CODE GUIDE

Numerically Solving the Tolman-Oppenheimer-Volkoff Equations to Model the Internal Structure of Neutron Stars

Mauritz Wicker

Ruprecht Karl University of Heidelberg
Department of Physics and Astronomy
Numerical Techniques for Modeling Relativistic Hydrodynamics

January 30th 2022

Abstract

This is a code guide to accompany the Github-Repository (mauritzwicker/tov-solver) for the **'tov-solver'**, a python based solver of the Tolman-Oppenheim-Volkoff Equations. An overview of the repository, as well as explanations for objects, functions, and parameters used in the code are given in this guide. Finally, outputs and possible analysis-tools are discussed.

Contents

1 Introduction

The Tolman-Oppenheimer-Volkoff (TOV) equations describe the internal structure of a spherically symmetric body that is in gravitational equilibrium. Derived in a general relativistic framework, it can be used to model the interior of neutron stars.

The investigation of the TOV-equations looks at the use of a numerical, python based TOV equation solver to model the interior of a spherical body and constrain neutron star parameters. Neutron stars are the result of a collapsed supergiant star, and classified as small, dense, and static objects. Neutron star masses are in the range of $1.0 M_\odot$ to $2.5 M_\odot$ and a radii of the order of 10 km. The TOV equations, solved numerically, explain the step by step profile of the object's pressure and mass. This profile is heavily dependent on the central density of the star, a quantity that sets the internal pressure at the core.

Along with this code guide, a report to accompany the TOV-solver is submitted. It can be found in the Github-directory in the folder called 'others'. The code for this project is publicly available in a Github-repository and can adapted for further investigations:
https://github.com/mauritzwicker/tov-solver

The 'tov-solver' is published with an MIT License.

2 Code Overview

The code for the 'tov-solver' is written in object-oriented python to make the code easy to ready and adopt for other situations. Key functions, such as the Runge-Kutta iterations are written to only pass essential variables, thereby trying to reduce the computational strain.

2.1 High Level View

The 'tov-solver' is run through six independent python files. These files contains objects and functions that are called on throughout the simulations. These files are:

- main.py
  - This is the main file to run the simulation.
- constant.py
  - This is to hold the Constants object, which is a collection of constant values.
- default_input.py
  - This is to hold the DefaultInput object, which is the default-inputs.
- user_input.py
  - This is to hold the UserInput object, which is the user-inputs.
- tovmodel.py
  - This is to hold the TOVmodel object, which hold the simulation variables.
- functions.py
  - This is for the functions used in the analysis.

Each run of the solver is orchestrated by the TOV-Model object, as seen in Figure 1. This object get the necessary constants, default variables, and user-defined variables to run the Runge-Kutta methods. The mechanism with with the code runs is as follows:

1. main function is called.
2. Constants object is initiated.
3. DefaultInput object is initiated.
4. UserInput object is initiated.

5. UserInput object is checked to make sure simulation runs smoothly (if user selects to do so).
6. TOVmodel object is created.
7. Runge-Kutta method function is called, starting our tov-solver simulation.
8. Once the simulation has finished (because Star-Mass/Radius is reached or because the grid has run through) the results are saved/printed/plotted.

