Module **`solver`**

Classes

`class EquationOfState`

Class to represent an equation of state, i.e. energy density as function of pressure.

Attributes

`number_densities` : `NumPy array`
: NumPy array that stores number densities in cm^-3.

`pressures` : `NumPy array`
: NumPy array that stores pressures in dyne/cm^2.

`energy_densities` : `NumPy array`
: NumPy array that stores energy densities in erg/cm^3.

Methods

load_from_file(file_path) Read data from csv file, given its path as parameter, and store respectively number densities, pressures and energy densities in corresponding NumPy arrays. plot() Plot energy density versus pressure. interpolate() Interpolate using cubic spline and return the equation of state in the form of CubicSpline representation in the form of energy density as function of pressure. get() Return NumPy arrays containing number densities, pressures and energy densities.

Initialise empty NumPy arrays to store number densities, pressures and energy densities.

▸ EXPAND SOURCE CODE

Methods

`def get(self)`

Return NumPy arrays containing respectively number densities, pressures and energy densities.

Returns

`tuple` : `[NumPy array, NumPy array, NumPy array]`
: NumPy arrays of number densities in cm^-3, pressures in dyne/cm^2 and energy densities in erg/cm^3.

`def interpolate(self)`

Interpolate pressures and energy densities into an equation of state using cubic spline (imported from SciPy). Return the equation of state in the form of CubicSpline representation in the form of energy density as function of pressure.

Returns

`CubicSpline`
CubicSpline representation of the equation of state in the form of energy density as function of pressure.

Raises

`ValueError`
No data is loaded to interpolate.

```
def load_from_file(self, file_path)
```

Given its path as a paramater, open csv file, read data and store respectively number densities, pressures and energy densities in corresponding NumPy arrays. The first row is skipped because it is the header. The delimiter must be comma for csv files. In the first column there must be number densities in cm^-3, in the second column there must be pressures in dyne/cm^2, in the third column there must be energy densities in erg/cm^3.

Parameters

`file_path` : `str`
Path of the csv file containing respectively number densities, pressures and energy densities in columns.

Raises

`IOError`
The file is not loaded.

`ValueError`
The file is empty. Pressures are not strictly increasing.

```
def plot(self)
```

Plot energy density versus pressure, using logarithmic scale for both x and y axis with a grid.

Raises

`ValueError`
No data is loaded to plot.

```
class SolverTOVRangePressure (eos, relativity_corrections=True)
```

Class to integrate numerically the system of 2 ordinary differential equations (mass and pressure equations) given a range of initial central pressures. The third equation of the TOV system (equation of state) is given as parameter. It uses the class SolverTOVSinglePressure for each initial central pressure.

Attributes

`eos` : `CubicSpline`
A CubicSpline representation of the equation of state in the form of energy density as function of pressure.

`relativity_corrections` : `bool`
: A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is used, if False the Newtonian pressure equation is used.

`radii` : `NumPy array`
: NumPy array that stores radii in km.

`masses` : `NumPy array`
: NumPy array that stores masses in solar masses.

`initial_pressures` : `NumPy array`
: NumPy array that stores initial pressures in dyne/cm^2.

Methods

solve(step_r, initial_pressures) Integrate the system for each pressure in the range of initial central pressure using the class SolverTOVSinglePressure. Store integrated radii and masses in corresponding NumPy arrays. print_max_mass() Print the maximum mass, its radius and central pressure. get() Return NumPy arrays containing radii, masses and initial pressure.s plot_MRvsP() Plot mass and radius versus initial pressure. plot_MvsR() Plot mass versus radius.

Take an equation of state and an option for relativity corrections and store them as attributes of the class. Initialise NumPy arrays to store radii in km, pressures in dyne/cm^2 and masses in solar masses.

Parameters

`eos` : `CubicSpline`
: A CubicSpline representation of the equation of state in the form of energy density as function of pressure.

`relativity_corrections` : `bool`
: A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is used, if False the Newtonian pressure equation is used.

