)
```

This parametrized function will check that $\cos(1)$ is within $1e-16$ of $\cos(-1)$, and that $\cos(pi/2)$ is within $1e-16$ of $\sin(0)$.

Please refer to the documentation for `run_test` and `run_test_equivalent_calls` to learn about the full capabilities of these pytest helper functions (including for testing functions that return `Quantity` objects).

**Warning:** The API within `pytest_helpers` is not yet stable and may change in the near future.

**Fixtures**

Fixtures provide a way to set up well-defined states in order to have consistent tests. We recommend using fixtures for complex tests that would be unwieldy to set up with parametrization as described above.

**Code Coverage**

PlasmaPy uses Codecov to show what lines of code are covered by the test suite and which lines are not. At the end of every Travis CI testing session, information on which lines were executed is sent to Codecov. Codecov comments on the pull request on GitHub with a coverage report.

**Test coverage of contributed code**

Code contributions to PlasmaPy are required to be well-tested. A good practice is for new code to have a test coverage percentage of at least about the current code coverage. Tests must be provided in the original pull request, because often a delayed test ends up being a test not written. There is no strict cutoff percentage for how high the code coverage must be in order to be acceptable, and it is not always necessary to cover every line of code. For example, it is often helpful for methods that raise a `NotImplementedError` to be marked as untested as a reminder of unfinished work.

Occasionally there will be some lines that do not require testing. For example, testing exception handling for an `ImportError` when importing an external package would usually be impractical. In these instances, we may end a line with `# coverage: ignore` to indicate that these lines should be excluded from coverage reports (or add a line to `.coveragerc`). This strategy should be used sparingly, since it is often better to explicitly test exceptions and warnings and to show the lines of code that are not tested.

**Generating coverage reports locally**

Coverage reports may be generated on your local computer by running

```
python setup.py test --coverage
coverage html
```

The coverage reports may be accessed by opening the newly generated `htmlcov/index.html` in your favorite web browser. These commands require the `pytest` and `coverage` packages to be installed.
Ignoring lines in coverage tests

Occasionally there will be lines of code that do not require tests. For example, it would be impractical to test that an `ImportError` is raised when running `import plasmapy` from Python 2.7.

To ignore a line of code in coverage tests, append it with `# coverage: ignore`. If this comment is used on a line with a control flow structure (e.g., `if`, `for`, and `while`) that begins a block of code, then all lines in that block of code will be ignored. In the following example, lines 3 and 4 will be ignored in coverage tests.

```python
try:
    import numpy
except ModuleNotFoundError as exc:
    # coverage: ignore
    raise RuntimeError from exc
```

The `.coveragerc` file is used to specify lines of code and files that should always be ignored in coverage tests.

**Note:** In general, untested lines of code should remain marked as untested to give future developers a better idea of where tests should be added in the future and where potential bugs may exist.

Footnotes

### 3.1.4 Documentation Guidelines

This document describes the documentation requirements and guidelines to be followed during the development of PlasmaPy and affiliated packages.

**Building documentation**

Documentation is built from the master branch on every commit pushed to it.

**Using sphinx within the project**

To build docs locally, either:

- use `Tox` with `tox -e build_docs` from within the main PlasmaPy repository directory, or
- enter the `docs` directory and run `make html`.

Afterwards, open `docs/_build/index.html` with your browser of choice.

Do try to solve warnings in documentation when writing your code.

**Docstrings**

- All public classes, methods, and functions should have docstrings.
- PlasmaPy uses the `numpydoc` standard for docstrings.
- Docstrings must be raw string literals if they contain backslashes. A raw string literal is denoted by having an `r` immediately precede quotes or triple quotes:
""" I did not like unstable eigenfunctions at first, but then they
grew on me.
"""

- Simple functions may need only a one-line docstring.

**Narrative Documentation**

- Each subpackage must have narrative documentation describing its use.

### 3.1.5 Release Guide

This document describes the procedure for making a release of PlasmaPy. This document is under development and should be updated during all releases.

The following is a partial list of tasks to be performed for each release. This list is currently under development. Developers should expand the instructions while performing each release, and may refer to Astropy's release procedures for guidance.

**Release**

- Create a new branch for the release that is separate from the master branch, e.g. v0.3.x
- Check that the Continuous Integration is passing for the correct version (see the latest commit on master). You can use `hub ci-status master` with the `hub` CLI tool.
- Turn changelog entries into a CHANGELOG.rst file via `towncrier --version v0.3.0` or equivalent. When asked about removing changelog entries, do so. Ensure the entries are in proper categories.
- Move the generated CHANGELOG.rst file into `docs/whatsnew/{version_number}.rst`. Add the corresponding entry in the table of contents in `docs/whatsnew/index.rst`.
- Add the note on include new contributors. To do this efficiently, borrow the SunPy Xonsh script `generate_releaserst.xsh` and `generate_releaserst.xsh` with `--auth --project-name=plasmapy --pretty-project-name=PlasmaPy`.
- Use `git shortlog -nse | cut -f 2 | vim .mailmap`
- Use `astropy-tools/author_lists.py` for `credits.rst`.

I would think about limiting this to the credits in new release entries in `docs/whatsnew` due to maintenance burden.

~Dominik

- Commit your changes up until now
- Make sure that tests pass and that documentation builds without issue (`tox`)
- Tag the new version with `git tag -s v<version> -m "Version v<version>"
  - Note that `-s` signs the commit with a GPG key
- Push the tagged commit to the version’s branch on GitHub: `git push --force --follow-tags upstream v0.3.x`

At this point, the OpenAstronomy Azure Pipelines infrastructure should do most of the work for you! Ensure that the pipeline goes through.