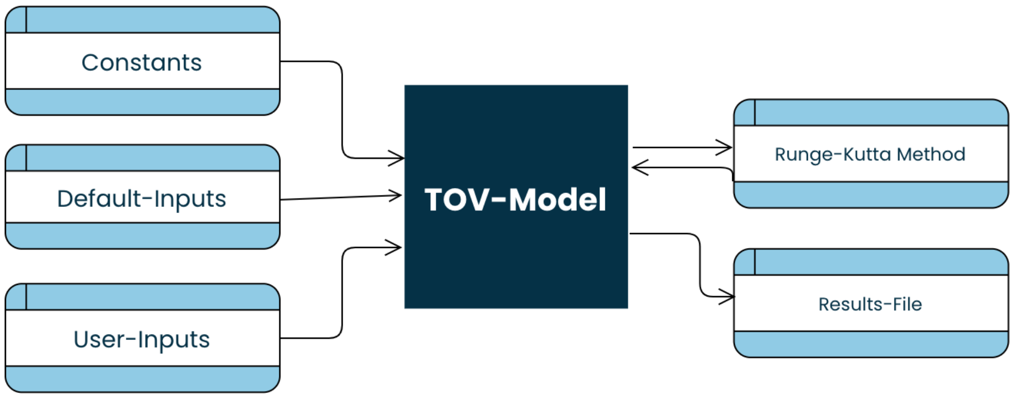


Figure 1: Code-Overview

2.2 Key Objects and Functions

Key Objects

The main objects used are defined below.

```
class Constants
```

Object to hold constants

```
class DefaultInput
```

Object to hold Default Inputs for simulation

```
class UserInput
```

Object to hold User Inputs for simulation `class TOVmodel`

Object for a TOV-model and to run RK-methods
- *Inputs*:
  usr := the check/unchecked UserInput object
  const := the Constants object

Key Functions

The functions, used to check-inputs, run the Runge-Kutta methods, and determine next-step solutions to the TOV-equations are defined below.

`def check_userinput`

Function to check user inputs (if set to do so) and assign default values if these inputs could cause errors.
- *Inputs*:
  const := constant values object
  defa := default values objects
  usr := usr values object
- *Returns/Outputs*:
  usr := usr := checked/unchecked and default assigned user values

`def eos`

Function for equation of state. For now set to work only for Polytropic-Equation-of-State, but can easily be adopted to include more.
- *Inputs*:
  rho := density value
  type := EoS type (default = 'poly')
  K := adiabatic constant (default = 1.982e-6)
  gamma := adiabatic Exponent (default = 2.75)
- *Returns/Outputs*:
  Pressure, calculated from EoS

`def eos_reversed`

Function for reversed equation of state. For now set to work only for Polytropic-Equation-of-State, but can easily be adopted to include more.
- *Inputs*:
  p := pressure value
  type := EoS type (default = 'poly')
  K := adiabatic constant (default = 1.982e-6)
  gamma := adiabatic Exponent (default = 2.75)
- *Returns/Outputs*:
  Density, calculated from EoS

`def eps`

Function to determine epsilon (specific internal energy).

- *Inputs*:
  p := pressure value
  rho := density value
  type := EoS type (default = 'poly')
  gamma := adiabatic Exponent (default = 2.75)
- *Returns/Outputs*:
  Specific-Internal-Energy, calculated from EoS

`def dpdr`

Function for TOV-Equation for Pressure.

- *Inputs*:
  r := position/radius from center value
  p := pressure value
  rho := density value
  m := mass value
  eps := specific internal energy (default = 0)
  c := speed of light (default = 3e10)
  G := Gravitational Constant (default = 6.6742e-8)
- *Returns/Outputs*:
  Next increment of TOV-Pressure-Equation $\frac{dp}{dr}$

`def dmdr`

Function for TOV-Equation for Mass.

- *Inputs*:
  r := position/radius from center value
  rho := density value
  eps := specific internal energy (default = 0)
  c := speed of light (default = 3e10)
- *Returns/Outputs*:
  Next increment of TOV-Mass-Equation $\frac{dm}{dr}$

`def dpdr_NEWT`

Function for TOV-Equation for Pressure in Newtonian limit.

- *Inputs*:
  r := position/radius from center value
  p := pressure value
  rho := density value
  m := mass value
  eps := specific internal energy (default = 0)
  c := speed of light (default = 3e10)
  G := Gravitational Constant (default = 6.6742e-8)
- *Returns/Outputs*:
  Next increment of TOV-Pressure-Equation $\frac{dp}{dr}$

`def dmdr_NEWT`

Function for TOV-Equation for Mass in Newtonian limit.
- *Inputs*:
  - r := position/radius from center value
  - rho := density value
  - eps := specific internal energy (default = 0)
  - c := speed of light (default = 3e10)
- *Returns/Outputs*:
  - Next increment of TOV-Mass-Equation $\frac{dm}{dr}$

`def check_Sim_Done`

Function to check if the Mass of the Star has been reached at the end of the simulation.
- *Inputs*:
  - tov := tov object
  - usr := user-input object
- *Returns/Outputs*:
  - tov := tov object with check for a completed simulation

`def check_Sim_Done`

Function to check if the Mass of the Star has been reached at each step -¿ defined Mstar.
- *Inputs*:
  - tov := tov object
  - usr := user-input object
  - i := the index of simulation we are at
- *Returns/Outputs*:
  - tov := tov object with check for a completed simulation at i

