pelpi Documentation

Release 0.1

lesnat

February 23, 2018

short description

CHAPTER

ONE

CONTENTS

Introduction

Module documentation ...

Tatata

Install

Get the source

Dependancies

Installation

Test integrity

Understand

Package aim

Estimate ...

Philosophy

All in methods

Code structure

Use

Setting target parameters

Comments ...

Material

class `pelpi.`**`Material`**`(`*density=None*, *atomic_mass=None*, *Z=None*`)`

Class for defining material properties.

Parameters

- **`density`** (*`mass/length**3 Quantity`*) – Material density
- **`atomic_mass`** (*`mass Quantity`*) – Atomic or molecular mass
- **`Z`** (*`dimensionless Quantity`*) – Atomic number or number of charges per molecule

`Z``()`

Returns **User input ‘Z‘**

Return type dimensionless Quantity

`atomic_mass``()`

Returns **User input ‘atomic_mass‘**

Return type mass Quantity

`density``()`

Returns **User input ‘density‘**

Return type mass/length**3 Quantity

Target

class `pelpi.`**`Target`**`(`*material*`)`

Class for defining the target characteristics.

Parameters **`material`** (*`object`*) – pelpi `Material` instance

Variables **`material`** (*`object`*) – Reference to the input *material* instance

Setting laser parameters

Comments ...

Profile

class `pelpi.`**`Profile`**`(`*profile=None*, *fwhm=None*, *radius=None*`)`

Bases: `pelpi._tools._PelpiObject`

Class for defining a geometrical profile.

Used for defining Laser profiles (spatial and temporal) and might be used in Target object in the future.

Parameters

- **`profile`** (*`str`*) – Geometrical profile.
- **`fwhm`** (*`Quantity, optional`*) – Full Width Half Maximum of the profile.
- **`radius`** (*`Quantity, optional`*) – Radius of the profile

Notes

Available profiles are

`gaussian1D` : one dimension gaussian profile. It equals 1 for x=0.

`gaussian2D` : two dimensions isotropic gaussian profile. It equals 1 for x=0.

`top-hat` : two dimensions top-hat isotropic profile. It equals 1 if |x|<radius, 0 otherwise.

If `profile` is `gaussian1D`, you must define `fwhm`.

If `profile` is `gaussian2D`, you must define `fwhm`.

If `profile` is `top-hat`, you must define `radius`.

Examples

You can set a laser time profile as follows

```
>>> import pelpi as pp
>>> tprof = pp.Profile(
...     profile   = "gaussian1D",
...     fwhm      = 30 * pp.unit('fs')
...     )
...
```

envelope (*x*)

Returns **Profile envelope at x**

Return type dimensionless Quantity

Parameters **x** (*Quantity*) – Axis

Notes

envelope is centered at x=0 and has a maximum value of 1.

fwhm ()

Returns **User input ‘fwhm‘**

Return type Quantity

integral1D ()

Returns **Integration of envelope under x**

Return type Quantity

Notes

Yet only analytical integrals are implemented, as the available profiles permit it.

Analytical solutions are

For `gaussian1D`

$$I_x = \sqrt{\pi}\frac{t_{FWHM}}{2\sqrt{\ln 2}}$$

integral2D()

Returns **Double integration of the envelope under x**

Return type Quantity

Notes

Yet only analytical integrals are implemented, as the available profiles permit it.

Analytical solutions are

For `gaussian2D`

$$I_x = \pi(\frac{x_{FWHM}}{2\sqrt{\ln 2}})^2$$

For `top-hat`

$$I_x = \pi r^2$$

profile()

Returns **User input 'profile'**

Return type str

radius()

Returns **User input 'radius'**

Return type Quantity

Laser

class `pelpi.`**`Laser`**(*time_profile=None*, *space_profile=None*, *wavelength=None*, *energy=None*)

Bases: `pelpi._tools._PelpiObject`

Class for defining laser characteristics, and do some simple calculations.

Parameters

- **time_profile** (*object*) – Pulse time profile. pelpi `Profile` instance
- **space_profile** (*object*) – Pulse space profile (waist). pelpi `Profile` instance
- **wavelength** (*length Quantity*) – Laser monochromatic wavelength
- **energy** (*energy Quantity*) – Total energy of the laser pulse

Variables

- **time_profile** (*object*) – Input time_profile instance
- **space_profile** (*object*) – Input space_profile instance

Examples

Assuming you defined two `Profile` objects as `tprof` and `sprof`, you can instanciate a `Laser` class as follows

```
>>> laser=pp.Laser(
...     wavelength        = 0.8 * pp.unit('um'),
...     energy            = 2.0 * pp.unit('J'),
...     time_profile      = tprof,
...     space_profile     = sprof
...     )
...
```

and then print some calculations

```
>>> I0 = laser.intensity(r=0*pp.unit('um'),t=0*pp.unit('fs'))
>>> print("Laser peak intensity : {}".format(I0))
>>> a0 = laser.intensity_peak_normalized()
>>> print("Laser normalized peak intensity : {}".format(a0))
>>> nc = laser.electron.number_density_critical()
>>> print("Critical number density : {}".format(nc))
```

`angular_frequency`()

Returns **Laser angular frequency**

Return type 1/time Quantity

Notes

angular_frequency is defined as follows

`energy`()

Returns **User input ‘energy‘**

Return type energy Quantity

`envelope`(*r*, *t*)

Returns **Pulse envelope at given time and radius**

Return type dimensionless Quantity

Parameters

- **r** (*length Quantity*) – Radius
- **t** (*time Quantity*) – Time

Notes

envelope is centered at t=0 and r=0, and has a maximum value of 1.