Post-release

• If necessary (for MINOR+ and not for BUGFIX versions) activate the new branch’s version on RTD.
• Update the stable branch on GitHub.
• Make the release on conda-forge
• Reserve a digital object identifier on Zenodo
• Update code metadata in codemeta.json
  – The Codemeta standard is relatively new, so check the standard for terms that have changed and new terms that may apply
• Upload the release to the Zenodo record corresponding to the reserved DOI
• Notify plasma physics communities about the release
• Post release announcement on social media sites
• Send release announcement to mailing list
• Update the release guide to reflect any changes

Compatibility with Prior Versions of Python, NumPy, and Astropy

PlasmaPy releases will generally abide by the following standards, which are adapted from NumPy Enhancement Proposal 29 for the support of old versions of Python, NumPy, and Astropy.

• PlasmaPy should support at least the minor versions of Python initially released 42 months prior to a planned project release date.
• PlasmaPy should support at least the 2 latest minor versions of Python.
• PlasmaPy should support minor versions of NumPy initially released in the 24 months prior to a planned project release date or the oldest version that supports the minimum Python version (whichever is higher).
• PlasmaPy should support at least the 3 latest minor versions of NumPy and Astropy.

The required major and minor version numbers of upstream packages may only change during major or minor releases of PlasmaPy, and never during patch releases.

Exceptions to these guidelines should only be made when there are major improvements or fixes to upstream functionality or when other required packages have stricter requirements.
4.1 About PlasmaPy

4.1.1 Authors and Credits

PlasmaPy Coordinating Committee

- Drew Leonard
- Nick Murphy
- Tulasi Parashar
- Dominik Stańczak

PlasmaPy Contributors

The people in the following list have contributed to PlasmaPy. Included in parentheses are ORCID author identifiers.

- Jasper Beckers
- Manas Bedmutha
- Justin Bergeron
- Ludovico Bessi
- BH4
- Sean Carroll
- Sean Chambers
- Apoorv Choubey
- cclauss
- Jacob Deal
• Leah Einhorn
• Erik Everson
• Thomas Fan
• Samatayah I. Farid
• Michael Fischer
• Brian Goodall
• Graham Goudeau
• Silvina Guidoni
• Colby Haggerty
• Julien Hillairet (0000-0002-1073-6383)
• Poh Zi How
• Yi-Min Huang (0000-0002-4237-2211)
• Nabil Humphrey
• Maria Isupova
• Pawel Kozlowski (0000-0001-6849-3612)
• Siddharth Kulshrestha
• Piotr Kuszaj
• Samuel Langendorf (0000-0002-7757-5879)
• Drew Leonard (0000-0001-5270-7487)
• lgoenner
• Ritiek Malhotra
• Stuart Mumford (0000-0003-4217-4642)
• Joshua Munn
• Nick Murphy (0000-0001-6628-8033)
• Nismirno
• nrb1324
• Tulasi Parashar (0000-0003-0602-8381)
• Neil Patel
• Francisco Silva Pavon
• Roberto Díaz Pérez
• Raajit Raj
• Antonia Savcheva (0000-0002-5598-046X)
• Chengcai Shen (0000-0002-9258-4490)
• Dawa Nurbu Sherpa
• Angad Singh
• Ankit Singh
This list contains contributors to PlasmaPy’s core package and vision statement, including a few people who do not show up as PlasmaPy contributors on GitHub. If you made a contribution that was merged and your name is missing from the list, your information is incorrect, or you do not wish to be listed, then please submit a pull request or email namurphy@cfa.harvard.edu.

Other Credits

The PlasmaPy Community thanks the SunPy and Astropy communities for inspiring this project in the first place, providing much helpful advice, and showing examples of how to build a community-wide open source scientific software package.

Acknowledgements

Early development on PlasmaPy was partially supported by the U.S. Department of Energy through grant DE-SC0016363 that was funded through the NSF-DOE Partnership on Basic Plasma Science and Engineering; a Scholarly Studies grant awarded by the Smithsonian Institution; and Google Summer of Code. PlasmaPy is being developed with support from the U.S. National Science Foundation through grants 1931388, 1931393, 1931429, and 1931435 that were awarded through a collaborative proposal submitted to the Cyberinfrastructure for Sustained Scientific Innovation (CSSI) program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any of the funding agencies or organizations that have supported PlasmaPy development.

4.1.2 Stability of Subpackages

This document summarizes the stability of PlasmaPy’s subpackages so that users understand where they might expect changes in the future, and which subpackages they can safely use for production code. Please note that backward compatibility is not guaranteed for the 0.*.* series of development releases. Starting with version 1.0.0, the major version number will be incremented when a release contains backward incompatible changes.

The classification is as follows:

PlasmaPy’s planned and existing subpackages are:

4.1.3 Change Log

This document lists the changes made during each release of PlasmaPy, including bug fixes and changes to the application programming interface (API).
Plasmapy v0.3.1 (2020-02-01)

Bug Fixes

- Fix various packaging issues encountered during the v0.3.1 release.