▸ EXPAND SOURCE CODE

Methods

```
def get(self)
```

Return NumPy arrays containing respectively radii, masses and initial central pressures.

Returns

`tuple` : `[NumPy array, NumPy array, NumPy array]`
: NumPy arrays of radii in km, masses in solar masses and initial central pressures in dyne/cm^2.

Raises

`ValueError`
: No data is computed to get.

```
def plot_MRvsP(self)
```

Plot two subplots: radius versus initial central pressure and mass versus initial central pressure, using logarithmic scale for x with a grid.

Raises

`ValueError`
: No data is computed to plot.

```
def plot_MvsR(self)
```

Show plot of mass versus radius with a grid.

Raises

`ValueError`
: No data is computed to plot.

```
def print_max_mass(self)
```

Print the maximum mass, its corresponding radius and central pressure.

Raises

`ValueError`
: No data is computed to print.

```
def solve(self, step_r, initial_pressures)
```

Integrate the system for each pressure in the range of initial central pressure using the class SolverTOVSinglePressure (based on a 4th-order Runge-Kutta algorithm with a breaking condition when the pressure becomes zero or negative, because it means that the surface of the object is reached). Store integrated radii and masses in corresponding NumPy arrays.

Parameters

`step_r` : `float`
: Radius step in cm.

`initial_pressures` : `NumPy array`
: NumPy array of initial central pressures in dyne/cm^2.

Raises

`ValueError`
: If radius step is negative. If initial pressure is negative.

```
class SolverTOVSinglePressure (eos, relativity_corrections=True)
```

Class to integrate numerically the system of 2 ordinary differential equations (mass and pressure equations) given a single initial central pressure. The third equation of the TOV system (equation of state) is given as parameter. Implement the 4th-order Runge-Kutta algorithm with a breaking condition when the pressure becomes zero or negative, because it means that the surface of the object is reached. Store integrated masses and pressures in corresponding NumPy arrays.

Attributes

radii : `NumPy array`

NumPy array that stores radii in km.

masses : `NumPy array`

NumPy array that stores masses in solar masses.

pressures : `NumPy array`

NumPy array that stores pressures in dyne/cm^2.

Methods

runge_kutta_4th_step(r, dr, p, m) Implement a single step of the Runge-Kutta 4th order algorithm to integrate numerically pressure and mass equations, according to current pressure, mass, radius and radius step of integration. solve(step_r, p0) Integrate the system until pressure drops to zero given the initial pressure and radius step of integration. Store radii, masses and pressures in corresponding NumPy arrays. print_mass_radius() Print the mass and the radius of the object. get() Return NumPy arrays containing radii, masses and pressures. plot() Plot mass and radius versus pressure.

Take an equation of state and an option for relativity corrections and use them as parameters to create an instace of the class TOVSystem.

Parameters

eos : `CubicSpline`

A CubicSpline representation of the equation of state in the form of energy density as function of pressure.

relativity_corrections : `bool`

A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is used, if False the Newtonian pressure equation is used.

▸ EXPAND SOURCE CODE

Methods

```
def get(self)
```

Return NumPy arrays containing respectively radii, masses and pressures.

Returns

tuple : `[NumPy array, NumPy array, NumPy array]`

NumPy arrays of radii in km, masses in solar masses and pressures in dyne/cm^2.

Raises

`ValueError`
: No data is computed to get.

```
def plot(self)
```

Plot two subplots: pressure versus radius and mass versus radius with a grid.

Raises

`ValueError`
: No data is computed to plot.

```
def print_mass_radius(self)
```

Print the radius in km and mass in solar masses of the object, rounded to fourth significant digits.

Raises

`ValueError`
: No data is computed to print.

```
def runge_kutta_4th_step(self, r, dr, p,
m)
```

Perform a single step of the 4th-order Runge-Kutta algorithm. Given the current radius, pressure, mass and radius step of integration, compute the 4 increment coefficients of the 4th-order Runge-Kutta step, using the mass and pressure equations, and then compute and return the new calculated values of pressure and mass.

Parameters

`r` : `float`
: Current radius in cm.