`intensity`(*r=<Quantity(0, ‘meter’)>, t=<Quantity(0, ‘second’)>*)

Returns **Intensity at given time and radius**

Return type power/length**2 Quantity

Parameters

- **r** (*length Quantity*) – Radius
- **t** (*time Quantity*) – Time

Notes

Default behaviour gives the peak intensity.

Intensity is defined as follows

`intensity_peak_normalized``()`

Returns **Normalized laser peak intensity**

Return type dimensionless Quantity

Notes

The normalized laser intensity a_0 is defined as follows

$$a_0 = 0.85 \times \sqrt{I_{18}\lambda_\mu^2}$$

with I_{18} the laser peak intensity in $10^{18} W.cm^{-2}$ and λ_μ the laser wavelength in $10^{-6} m$.

`power``(`*r=<Quantity(0, 'meter')>*, *t=<Quantity(0, 'second')>*`)`

Returns **Power at given time and radius**

Return type power Quantity

Parameters

- **`r`** (*`length Quantity`*) – Radius
- **`t`** (*`time Quantity`*) – Time

Notes

Default behaviour gives the peak power.

power is defined as follows

`wavelength``()`

Returns **User input 'wavelength'**

Return type length Quantity

Estimate laser plasma interaction

Comments ...

LaserPlasmaInteraction

class `pelpi.`**`LaserPlasmaInteraction`**`(`*laser*, *target*`)`

Bases: `pelpi._tools._PelpiObject`

Class for estimations in laser-plasma interaction.

Parameters

- **`laser`** (*`object`*) – pelpi `Laser` instance

- **target** (*object*) – pelpi Target instance

Variables

- **laser** (*object*) – Input laser instance
- **target** (*object*) – Input target instance
- ***model*** (*object*) – Contains the all available models. Class attribute.
- **plasma** (*object*) – Contains usual plasma parameters
- **electron** (*object*) – Contains estimations about electrons

Examples

TODO

Notes

### New methods in input objects

TODO

### Keyword arguments (**kwargs) in estimation methods

Even if the code structure permit not to give a lot of parameters to perform an estimate, some complex models may need several of them to give the result.

These parameters can be choosen by the user (for order of magnitude) or calculated by other models and passed as argument.

To do this, pelpi needs the parameters to be defined explicitly, i.e. with the parameter name in the method call.

Here is a quick example, assuming `lpi` is an instance of `LaserPlasmaInteraction`

```
>>> eh = lpi.electron.hot
>>> # Define the laser absorption efficiency
>>> eta_l = 0.1 * pp.unit('')
>>> # Use a simple model to get a temperature estimate
>>> Teh = eh.temperature(model='Haines2009')
>>> # The model needs a temperature & absorption efficiency to return a result
>>> neh = eh.number_total(model="Common",    ...                         temperature = Teh,     ...
...
>>> neh = eh.number_total(model="Common", Teh, eta_l) # This does not work
```

Refer to the desired method documentation for more informations about parameters of each model. You can access it via pelpi.LaserPlasmaInteraction.model.[Model], or lpi.model.[Model].

`model` = <module 'pelpi.models' from '/home/users1/esnault/Installation/pelpi/pelpi/models.pyc'>

plasma

electron

`LaserPlasmaInteraction.`**`_Electron`** **= <class 'pelpi.lpi._Electron'>**

hot

Estimate PIC parameters

ParticleInCell

class `pelpi.`**`ParticleInCell`**(*lpi*)

Bases: `pelpi._tools._PelpiObject`

Class for estimate Particle-In-Cell numerical parameters.

Parameters **`lpi`** (*object*) – pelpi `LaserPlasmaInteraction` instance

Variables

- **`lpi`** (*object*) – Input lpi instance
- **`code`** (*object*) – Contains specific code calculations

`length_cell`(*lim*, *temperature=None*)

Returns

Return type Maximal cell length to use, according to the choosen limitation (target, laser or both).

Parameters

- **`lim`** (*str*) – Which limitation to choose
- **`temperature`** (*energy Quantity, optional*) – Choosen temperature for the estimate

Notes

Available *lim* parameters are

“target”, for a result depending on the target limitation of cell length (3.4 Debye length) In this case *temperature* might be defined.