Plasmapy v0.3.0 (2020-01-25)

Backwards Incompatible Changes

- Create simulation subpackage; move Species particle tracker there; rename to particletracker (#665)
- Changed `plasmapy.classes.Species` to `plasmapy.simulation.ParticleTracker` (#668)
- Move pytest helper functionality from `plasmapy.utils` to `pytest_helpers` (#674)
- Move `plasmapy.physics`, `plasmapy.mathematics` and `plasmapy.transport` into the common `plasmapy.formulary` subpackage (#692)
- Change `ClassicalTransport` methods into attributes (#705)

Deprecations and Removals

- Remove `parameters_cython.pyx`, switching to Numba for the future of computationally intensive code in PlasmaPy (#650)
- Remove `plasmapy.constants`, which was a thin wrapper around astropy.constants with no added value (#651)

Features

- Generalize `ion_sound_speed` function to work for all values of $k^2 \lambda_p^2$ (i.e. not just in the non-dispersive limit). (#700)
- Optimize `add__magnetostatics` for a 16x speedup in tests! (#703)

Bug Fixes

- Define `preserve_signature` decorator to help IDEs parse signatures of decorated functions. (#640)
- Fix Pytest deprecations of `message` argument to `raise` and `warn` functions. (#666)
- Fix `h5py` warning in OpenPMD module, opening files in read mode by default (#717)

Improved Documentation

- Added real-world examples to examples/plot_physics.py and adjusted the plots to be more human-friendly. (#448)
- Add examples images to the top of the main doc page in `docsindex.rst` (#655)
- Added examples to the documentation to `mass_density` and Hall_parameter functions (#709)
- Add docstrings to decorator `plasmapy.utils.decorators.converter.angular_freq_to_hz()` (#729)
Trivial/Internal Changes

- Replace decorator `plasmapy.utils.decorators.checks.check_quantity()` with decorator `plasmapy.utils.decorators.validators.validate_quantities()`. Permanently delete decorator `check_quantity()` and its supporting code. For functions `plasmapy.formulary.quantum.chemical_potential()` and `plasmapy.formulary.quantum._chemical_potential_interp()`, add a `RaiseNotImplementedError` due to bug outlined in issue [https://github.com/PlasmaPy/PlasmaPy/issues/726](https://github.com/PlasmaPy/PlasmaPy/issues/726). Associated pytests are marked with `pytest.mark.xfails` and doctests are marked with doctests: +SKIP(#722)

- Add Towncrier automated changelog creation support (#643)
- Move existing “check” decorators to new `plasmapy.utils.decorators` module (#647)
- Allow running our sphinx-gallery examples as Jupyter notebooks via Binder (#656)
- Overhaul CI setup, following the example of SunPy (#657)
- Patch `sphinx_gallery.binder` to output custom links to Binder instance (#658)
- Remove the now unnecessary `astropy_helpers` submodule (#663)
- Followup PR to CI overhaul (#664)
- Add a Codemeta file (`codemeta.json`) (#676)
- Overhaul and simplify CI, add Python 3.8 to tests, bump minimal required package versions, fix docs. (#712)
- Update communication channels in docs (#715)
- Code style fixes to the `atomic` subpackage (#716)
- Clean up main package namespace, removing `plasmapy.test` (#718)
- Reduce precision of tests and doctests to allow for refinements of fundamental constants. (#731)
- Create decorators for checking/validating values and units of function/method input and return arguments. Defined decorators include `check_values()`, `check_units()`, and `validate_quantities()`. These decorators are fully defined by “decorator classes” `CheckBase`, `CheckValues`, `CheckUnits`, and `ValidateQuantities`. (#648)
- Create a decorator to change output of physics functions from “radians/s” to “hz” (#667)
- Added `pytest.mark.slow` to `pytest` markers. Updated documentation to notify developers of functionality. (#677)

Version 0.2.0

Version 0.2.0 is the second development release of PlasmaPy. Alongside a few new features, it brings plentiful refactoring, documentation and back stage improvements.

New Features

- Implement machinery for a Plasma class factory based on PLEP 6
- Create an openPMD Plasma subclass
- Create classes to represent ionization state distributions for one or more elements or isotopes.
- Add basic particle drifts to `plasmapy.physics.drifts`
- Turn most dependencies into optional, subpackage-specific ones
Bug Fixes

- Improve handling of NumPy arrays for plasma parameter and transport functions.
- Vendor the roman package so as to allow installation via Conda
- Decrease strictness of check_quantity to allow nan and inf by default

Changes to API

- Move transport from physics to its own subpackage.

Version 0.1.1

Version 0.1.1 is a bugfix patch release correcting a number of issues that arose during the release process and adding two minor convenience features.

New Features

- Add plasmapy.online_help()
- Add plasmapy.__citation__ containing a BibTeX reference.

Bug Fixes

- Bring back mistakenly removed Cython versions of plasma parameters.
- Optimize check_relativistic.
- Correct a failing import statement.
- Fix a number of issues with the Maxwellian distribution in physics.distribution.

Version 0.1.0

Version 0.1.0 is the initial development release of PlasmaPy. This version is a prototype and a preview, and is not feature complete. Significant changes to the API are expected to occur between versions 0.1.0 and 0.2.0, including backward incompatible changes.

New Features

- Adopted the PlasmaPy Community Code of Conduct.
- Created a guide on Contributing to PlasmaPy.
- Adopted a permissive BSD 3-clause license with protections against software patents.
- Set up continuous integration testing with Travis CI, CircleCI, and AppVeyor, along with test coverage checks with Coveralls.
- Decided upon code and docstring style conventions and set up automated code style checks with pep8speaks.
- Developed online documentation for PlasmaPy that is hosted by Read the Docs.
Automated documentation builds with Sphinx.

Wrote narrative documentation for each subpackage.

- Adopted use of units as a units package.
- Created the atomic subpackage to provide easy access to commonly used atomic data.
  - Created a functional interface to access particle properties and find the energy released from nuclear reactions.