`dr` : `float`
: Radius step of integration in cm.

`p` : `float`
: Current pressure in dyne/cm^2.

`m` : `float`
: Current mass in g.

Returns

`tuple` : `[float, float]`
: New calculated values of pressure and mass.

```
def solve(self, step_r, p0)
```

Perform a while loop until pressure drops to zero, which means the integration has to stop because the surface is reached. For each iteration, compute the new calculated values of radius, pressure and mass using the 4th-order Runge-Kutta step and store them in corresponding NumPy arrays. Change unit of measure of mass from g to solar masses and radius from cm to km.

Parameters

`step_r` : `float`
: Radius step of integration in cm.

`p0` : `float`
: Initial central pressure in dyne/cm^2.

Raises

`ValueError`
: If radius step is negative. If initial pressure is negative.

```
class TOVSystem (eos, relativity_corrections=True)
```

Class to represent the system of ordinary differential equations describing a spherically symmetric physical object which is in static gravitational equilibrium. The system is composed of 3 equations: the mass equation, the pressure equation and the equation of state. The pressure equation can be either Newtonian or relativistic (the latter is called properly TOV equation). The equation of state is not general, because it depends on the matter content of the object, and so it must be given as input.

Attributes

`eos` : `CubicSpline`
: A CubicSpline representation of the equation of state in the form of energy density as function of pressure.

`relativity_corrections` : `bool`
: A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is used, if False the Newtonian pressure equation is used.

Methods

check_physical_constraints(r, p, m) Control values of mass, pressure and radius, since for physical reasons they must all be positive. dmdr(r, p, m) Compute and return the mass equation according to current radius, pressure and mass. dpdr(r, p, m) Compute and return the pressure equation according to current radius, pressure, mass. It can consider relativity corrections if attribute relativity_corrections is True.

Take an equation of state and an option for relativity corrections and store them as attributes of the class.

Parameters

`eos` : `CubicSpline`
: A CubicSpline representation of the equation of state in the form of energy density as function of pressure.

`relativity_corrections` : `bool`
: A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is used, if False the Newtonian pressure equation is used.

▸ EXPAND SOURCE CODE

Methods

`def check_physical_constraints(self, r, p, m)`

Check the values of radius, pressure and mass, which must be all positive. Mass and radius are positive by definition. Pressure can be at most zero at the surface, so that when pressure is negative or zero, return False to signal that the surface of the object is reached and integration has to stop.

Parameters

r : `float`

Radius in cm.

p : `float`

Pressure in dyne/cm^2.

m : `float`

Mass in g.

Raises

`ValueError`
: If radius is negative or zero. If mass is negative or zero.

Returns

`bool`
: True if all physical constraints are satisfied, False if pressure is negative.

`def dmdr(self, r, p, m)`

Compute and return the mass equation (dm/dr) based on the current radius, pressure and mass.

Parameters

r : `float`

Radius in cm.

p : `float`

Pressure in dyne/cm^2.

m : `float`

Mass in g.

Returns

`float`
: Value of the mass equation of the system (dm/dr).

```
def dpdr(self, r, p, m)
```

Compute and return the pressure equation (dp/dr) based on the current radius, pressure and mass. If relativity corrections are included, return the TOV equation, otherwise return the Newtonian pressure equation.

Parameters

`r` : `float`
: Radius in cm.

`p` : `float`

Pressure in dyne/cm^2.
`m` : `float`
: Mass in g.

Returns

`float`
: Value of the pressure equation of the system (dp/dr).

Classes

- **`EquationOfState`**
  - `get`
  - `interpolate`
  - `load_from_file`
  - `plot`
- **`SolverTOVRangePressure`**
  - `get`
  - `plot_MRvsP`
  - `plot_MvsR`
  - `print_max_mass`
  - `solve`
- **`SolverTOVSinglePressure`**
  - `get`
  - `plot`
  - `print_mass_radius`

- runge_kutta_4th_step
- solve

- **TOVSystem**
- check_physical_constraints
- dmdr
- dpdr