`def radius_Schwarzschild`

Function to determine Schwarzschild radius for specific mass
- *Inputs*:
  - G := gravitational constant
  - m := cummulative mass at specific radius
  - c := speed of light
- *Returns/Outputs*:
  - schwarzschild radius

`def rk1`

Function to perform 1-Order Runge-Kutta (Euler) Method
- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  assigns variable values in tov (TOVmodel object)

`def rk2`

Function to perform 2-Order Runge-Kutta (Euler) Method
- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  assigns variable values in tov (TOVmodel object)

`def rk3`

Function to perform 3-Order Runge-Kutta (Euler) Method
- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  assigns variable values in tov (TOVmodel object)

`def rk4`

Function to perform 4-Order Runge-Kutta (Euler) Method
- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  assigns variable values in tov (TOVmodel object)

`def rk1_incl_epsilon`

Function to perform 1-Order Runge-Kutta (Euler) Method considering Epsilon (Specific-Internal-Energy).
- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  assigns variable values in tov (TOVmodel object)

`def rk1_newtonian`

Function to perform 1-Order Runge-Kutta (Euler) Method in Newtonian limit.

- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  assigns variable values in tov (TOVmodel object)

`def save_res`

Function to save results from RK-run.

- *Inputs*:
  usr1 := user input Object
  const := constants Object
  tov1 := TOVmodel Object
- *Returns/Outputs*:
  Saves pickle file with results into directory set in usr1(UserInput-object)

`def graph_res`

Function to graph results from RK-run. Will plot the of mass, pressure, and density profiles for the run.

- *Inputs*:
  usr1 := user input Object
  const := constants Object
  tov1 := TOVmodel Object
- *Returns/Outputs*:
  Displays plots with results from RK-run.

`def run_results_output`

Function to print (and coordinate saving/plotting) the results of the RK-run in terminal.

- *Inputs*:
  usr := user input Object
  const := constants Object
  tov := TOVmodel Object
- *Returns/Outputs*:
  Displays results (Mass, Radius, Simulation-Steps) of the simulation, if assigned to do so in user-inputs.

3 Model Parameters

Here we can see the different types of parameters for the model, both physical and computational. Along with this we will show the default values for these parameters. Lastly, we will show which parameters are user definable and what reasonable values are. These parameters are defined in the UserInput and DefaultInput objects.

3.1 Physical Parameters

Physical parameters, used to numerically solve the TOV-equations, are defined below.

Equation of State

```
eos_set = 'poly'
```

Defines the equation of state used. For now the code only runs for Polytropic Equations-of-State; however can be modified easily to include others.
Default: 'poly' := polytropic EoS

Adiabatic Constant

```
K = 1.982e-6
```

Defines the adiabatic constant for the Polytropic Equation-of-State.
Default: 1.982e-6 := sensible choice for a neutron star in cgs-units.

Adiabatic Coefficient

```
gamma = 2.75
```

Defines the adiabatic coefficient for the Polytropic Equation-of-State.
Default: 2.75 := sensible choice for a neutron star in cgs-units.

Central-Density of Star

```
rho_c = 2 * const.rho_nuc
```

Defines the central density of the neutron star (ie. at the first simulation step r0).
Default: 2 const.rho_nuc := sensible choice for a neutron star in cgs-units. (const.rho_nuc := nuclear density $(2.7e14\frac{g}{cm3})$)

Central-Pressure of Star

```
p_c = 0
```

Defines the central pressure of the neutron star (ie. at the first simulation step r0).
Default: 0 := set to zero as it is defined by the equation-of-state from the central density.

Central-Specific-Internal-Energy of Star

```
eps_c = 0
```

Defines the central specific-internal-energy of the neutron star (ie. at the first simulation step r0).
Default: 0 := set to zero as it is defined by the equation-of-state from the central density.

Central-Metric-Potential of Star

```
phi_c = 0
```

Defines the central metric-potential of the neutron star (ie. at the first simulation step r0).
Default: 0 := set to zero as it is defined by the equation-of-state from the central density and pressure.

Central-Mass of Star

```
m_c = 0
```

Defines the central mass of the neutron star (ie. at the first simulation step r0).
Default: 0 := set to zero as it is defined by the central-pressure and density.

Central-Radius of Star

```
r0 = 0
```

Defines the central radius of the neutron star (ie. at the first simulation step).
Default: 1e-8 := set close to zero; however tests showed that values smaller than the step-size produce precise results.