  - Created the Particle class as an object-oriented interface to the atomic subpackage.
  - Created the particle_input decorator.
- Created the classes subpackage that includes the prototype Plasma3D, PlasmaBlob, and Species classes.
- Created the constants subpackage.
- Created the mathematics subpackage that contains analytical functions commonly used in plasma physics.
- Created the physics subpackage with its transport module to calculate plasma parameters, transport coefficients, dielectric tensor elements, collision rates, and relativity/quantum physics parameters used in plasma physics.
- Created the utils subpackage.
  - Created check_quantity and check_relativistic decorators.
  - Created custom exceptions.
  - Added import helper and test helper functionality.
- Began development of the diagnostics subpackage.
  - Created a module to interpret Langmuir probe data.
- Created a repository for PlasmaPy Enhancement Proposals.
- Began using type hint annotations.
- Set up architecture to incorporate Cython into performance-critical sections of code.
- Incorporated import and setup tools from the astropy_helpers package.
- Set up a page describing the Stability of Subpackages.

Changes to API

- PlasmaPy now has an API.

Bug Fixes

- Fixed bug in universe that cause solar neutrinos to oscillate between different flavors.

4.1.4 PlasmaPy’s Vision Statement

About PlasmaPy

PlasmaPy is a community-developed and community-driven free and open source Python package that provides common functionality required for plasma physics in a single, reliable codebase.
Motivation

In recent years, researchers in many different scientific disciplines have worked together to develop core Python packages such as Astropy, SunPy, and SpacePy. These packages provide core functionality, common frameworks for data visualization and analysis, and educational tools for their respective scientific disciplines. We believe that a similar cooperatively developed package for plasma physics will greatly benefit our field. In this document, we lay out our vision for PlasmaPy: a community-developed and community-driven open source core Python software package for plasma physics.

There is considerable need in plasma physics for open, general purpose software framework using modern best practices for scientific computing. As most scientific programmers are largely self-taught, software often does not take advantage of these practices and is instead written in a rush to produce results for the next research paper. The resulting code is often difficult to read and maintain, the documentation is usually inadequate, and tests are typically implemented late in the development process if at all. Legacy code is often written in low level languages such as Fortran, which typically makes compiling and installing packages difficult and frustrating, especially if it calls external libraries. It is also unusual to share code, and access to major software is often restricted in some way, resulting in many different programs and packages which do essentially the same thing but with little or no interoperability. These factors lead to research that is difficult to reproduce, and present a significant barrier to entry for new users.

The plasma physics community is slowly moving in the open source direction. Several different types of packages and software have been released under open source licences, including the UCLA PIC codes, PICCANTE, EPOCH, VPIC, PICConGPU, WARP, the FLASH framework, Athena, and PENCIL. These projects are built as individual packages, are written in different programming languages, and often have many dependencies on specific packages. Python packages such as Astropy, SunPy, and SpacePy have had notable success providing open source alternatives to legacy code in related fields. We are grateful to these communities for their hard work, and hope to build upon their accomplishments for the field of plasma physics.

An end user might not always be interested in a complicated powerpack to perform one specific task on supercomputers. She might also be interested in performing some basic plasma physics calculations, running small desktop scale simulations to test preliminary ideas (e.g., 1D MHD/PIC or test particles), or even comparing data from two different sources (simulations vs. spacecraft). Such tasks require a central platform. This is where PlasmaPy comes in.

PlasmaPy Community Code of Conduct

Please see the attached PlasmaPy Community Code of Conduct.

Organizational Structure

The Coordinating Committee (CC) will oversee the PlasmaPy project and code development. The CC will ensure that roles are being filled, facilitate community-wide communication, coordinate and delegate tasks, manage the project repository, oversee the code review process, regulate intercompatibility between different subpackages, seek funding mechanisms, facilitate compromises and cooperation, enforce the code of conduct, and foster a culture of appreciation.

The Community Engagement Committee (CEC) will be responsible for organizing conferences, trainings, and workshops; maintaining and moderating social media groups and accounts; overseeing PlasmaPy’s website; and communicating with the PlasmaPy and plasma physics communities. The CEC will facilitate partnerships with groups such as Software Carpentry.

Each subpackage will have lead and deputy coordinators who will guide and oversee the development of that subpackage.

The Accessibility Coordinator will work to ensure that the PlasmaPy codebase, documentation, and practices are accessible to disabled students and scientists. Additional roles include the Webpage Maintainer, the Release Coordinator, and the Testing Coordinator.
The work undertaken by each of these groups and coordinators should be done openly and transparently, except where confidentiality is needed. We will strive to have multiple subfields from plasma physics in each committee. Major decisions should ideally be made by general consensus among the PlasmaPy community, but when consensus is not possible then the committees may decide via majority vote. Much of this section is following the organizational structure of Astropy.

**Development Procedure**

The initial developers of PlasmaPy will create a flexible development roadmap that outlines and prioritizes subpackages to be developed. The developers will survey existing open source Python packages in plasma physics. Priority will be given to determining how data will be stored and structured. Developers will break up into small groups to work on different subpackages. These small groups will communicate regularly and work towards interoperability and common coding practices.

Because Python is new to many plasma physicists, community engagement is vital. The CEC will arrange occasional informal trainings early in the project that are director towards the initial developers.

New code and edits should be submitted as a pull request to the development branch of the PlasmaPy repository on GitHub. The pull request will undergo a code review by the subpackage maintainers and/or the CC, who will provide suggestions on how the contributor may update the pull request. Subpackage maintainers will generally be responsible for deciding on pull requests with minor changes, while pull requests with major changes should be decided jointly by the subpackage maintainers and the CC. The CC and CEC will develop a friendly guide on how users may contribute new code to PlasmaPy.

