Module `solver`

Classes

```
class EquationOfState
```

Class to represent an equation of state.

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 a file and store number densities, pressures and energy densities in corresponding NumPy arrays. plot() Plot energy densities versus pressure. interpolate() Interpolate using cubic spline and return the equation of state in the form of energy density as a function of pressure. get() Return NumPy arrays containing pressure and energy density.

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

▶ EXPAND SOURCE CODE

Methods

```
def get(self)
```

Return NumPy arrays containing pressures and energy densities.

Returns

`tuple`
NumPy arrays of pressures and energy densities.

```
def interpolate(self)
```

Interpolate pressures and energy densities into an equation of state using cubic spline (imported from SciPy).

Returns

`CubicSpline`
CubicSpline representation of the equation of state, which gives energy density as output once

pressure is given as input.

Raises

`ValueError`
No data is loaded to interpolate.

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

Open file given its path, read data and store quantities 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 1/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 file containing number densities, pressures and energy densities in respective columns.

Raises

`InputError`
The file cannot be opened.

`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 SolverRangePressure (eos, relativity_corrections=True)
```

Class to solve numerically the system of ordinary differential equations given a range of initial pressures. It uses the class SolverSinglePressure for each initial pressure.

Attributes

`eos` : `CubicSpline`
A CubicSpline representation of the equation of state.

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

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 until pressure drops to zero given the initial pressure and radius step. Store radii, masses and pressures in corresponding NumPy arrays. print_max_mass() Print the maximum mass, its radius and central pressure. get() Return NumPy arrays containing radius, mass and initial pressure. plot_MRvsP() Plot mass and radius versus initial pressure. plot_RvsM() Plot radius versus mass.

Take an equation of state and an option for relativity corrections and copy them to 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.

relativity_corrections : bool
A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is considered, if False the Netwonian equation is considered.

▶ EXPAND SOURCE CODE

Methods

```
def get(self)
```

Return the NumPy arrays of radius, mass and initial pressure.

Returns

tuple
NumPy array of radius in km, mass in solar masses and initial pressure in dyne/cm^2.

Raises

ValueError
No data is computed to get

```
def plot_MRvsP(self)
```

Plot mass and radius versus initial pressure. Show two subplots: radius versus initial pressure and mass versus initial pressure in logarithmic scal for the x axis with a grid.

Raises

`ValueError`
No data is computed to plot

```
def plot_RvsM(self)
```

Plot radius versus mass 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)
```

Solve the system for a range of initial pressures. Using the class SolverSinglePressure, compute the mass and the radius and store them for each initial pressure.

Parameters

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

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

Raises

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

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

Class to solve numerically the system of ordinary differential equations given a single initial pressure. Implement the 4th-order Runge-Kutta with a breaking condition when the pressure becomes zero or negative, because it means that the surface of the object is reached.

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 the Runge-Kutta 4th order algorithm for a single step to solve numerically a system of two differential equations, according to current pressure, mass, radius and radius step. solve(step_r, p0) Solve the system until pressure drops to zero given the initial pressure and radius step. Store radii, masses and pressures in corresponding NumPy arrays. print_mass_radius() Print the mass and the radius of the object. get() Return radius, mass and pressure NumPy arrays. plot() Plot mass and radius versus pressure.

Initialize the class TOVSystem with an equation of state and an option for relativity corrections.

Parameters

eos : CubicSpline
A CubicSpline representation of the equation of state.

relativity_corrections : bool
A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is considered, if False the Netwonian equation is considered.

▶ EXPAND SOURCE CODE

Methods

```
def get(self)
```

Return radius, mass and pressure NumPy arrays.

Returns

tuple
NumPy array of radius in km, mass in solar masses and pressures in dyne/cm^2.

Raises

ValueError
No data is computed to get.

```
def plot(self)
```

Plot mass and radius versus pressure. Show two subplots: radius versus pressure and mass versus

pressure 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, compute the 4 coefficients of the algorithm, using the equations in the methods of TOVSystem, and then compute and return the new calculated value of pressure and mass.

Parameters

r : `float`
Current radius in cm.

dr : `float`
Radius step in cm.

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

m : `float`
Current mass in g.

Returns

`tuple`
NumPy arrays of pressures and masses.

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

Apply the 4th-order Runge-Kutta step using a while loop for the given initial pressure until pressure drops to zero. For every iteration, update the radius, pressure and mass values in the 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 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 for a spherically symmetric physical object which is in static gravitational equilibrium. The system is composed in 3 equations: the mass equation, the pressure equation and the equation of state. The pressure equation can be either Newtonian or relativistic (the latter called properly TOV equation).

Attributes

eos : CubicSpline
A CubicSpline representation of the equation of state.

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 equation is used.

Methods

check_physical_constraints(r, p, m) Control if physical quantities, like mass, pressure and radius, are positive. dmdr(r, p, m) Compute and return the mass equation accordingly to current radius, pressure and mass. dpdr(r, p, m) Compute and return the pressure equation accordingly 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 copy them as attributes of the class.

Parameters: eos : EquationOfState A CubicSpline representation of the equation of state. relativity_corrections : bool A Boolean variable to determined whether to include general relativity corrections. If True the TOV equation is used, if False the Netwonian equation is used.

▶ EXPAND SOURCE CODE

Methods

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

Check the physical constraints for radius, pressure and mass, which must be all positive. When pressure is negative, 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. If mass is negative.

Returns

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

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

Compute 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`
: Mass equation of the system (dm/dr).

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

Compute 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`
Pressure equation of the system (dp/dr).

Classes

- **EquationOfState**
- get
- interpolate
- load_from_file
- plot
- **SolverRangePressure**
- get
- plot_MRvsP
- plot_RvsM
- print_max_mass
- solve
- **SolverSinglePressure**
- get
- plot
- print_mass_radius
- runge_kutta_4th_step
- solve
- **TOVSystem**
- check_physical_constraints
- dmdr
- dpdr