3.2 Computational Parameters

Computational parameters, used to define for how long and how accurately the simulations are run, are defined below.

Grid-size

```
grid = int(4e6)
```

Defines the grid-size of the simulation. A larger grid enables a smaller step-size but also increases computational time.
Default: int(4e6) := set so that for a step-size of 100cm the limit is reached for relevant input-parameters.

Step-size

```
stepsize = 100 #cm
```

Defines the step-size of the simulation. A smaller step-size increase simulation accuracy but also increases computational time.
Default: 100 := a step-size that produces accurate results without being too computationally costly.

Runge-Kutta Method

```
rk_method = 'rk4'
```

Defines the Runge-Kutta method used for the simulation. Can be either of the following: ['rk1', 'rk2', 'rk3', 'rk4']
Default: 'rk4' := produces the most accurate results, but also is most computationally costly.

Break Mass

```
m_break = 100 * self.const.M_sun # g
```

Defines the mass at which, if the simulation is not done yet (pressure is greater than zero), to quit the simulation. This is to not end up in long simulation loops, and to instead quit the run.
Default: 1000 * const.M_sun := most likely we want to set the initial-parameters so that the maximum is not reached, so m_break should be large enough that it is not easily reached.

Break Radius

```
r_break = 1 * self.const.R_sun
```

Defines the radius at which, if the simulation is not done yet (pressure is greater than zero), to quit the simulation. This is to not end up in long simulation loops, and to instead quit the run.
Default: 1000 * const.R_sun := most likely we want to set the initial-parameters so that the maximum is not reached, so r_break should be large enough that it is not easily reached.

3.3 User-Defined Parameters

User defined parameters, other than the numerical ones which a user can change depending on the initial simulation parameters they want, allow the user to decide whether input-values are checked and what the output of the simulation should be.

Use Default or User-Input

```
use_default = False
```

Defines whether to use the user-input information or the default values held by the DefaultInput object.
Default: False := most likely the user wants to define their own input variables.

Check User-Inputs

```
check_inputs = True
```

Defines whether to check the user-inputs and define default values in case input values will cause problems. It will also check the grid-size and step-size to warn from costly-computations.
Default: True := it is recommened to check this to not have problems with the simulation.

Show Graphs after Run

```
show_graph_res = False
```

Defines whether to show the graphs of the mass, pressure, and density profiles at the end of the run.
Default: False := most likely a user will want the raw-data.

Save Results

```
save_results = True
```

Defines whether to save the raw data or not (it will be saved as a python-dictionary in a pickle file).
Default: True := most likely a user will want to saw raw-data.

Save Path

```
save_path = './results/'
```

Defines where to save the raw-data.
Default: './results/' := if no path is defined, a new directory called 'results' will be created and data will be saved there.

Save Name

```
save_name = 'tov-solver_sim1'
```

Defines under what name to save the raw-data.
Default: 'tov-solver_sim1' := if no name is defined, this name is choosen out of simplicity.

Save Over Existing Files

```
save_over = False
```

Defines whether or not to replace older files with the same name. A user will be warned of this at the end of the simulation when saving, however it should be defined beforehand.
Default: False := most likely a user will not want to delete older files.

4 Output

Here we will look at what will output when the code is run. This is user defined and can also be adapted manually.

4.1 Raw-Output

The user-inputs, constants, and tov-object are saved as a python dictionary in a pickle file. Additionally, a key is added to the dictionary to hold the most important results. This dict called 'FinalResults', holds the Star-Mass (in Solar-Masses), Star-Radius (in kilometers) and number of simulation steps. Storing the data in a pickle file is easy and efficient; however the filetype can easily be adapoted in the save_res() function in functions.py. The user can define where and under what name the files are saved.

4.2 Analysis and Plotting

There is a small Analysis-Package in the tov-solver repository which consists of four files that can be used to analyze the pickle files created.
(https://github.com/mauritzwicker/tov-solver/analysis/) It is written to produce clean, simple, and easy to read plots for the user. There is also the possibility to include user-written functions to plot other parameters of interest. The four files of the analysis-package are:

- run_analysis.py
  - This is to run the analysis.
- analysis_user_input.py
  - This is for the user to set what to analyze.
- results_loaded.py
  - This is to hold the ResultsLoaded object, which is an object for each of the pickle result files to analyze.
- analysisfunctions.py
  - This is for the functions used in the analysis.