New code should conform to the PEP 8 style guide for Python code and the established coding style within PlasmaPy. New code should be submitted with documentation and tests. Documentation should be written primarily in docstrings and follow the numpydoc documentation style guide. Every new module, class and function should have an appropriate docstring. The documentation should describe the interface and the purpose for the method, but generally not the implementation. The code itself should be readable enough to be able to explain how it works. Documentation should be updated when the code is edited. The tests should cover new functionality (especially methods with complex logic), but the tests should also be readable and easy to maintain. Existing tests should be updated when necessary [e.g., during the initial development of a new feature when the application program interface (API) is not yet stable], but with caution since this may imply loss of backwards compatibility.

Members of the PlasmaPy community may submit PlasmaPy Enhancement Proposals (PLEPs) to suggest changes such as major reorganization of a subpackage, creation of a new subpackage, non-backwards compatible changes to a stable package, or significant changes to policies and procedures related to the organization of this project. The issues list on GitHub will generally be more appropriate for changes that do not require community discussion. The CC shall maintain a GitHub repository of PLEPs. PLEPs will be made openly available for community discussion and transparency for a period of at least four weeks, during which time the proposal may be updated and revised by the proposers. The CC shall approve or decline these proposals after seeking community input. The rationale behind the decision and a summary of the community discussion shall be recorded along with the PLEP.

**Programming Guidelines**

**Choice of Languages**

PlasmaPy shall be written using Python 3. PlasmaPy shall initially guarantee compatibility with Python 3.6 and above. Python 3 is continually growing, so we will proceed on the general principle that future updates to PlasmaPy remain compatible with releases of Python that are up to two years old. Python 2.7 and below will not be supported as these versions will no longer be updated past 2020. The core package will initially be written solely in Python.

Code readability is more important than optimization, except when performance is critical. Code should be optimized only after getting it to work, and primarily for where there is a performance bottleneck. Performance-critical parts of
the core package will preferably be written using Numba to achieve compiled speeds while maintaining the significant advantages of using a high level language.

Versioning

PlasmaPy will use Semantic Versioning. Releases will be given version numbers of the form MAJOR.MINOR.PATCH, where MAJOR, MINOR, and PATCH are nonnegative integers. Starting with version 1.0, MAJOR will be incremented when backwards incompatible changes are made, MINOR will be incremented when new backwards-compatible functionality is added, and PATCH will be incremented when backwards-compatible bug fixes are made.

Development releases will have MAJOR equal to zero and start at version 0.1. The API should not be considered stable during the development phase. PlasmaPy will release version 1.0 once it has a stable public API that users are depending on for production code.

All releases will be provided with release notes and change log entries, and a table will be provided that describes the stability of the public API for each PlasmaPy subpackage.

Dependencies

Dependencies have the advantage of providing capabilities that will enhance PlasmaPy and speed up its development, but the disadvantage that they can make manual installation more difficult and potentially frustrating. Package managers such as Anaconda and Homebrew greatly simplify installation of Python packages, but there will be situations where manual installation is necessary (e.g., on some supercomputers without package managers). The core package should be able to be imported using a minimal number of packages (e.g., NumPy, SciPy, and matplotlib) without getting an import error. Additional packages may be included as dependencies of the core package if there is a strong need for it, and if these packages are easily installed with currently available package managers. Subpackages may use additional dependencies when appropriate.

Affiliated Packages

We will follow the practice of Astropy by having a core package and affiliated packages. The core package will contain common tools and base functionality that most plasma physicists will need. The affiliated packages contained in separate repositories will include more specialized functionality that is needed for subfields of plasma physics. This approach will reduce the likelihood of scope creep for the core package while maintaining avenues for broader development.

Units

Multiple sets of units are used by plasma physicists. There exist some peculiarities with how units are used within plasma physics, such as how an electron volt is typically used as a measurement of temperature. Code will be most readable and maintainable if written assuming a particular set of units, but there should be enough flexibility for people in different subfields to choose their preferred set of units. As the generally most common accepted international standard, SI base units will be utilized. We will use an existing Python module (e.g., astropy.units or pint) to assign units to variables and allow straightforward conversion between different systems of units.