“laser”, for a result depending on the laser limitation of cell length (wavelength / 10)

“both”, for taking the minimum of the two previous results. *temperature* might then also be defined.

More informations can be found in Tskahya et al.

Examples

TODO

`space_resolution`(*lim*, *temperature=None*)

Returns

Return type Minimal space resolution to use, according to the choosen limitation (target, laser or both).

Parameters

- **`lim`** (*str*) – Which limitation to choose

- **temperature** (*energy Quantity, optional*) – Choosen temperature for the estimate

Notes

space_resolution is calculated via the length_cell method, so see length_cell documentation for more information about *lim* and *temperature* parameters.

time_resolution (*lim*, *CFL*, *temperature=None*)

Returns

- *Minimal time resolution to use, according to the choosen limitation (target, laser or both)*
- *and if the Courant–Friedrichs–Lewy condition might be satisfied.*

Parameters

- **lim** (*str*) – Which limitation to choose
- **CFL** (*bool*) – True if CFL condition might be satisfied, False otherwise
- **temperature** (*energy Quantity, optional*) – Choosen temperature for the estimate

Notes

time_resolution is calculated via the length_cell method, so see length_cell documentation for more information about *lim* and *temperature* parameters.

time_step (*lim*, *CFL*, *temperature=None*)

Returns

- *Maximal time step to use, according to the choosen limitation (target, laser or both)*
- *and if the Courant–Friedrichs–Lewy condition might be satisfied.*

Parameters

- **lim** (*str*) – Which limitation to choose
- **CFL** (*bool*) – True if CFL condition might be satisfied, False otherwise
- **temperature** (*energy Quantity, optional*) – Choosen temperature for the estimate

Notes

time_step is calculated via the length_cell method, so see length_cell documentation for more information about *lim* and *temperature* parameters.

Models Reference

LaserPlasmaInteraction

Examples

Print informations

```
# coding:utf8
import sys
Modules_path="../"
if sys.path[0]!=Modules_path:sys.path.insert(0, Modules_path)

# Import modules
import numpy as np
import pelpi as pp
u=pp.unit

# Set user units (default : SI).
# More informations about units can be found in the pint package documentation
pp.default_unit['energy']      = u('MeV')      # Units can be defined like this
pp.default_unit['temperature'] = u('MeV')
pp.default_unit['length']      = u.um          # or like this
pp.default_unit['time']        = u('fs')
pp.default_unit['intensity']   = u.W/u.cm**2   # and can be combined
pp.default_unit['power']       = u('TW')       # It accepts prefixes # long names or short

# Define temporal and spatial laser profiles
tprof=pp.Profile(
    profile         = "gaussian1D",              # Check the doc for available profiles
    fwhm            = 44 * u.fs,                 # Always define value + unit
)
sprof=pp.Profile(
    profile         = "gaussian2D",
    fwhm            = 12 * u.um
)

# Define laser from profiles
laser=pp.Laser(
    wavelength      = 0.8 * u.um,
    energy          = 0.154 * u.J,

    time_profile    = tprof,
    space_profile   = sprof
)

# Print & save results
print("I0 = {}".format(laser.intensity()))
print("a0 = {}".format(laser.intensity_peak_normalized()))
nc = laser.electron.number_density_critical()

# Define a material
Al=pp.Material(
    density         = 2.69890e3 * u('kg/m**3'),
    atomic_mass     = 26.98154 * u('amu'),
    Z               = 13 * u(''),                # dimensionless unit
)

# Instanciate a Target object with a Material object
# TODO: add geometrical stuff via a Profile object
target=pp.Target(Al)

# Save & print results
ne = target.material.electron.number_density()
print("\nElectron density (in critical density)           : {}".format(ne/nc))



# Instanciate a LaserPlasmaInteraction object with Laser and Target objects
lpi=pp.LaserPlasmaInteraction(laser,target)

# Save & print some estimates
Teh = lpi.electron.hot.temperature(model="Haines2009")
print("\nHot electron temperature  (model = Haines2009)    : {}".format(Teh))

# TODO: *args in _Estimate
# n0 = lpi.electron.hot.number_total(model="Common",temperature=Teh,absorption_efficiency=0.4)
# print("Hot electron total number (model = Common)        : {}".format(n0))


# Compare to simulation/experiments
# ...

# Instanciate a ParticleInCell object with a LaserPlasmaInteraction object
pic=pp.ParticleInCell(lpi)

# Set default parameters
# lpi.electron.hot.set('temperature',Teh)

# Get estimates
dx=pic.length_cell('both',temperature=Teh)
Lr=pic.code.smilei.length()

print("\ndx           = {}        = {}".format(dx,dx/Lr))
resx=pic.space_resolution('both',temperature=Teh)
print("resx         = {}        = {}".format(resx,resx*Lr))
print("2 pi * resx  = {}".format(2 * np.pi * resx*Lr))
```

PYTHON MODULE INDEX

INDEX