The Analysis works so that the user defines which files to analyze (by defining their paths) and how to analyze them. This is done by selecting with functions are run in the analysis_user_input.py file. The options are defined below.

```
def print_results
```

This is to simply print the results, in the form of the Star-Mass [M_sun], Star-Radius [km], and Simulation-Steps, for each simulation.

`def plot_MProfiles`

This is to plot the Mass-Profile for each simulation.

`def plot_PProfiles`

This is to plot the Pressure-Profile for each simulation.

`def plot_RhoProfiles`

This is to plot the Density-Profile for each simulation.

`def plot_MPRhoProfiles`

This is to plot the Mass,Pressure,Density-Profiles for each simulation in subplots.

`def plot_MRcentralrho`

This is to plot the Star-Mass and Star-Radius as the defined central-density for each simulation.

`def plot_MRRelation`

This is to plot the Star-Mass and Star-Radius relation.

`def analyseDataSelf`

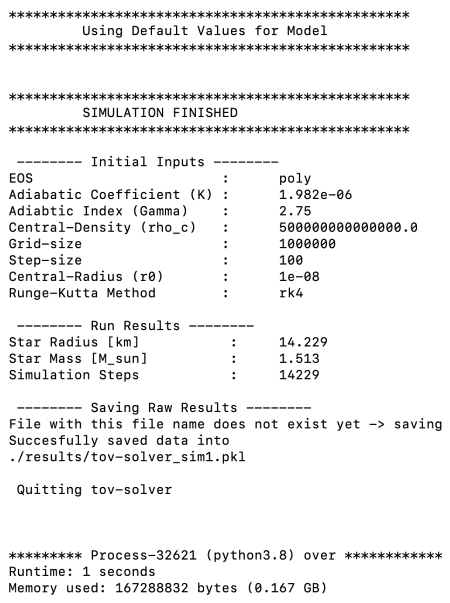
This is an example function for a user to call if they want to do their own analysis.

5 Example Usage

We will now look at an example of running the 'tov-solver'. This example uses the default-input paramters as defined in the DefaultInput object.
The terminal-command to run the 'tov-solver' is:

python ./main.py

The terminal-output for running the 'tov-solver' is as follows:



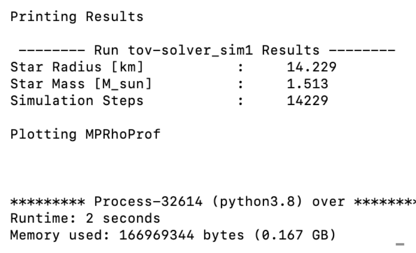
```
************************************************
         Using Default Values for Model
************************************************


************************************************
         SIMULATION FINISHED
************************************************

 -------- Initial Inputs --------
EOS                        :      poly
Adiabatic Coefficient (K) :      1.982e-06
Adiabtic Index (Gamma)    :      2.75
Central-Density (rho_c)   :      500000000000000.0
Grid-size                 :      1000000
Step-size                 :      100
Central-Radius (r0)       :      1e-08
Runge-Kutta Method        :      rk4

 -------- Run Results --------
Star Radius [km]           :     14.229
Star Mass [M_sun]          :     1.513
Simulation Steps           :     14229

 -------- Saving Raw Results --------
File with this file name does not exist yet -> saving
Succesfully saved data into
./results/tov-solver_sim1.pkl

 Quitting tov-solver



********* Process-32621 (python3.8) over ************
Runtime: 1 seconds
Memory used: 167288832 bytes (0.167 GB)
```

Figure 2: Terminal-Output for Example-Run

To analyse the results, one has to enter the 'analysis'-directory and define the analysis-parameters wanted. For this example we have selected to print the results of the simulation and to plot the mass, pressure, and density profiles in a combined figure.
Once in the 'analysis'-directory, the terminal-command to run the 'tov-solver'-analysis is:

python ./run_analysis.py

The terminal-output and plots for running the 'tov-solver'-analysis with our inputs is as follows:



```
Printing Results

 -------- Run tov-solver_sim1 Results --------
Star Radius [km]            :     14.229
Star Mass [M_sun]           :     1.513
Simulation Steps            :     14229

Plotting MPRhoProf



********* Process-32614 (python3.8) over ************
Runtime: 2 seconds
Memory used: 166969344 bytes (0.167 GB)
```