CHAPTER 5

Index

- genindex
- modindex
- search


p

plasmapy.atomic, 111
plasmapy.classes, 101
plasmapy.classes.sources, 104
plasmapy.diagnostics.langmuir, 90
plasmapy.formulary.braginskii, 61
plasmapy.formulary.collisions, 72
plasmapy.formulary.dielectric, 12
plasmapy.formulary.dimensionless, 17
plasmapy.formulary.dispersionfunction, 18
plasmapy.formulary.distribution, 20
plasmapy.formulary.drifts, 31
plasmapy.formulary.magnetostatics, 32
plasmapy.formulary.mathematics, 11
plasmapy.formulary.parameters, 37
plasmapy.formulary.quantum, 55
plasmapy.formulary.relativity, 60
plasmapy.utils, 159
plasmapy.utils.decorators, 159
plasmapy.utils.exceptions, 176
plasmapy.utils.pytest_helpers, 177
Symbols

__call__() (plasmapy.utils.decorators.CheckUnits method), 171
__call__() (plasmapy.utils.decorators.CheckValues method), 172
__call__() (plasmapy.utils.decorators.ValidateQuantities method), 175

A

abundances (plasmapy.atomic.IonizationStates attribute), 144
add_magnetostatic() (plasmapy.classes.sources.Plasma3D method), 106
alfven_speed (plasmapy.classes.sources.Plasma3D attribute), 106
Alfven_speed() (in module plasmapy.formulary.parameters), 39
all_variables (plasmapy.formulary.braginskii.ClassicalTransport attribute), 69
angular_freq_to_hz() (in module plasmapy.atomic), 149
antiparticle (plasmapy.atomic.Particle attribute), 149
assert_can_handle_nparray() (in module plasmapy.utils.pytest_helpers), 178
atomic_number (plasmapy.atomic.IonizationState attribute), 140
atomic_number (plasmapy.atomic.Particle attribute), 149
atomic_number() (in module plasmapy.atomic), 120
atomic_symbol() (in module plasmapy.atomic), 120
average_ionization (plasmapy.classes.BasePlasma attribute), 102
average_ionization() (plasmapy.classes.GenericPlasma method), 103

B

baryon_number (plasmapy.atomic.Particle attribute), 149
base_particle (plasmapy.atomic.IonizationState attribute), 140
base_particles (plasmapy.atomic.IonizationStates attribute), 144
BasePlasma (class in plasmapy.classes), 102
beta() (in module plasmapy.formulary.dimensionless), 17
bias (plasmapy.diagnostics.langmuir.Characteristic attribute), 100
binding_energy (plasmapy.atomic.Particle attribute), 150
boris_push() (plasmapy.simulation.particletracker.ParticleTracker method), 110

C

categories (plasmapy.atomic.Particle attribute), 150
Characteristic (class in plasmapy.diagnostics.langmuir), 100
charge (plasmapy.atomic.Particle attribute), 150
charge_density (plasmapy.classes.sources.HDF5Reader attribute), 104
check_relativistic() (in module plasmapy.utils.pytest_helpers), 178
check_units() (in module plasmapy.utils.decorators), 161
check_values() (in module plasmapy.utils.decorators), 162
CheckBase (class in plasmapy.utils.decorators), 168
checks (plasmapy.utils.decorators.CheckBase attribute), 168
CheckUnits (class in plasmapy.utils.decorators), 168
CheckValues (class in plasmapy.utils.decorators), 168
chemical_potential() (in module plasmapy.formulary.quantum), 59
CircularWire (class in plasmapy.formulary.magnetostatics), 33
Classical() (in module plasmapy.formulary.collisions), 74

ClassicalTransport (class in plasmapy.formulary.braginskii), 67
close() (plasmapy.classes.sources.HDF5Reader method), 104
cold_plasma_permittivity_LRP() (in module plasmapy.formulary.dielectric), 14
cold_plasma_permittivity_SDP() (in module plasmapy.formulary.dielectric), 13
collision_frequency() (in module plasmapy.formulary.collisions), 78
coupling() (plasmapy.classes.sources.PlasmaBlob attribute), 107
coupling_parameter() (in module plasmapy.formulary.collisions), 88
couplingWarning, 176
current (plasmapy.diagnostics.langmuir.Characteristic attribute), 100

deBroglie_wavelength() (in module plasmapy.formulary.quantum), 56
debye_length() (in module plasmapy.formulary.parameters), 50
debye_number() (in module plasmapy.formulary.parameters), 50
density (plasmapy.classes.sources.Plasma3D attribute), 105
domain_shape (plasmapy.classes.sources.Plasma3D attribute), 105

eff_m (plasmapy.simulation.particletracker.ParticleTracker attribute), 110
eff_q (plasmapy.simulation.particletracker.ParticleTracker attribute), 109
electric_charge() (in module plasmapy.atomic), 122
electric_current (plasmapy.classes.sources.HDF5Reader attribute), 104
electric_field (plasmapy.classes.sources.HDF5Reader attribute), 104
electric_field_strength (plasmapy.classes.sources.Plasma3D attribute), 106

electron_density (plasmapy.classes.BasePlasma attribute), 102
electron_density (plasmapy.classes.sources.PlasmaBlob attribute), 107
electron_density() (plasmapy.classes.GenericPlasma method), 103
electron_number (plasmapy.atomic.Particle attribute), 150
electron_temperature (plasmapy.classes.BasePlasma attribute), 102
electron_temperature (plasmapy.classes.sources.PlasmaBlob attribute), 107
electron_temperature() (plasmapy.classes.GenericPlasma method), 103
electron_thermal_conductivity (plasmapy.formulary.braginskii.ClassicalTransport attribute), 69
electron_thermal_conductivity() (in module plasmapy.formulary.braginskii), 65
electron_viscosity (plasmapy.formulary.braginskii.ClassicalTransport attribute), 70
electron_viscosity() (in module plasmapy.formulary.braginskii), 66
element (plasmapy.atomic.IonizationState attribute), 140
element (plasmapy.atomic.Particle attribute), 151
element_name (plasmapy.atomic.Particle attribute), 151
element_name() (in module plasmapy.atomic), 123
equil_ionic_fractions (plasmapy.atomic.IonizationState attribute), 140
equilibrate() (plasmapy.atomic.IonizationState method), 141
equilibrate() (plasmapy.atomic.IonizationStates method), 144

exB_drift() (in module plasmapy.formulary.drifts), 31

extract_exponential_section() (in module plasmapy.diagnostics.langmuir), 95
extract_ion_section() (in module plasmapy.diagnostics.langmuir), 96
extrapolate_electron_current() (in module plasmapy.diagnostics.langmuir), 97
extrapolate_ion_current_OML() (in module plasmapy.diagnostics.langmuir), 99
PlasmaPy Documentation, Release 0.2.0

Index 255

G

Fermi_integral() (in module plasmapy.formulary.quantum), 57
Fermi_integral() (in module plasmapy.formulary.mathematics), 11
FiniteStraightWire (class in plasmapy.formulary.magnetostatics), 34
force_drift() (in module plasmapy.formulary.drifts), 32
fundamental_electron_collision_freq() (in module plasmapy.formulary.collisions), 80
fundamental_ion_collision_freq() (in module plasmapy.formulary.collisions), 81
G

GeneralWire (class in plasmapy.formulary.magnetostatics), 34
GenericPlasma (class in plasmapy.classes), 103
get_EEDF() (in module plasmapy.diagnostics.langmuir), 99
get_electron_density_LM() (in module plasmapy.diagnostics.langmuir), 95
get_electron_saturation_current() (in module plasmapy.diagnostics.langmuir), 93
get_electron_temperature() (in module plasmapy.diagnostics.langmuir), 96
get_floating_potential() (in module plasmapy.diagnostics.langmuir), 93
get_ion_density_LM() (in module plasmapy.diagnostics.langmuir), 94
get_ion_density_OML() (in module plasmapy.diagnostics.langmuir), 98
get_ion_saturation_current() (in module plasmapy.diagnostics.langmuir), 94
get_padded_limit() (plasmapy.diagnostics.langmuir.Characteristic method), 100
get_plasma_potential() (in module plasmapy.diagnostics.langmuir), 92
get_unique_bias() (plasmapy.diagnostics.langmuir.Characteristic method), 100
grid (plasmapy.classes.sources.Plasma3D attribute), 105
gyrofrequency() (in module plasmapy.formulary.parameters), 46
gyroradius() (in module plasmapy.formulary.parameters), 47
H

half_life (plasmapy.atomic.Particle attribute), 151
half_life() (in module plasmapy.atomic), 124
Hall_parameter() (in module plasmapy.formulary.parameters), 45
HDF5Reader (class in plasmapy.classes.sources), 104

I

impact_parameter() (in module plasmapy.formulary.collisions), 76
impact_parameter_perp() (in module plasmapy.formulary.collisions), 75
ImplicitConversionsWarning, 177
InconsistentTypeError, 182
IncorrectResultError, 182
inertial_length() (in module plasmapy.formulary.parameters), 51
InfiniteStraightWire (class in plasmapy.formulary.magnetostatics), 35
info() (plasmapy.atomic.IonizationState method), 141
info() (plasmapy.atomic.IonizationStates method), 145
integer_charge (plasmapy.atomic.Particle attribute), 151
integer_charge() (in module plasmapy.atomic), 124
integer_charges (plasmapy.atomic.IonizationState attribute), 140
InvalidTestError, 182
ion_density (plasmapy.classes.BasePlasma attribute), 102
ion_density() (plasmapy.classes.GenericPlasma method), 103
ion_sound_speed() (in module plasmapy.formulary.parameters), 40
ion_temperature (plasmapy.classes.BasePlasma attribute), 102
ion_temperature() (plasmapy.classes.GenericPlasma method), 103
ion_thermal_conductivity (plasmapy.formulary.braginskii.ClassicalTransport attribute), 70
ion_thermal_conductivity() (in module plasmapy.formulary.braginskii), 64
ion_viscosity (plasmapy.formulary.braginskii.ClassicalTransport attribute), 70
ion_viscosity() (in module plasmapy.formulary.braginskii), 66
ionic_fractions (plasmapy.atomic.IonizationState attribute), 140
ionic_fractions (plasmapy.atomic.IonizationStates attribute), 144
ionic_symbol (plasmapy.atomic.Particle attribute), 152
ionic_symbol() (in module plasmapy.atomic), 125
ionic_symbols (plasmapy.atomic.IonizationState attribute), 140
ionization (plasmapy.classes.sources.PlasmaBlob attribute), 107
IonizationState (class in plasmapy.atomic), 138
IonizationStates (class in plasmapy.atomic), 142
ionize() (plasmapy.atomic.Particle method), 156
is_category() (plasmapy.atomic.Particle method), 157
is_datasource_for() (plasmapy.classes.sources.HDF5Reader class method), 104
is_datasource_for() (plasmapy.classes.sources.Plasma3D class method), 106
is_datasource_for() (plasmapy.classes.sources.PlasmaBlob class method), 107
is_electron (plasmapy.atomic.Particle attribute), 152
is_ion (plasmapy.atomic.Particle attribute), 152
is_stable() (in module plasmapy.atomic), 126
isotope (plasmapy.atomic.IonizationState attribute), 140
isotope (plasmapy.atomic.Particle attribute), 152
isotope_name (plasmapy.atomic.Particle attribute), 152
isotope_symbol() (in module plasmapy.atomic), 127
isotopic_abundance (plasmapy.atomic.Particle attribute), 153
isotopic_abundance() (in module plasmapy.atomic), 127

Kappa (plasmapy.atomic.IonizationState attribute), 140
kappa (plasmapy.atomic.IonizationStates attribute), 144
kappa_thermal_speed() (in module plasmapy.formulary.parameters), 44
kappa_velocity_1D() (in module plasmapy.formulary.distribution), 28
kappa_velocity_3D() (in module plasmapy.formulary.distribution), 29