Figure 3: Terminal-Output for Example-Analysis

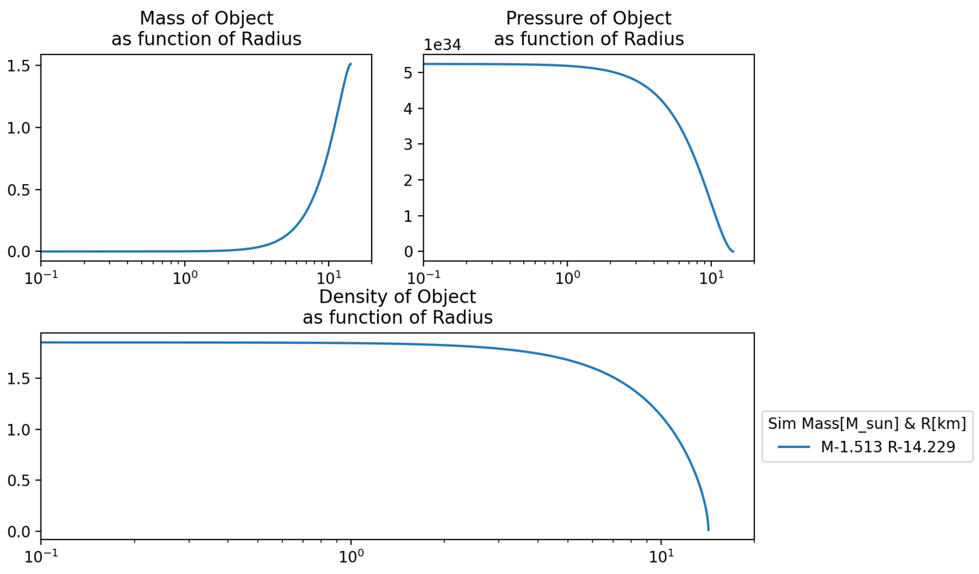


Figure 4: Example-Run Mass/Pressure/Density Profile

Keep in mind that the analysis can also be done for multiple models. By simply adding the filenames in the user-input file for analysis. The plots then look as follows.

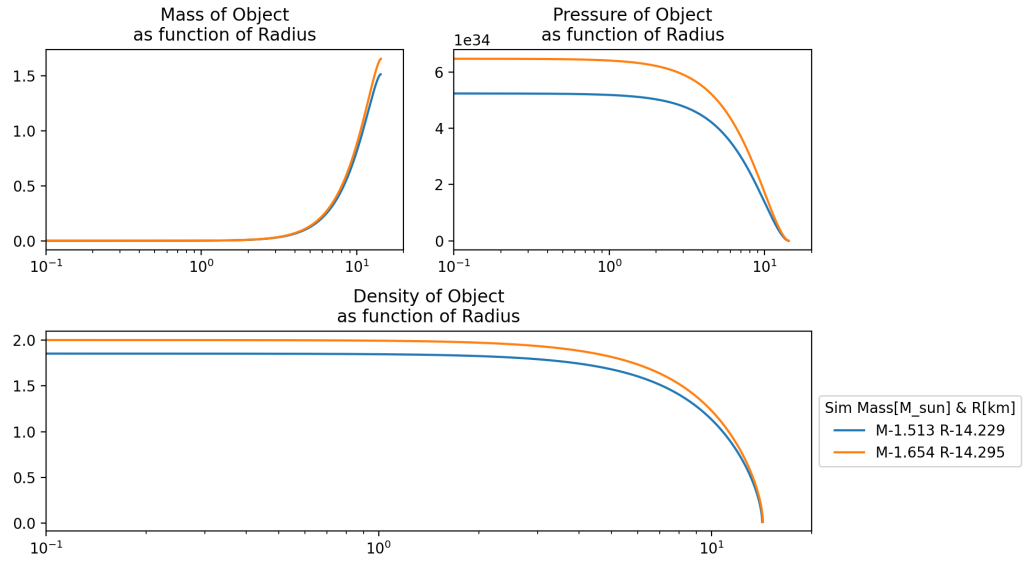


Figure 5: Example-Run Mass/Pressure/Density Profile for 2-Simulations

Links

https://github.com/mauritzwicker/tov-solver