kinetic_energy (plasmapy.simulation.particletracker.ParticleTracker attribute), 110
kinetic_energy_history (plasmapy.simulation.particletracker.ParticleTracker attribute), 110
known_isotopes() (in module plasmapy.atomic), 128
Knudsen_number() (in module plasmapy.formulary.collision), 87

L
lepton_number (plasmapy.atomic.Particle attribute), 153
log_abundances (plasmapy.atomic.IonizationStates attribute), 144
Lorentz_factor() (in module plasmapy.formulary.relativity), 60
lower_hybrid_frequency() (in module plasmapy.formulary.parameters), 54

M
m (plasmapy.simulation.particletracker.ParticleTracker attribute), 109
magnetic_energy_density() (in module plasmapy.formulary.parameters), 53
magnetic_field (plasmapy.classes.sources.HDF5Reader attribute), 104
magnetic_field (plasmapy.classes.sources.Plasma3D attribute), 105
magnetic_field() (plasmapy.formulary.magnetostatics.CircularWire method), 33
magnetic_field() (plasmapy.formulary.magnetostatics.FiniteStraightWire method), 34
magnetic_field() (plasmapy.formulary.magnetostatics.GeneralWire method), 35
magnetic_field() (plasmapy.formulary.magnetostatics.InfiniteStraightWire method), 36
magnetic_field() (plasmapy.formulary.magnetostatics.MagneticDipole method), 36
magnetic_field() (plasmapy.formulary.magnetostatics.MagnetoStatics method), 37
magnetic_field_strength (plasmapy.classes.sources.Plasma3D attribute), 106
magnetic_pressure() (in module plasmapy.formulary.parameters), 52
Mass (plasmapy.atomic.Particle attribute), 153
mass_density() (in module plasmapy.formulary.parameters), 38
mass_number (plasmapy.atomic.Particle attribute), 154
mass_number() (in module plasmapy.atomic), 129
Maxwellian_1D() (in module plasmapy.formulary.distribution), 21
Maxwellian_speed_1D() (in module plasmapy.formulary.distribution), 25
Maxwellian_speed_2D() (in module plasmapy.formulary.distribution), 26
Maxwellian_speed_3D() (in module plasmapy.formulary.distribution), 27
Maxwellian_velocity_2D() (in module plasmapy.formulary.distribution), 22
Maxwellian_velocity_3D() (in module plasmapy.formulary.distribution), 23
mean_free_path() (in module plasmapy.formulary.collisions), 83
MissingExceptionError, 182
MissingWarningError, 182
mobility() (in module plasmapy.formulary.collisions), 85
momentum (plasmapy.classes.sources.Plasma3D attribute), 105
n (plasmapy.atomic.IonizationStates attribute), 144
n_e (plasmapy.atomic.IonizationStates attribute), 141
n_e (plasmapy.atomic.IonizationStates attribute), 144
n_elem (plasmapy.atomic.IonizationState attribute), 141
neutron_number (plasmapy.atomic.Particle attribute), 154
normalize() (plasmapy.atomic.IonizationState method), 141
normalize() (plasmapy.atomic.IonizationStates method), 145
nuclear_binding_energy() (in module plasmapy.atomic), 130
nuclear_reaction_energy() (in module plasmapy.atomic), 130
nuclide_mass (plasmapy.atomic.Particle attribute), 155
number_densities (plasmapy.atomic.IonizationState attribute), 141
number_densities (plasmapy.atomic.IonizationStates attribute), 144

P
Particle (class in plasmapy.atomic), 145
class in plasmapy.atomic, 145
particle (plasmapy.atomic.Particle attribute), 155
particle_input() (in module plasmapy.atomic), 132
particle_mass() (in module plasmapy.atomic), 134
particle_symbol() (in module plasmapy.atomic), 134
ParticleTracker (class in plasmapy.simulation.particletracker), 109
periodic_table (plasmapy.atomic.Particle attribute), 155
nuclear_binding_energy() (in module plasmapy.formulary.dielectric), 15
plasma_dispersion_func() (in module plasmapy.formulary.dispersionfunction), 18
plasma_frequency() (in module plasmapy.formulary.parameters), 19
PlasmaBlob (class in plasmapy.classes.sources), 106
PlasmaFactory (class in plasmapy.classes.plasma_factory), 101
plasmapy.atomic (module), 111
plasmapy.classes (module), 101
plasmapy.classes.sources (module), 104
plasmapy.diagnostics.langmuir (module), 90
plasmapy.formulary.braginskii (module), 61
plasmapy.formulary.collisions (module), 72
plasmapy.formulary.dielectric (module), 12
plasmapy.formulary.dimensionless (module), 17
plasmapy.formulary.dispersionfunction (module), 18
plasmapy.formulary.distribution (module), 20
plasmapy.formulary.drifts (module), 31
plasmapy.formulary.magnetostatics (module), 32
plasmapy.formulary.mathematics (module), 11
plasmapy.formulary.parameters (module), 37
plasmapy.formulary.quantum (module), 55
plasmapy.formulary.relativity (module), 60
plasmapy.utils (module), 159
plasmapy.utils.decorators (module), 159
plasmapy.utils.exceptions (module), 176
plasmapy.utils.pytest_helpers (module), 177
PlasmaPyError, 176
PlasmaPyWarning, 176
plot() (plasmapy.diagnostics.langmuir.Characteristic method), 100
plot_time_trajectories() (plasmapy.simulation.particletracker.ParticleTracker method), 111
plot_trajectories() (plasmapy.simulation.particletracker.ParticleTracker method), 111
position_history (plasmapy.simulation.particletracker.ParticleTracker attribute), 109
preserve_signature() (in module plasmapy.utils.decorators), 164
pressure (plasmapy.classes.sources.Plasma3D attribute), 105

Q
q (plasmapy.simulation.particletracker.ParticleTracker attribute), 109
Z\textsubscript{most\_abundant} \textit{(plasmapy.atomic.IonizationState attribute)}, 140
Z\textsubscript{rms} \textit{(plasmapy.atomic.IonizationState attribute)}, 140