Documentation of PyUltraLight_SI

Matthias Stallovits

September 2024

Prolog

This is the documentation of the PyUltraLight_SI-package, which is an upgraded form of the PyUltraLight-package. The original PyUltraLight-package was developed by Faber Edwards, Emily Kendall, Shaun Hotchkiss, and Richard Easther and was published together with their paper.[1] To understand the new version, it is recommended to look at their paper, since not everything will be explained here, since the most stays the same. Therefore, we will focus on the specific upgrades that were developed to improve the original package. To see some details on the development and the first results using this package, look at the paper for which this version was created.[2] All the major components of the package can be found in the *PyUltraLight_SI-master*-directory. It should include the following files:

- *.ipynb_checkpoints* (directory)
- *__pychache__* (directory)
- *Soliton Profile Files* (directory)
  - *soliton_solution.py* (Python-file)
  - *Info_soliton_solution.txt* (text-document)
- *TestOutput* (directory)
- *Visualisations* (directory)
- *.gitignore* (text-document)
- *density_profile_V4.py* (Python-file)
- *Documentation_of_PyUltraLight_SI.pdf* (pdf-file)
- *LICENSE.md* (text-document)
- *PyUltraLight_SI.py* (Python-file)
- *PyUltraLight_notebook_V2.ipynb* (Jupyter-Python-notebook)
- *README.md* (text-document)

Note, that in the *Soliton Profile Files*-directory, there is an additional Python-file. Again, we will focus mainly on the changes made to the original package, but we will also include a short explanation of the major components. If you are looking for explanations of the algorithm or the mathematics behind it, it is recommended to look at the original or our paper.

[1] For the paper, please look here: `https://arxiv.org/pdf/1807.04037.pdf`
[2] The paper can be found here: `https://arxiv.org/pdf/2406.07419.pdf`

Short explanations of the components in the package

Here we will give a short explanation of all the components in the package. For a little more detail on the major components, see below.

- *.ipynb_checkpoints*: For saving the current state of the notebook.
- *__pychache__*: The cache files for the notebook.
- *Soliton Profile Files*: The directory where the initial soliton profiles are saved. Keep in mind that the code loads the initial soliton profiles from this directory. Also, this directory includes the following Python- and text-file:
  - *soliton_solution.py*: The Python-file that generates the initial soliton profiles.
  - *Info_soliton_solution.txt*: A text file that stores some crucial information about the last run with *soliton_solution.py*. It is also necessary for the functionality of the main code.
- *TestOutput*: The directory where simulation data is saved.
- *Visualisations*: The directory where the visualizations of the simulation are saved.
- *.gitignore*, *LICENSE.md* and *README.md*: Basic information on the original package, the license and some prerequisites.
- *density_profile_V4.py*: The Python-file that generates the radial density profiles from the simulation data, fits it and calculates the total mass by integrating the density profile.
- *Documentation_of_PyUltraLight_SI.pdf*: This PDF you are currently reading.
- *PyUltraLight_SI.py*: The main code that evolves the initial soliton profiles with the user-specific inputs.
- *PyUltraLight_notebook_V2.ipynb*: The notebook, where nearly all inputs from the user can be inserted. Note, that there are some specific variables that have to be changed in the other main code files.

Bug fixing and other small changes

Since the solver was originally written for a Linux-operating system, some changes had to be made such that PyUltraLight_SI works also on Windows. For that, line 385 in *PyUltraLight_SI.py* had been modified. In this line, the name of the timestamp is coded. In the original version, the time (hours, minutes, and seconds), which is written in the name of the directory for each simulation output, is separated by a ":"-sign. However, this sign is not allowed in filenames on Windows. Therefore, the separation-sign was changed to "h" and "m" (h=hours, m=minutes) to indicate the hour and minute of the current time of the simulation. The seconds do not need a separation-sign because, after the seconds, the resolution is written down with a "_" sign as a separation-sign. Due to this change, PyUltraLight_SI should now also work on all operating systems.

Also, one bug was found in the Jupyter notebook that comes with the original PyUltraLight-package. In the section where the visualizations were saved, there is an if-statement that saves the animation and gives the created file a name. This name corresponds to the completed simulation. The decision, whose name is chosen, depends on the save-variable, which also specifies the kind of simulation. In the original Jupyter notebook that comes with the PyUltraLight-package, the if-statement to save the animation as HTML has for both possible options the same save variable, namely "1". This is the correct variable to save the plane-animation, but for the line-animation, the correct variable is "3". Therefore, the variable for the line-option was changed to "3" to fix the bug.

Of course, there were also some changes made, where customization was possible. For example, for figure sizes or font sizes, but since this is user-specific, we will not talk about that here. The other changes that were made mostly include completely new code, which is described below.

soliton_ solution

Small explanation

This file generates the initial soliton profile, which is then loaded and evolved with the algorithm in PyUltraLight_SI. The initial soliton profiles are calculated by imposing spherical symmetry and time independence on the coupled differential equation system. This system is the Schrödinger-Poisson-system in the case without self-interaction and the Gross-Pitaevskii-Poisson-system in the case with self-interaction. From there, a simplified system of differential equations can be derived, which are then solved with a 4. order Runge-Kutta and the shooting method. If one executes the code, a file will be generated that contains the initial soliton profile as well as a short information text-file.

Changes

Besides the fact that this file is now in the *Soliton Profile Files*-directory, there are only three new things in comparison to the original: First, the inclusion of the self-interaction in the equations; second, a variable to choose the dimensionless coupling strength; and third, the creation of a text-file, which contains some information about the initial profile. The mentioned variable is called *Lambda_ hat*. Since, when the file is executed, the resulting initial soliton profile is specific for the chosen *Lambda_ hat*, one has to change the *Lambda_ hat*-value before executing the file.

The change in the equations was made in the function *g1*, since the function *g2* describes the reduced Poisson-equation, which does not change when self-interaction is included. Thus, only the *g1* function, which is the equation of motion, had to be modified. To get from the Schrödinger-equation to the Gross-Pitaveskii-equation one can follow the same steps as described in the third section of the original PyUltraLight-paper (see footnote 1) or in detail in the development-paper for this upgraded package (see footnote 2). In the end, one will get one additional term, which is *Lambda_ hat*a**3*. This is also the only change that has to be made, to change the reduced Schrödinger-equation into the reduced Gross-Pitaevskii-equation.

The dimensionless coupling constant *Lambda_ hat* comes from the coupling constant g, which is defined with a scattering length a_s as

$$g = 4\pi\hbar^2 \frac{a_s}{m}, \tag{1}$$

where $\hbar$ is the reduced Planck's constant and m is the axion mass. The scattering length is derived from the scattering cross-section σ_s for elastic scattering of indistinguishable particles, which is defined by

$$\sigma_s = 8\pi a_s^2. \tag{2}$$

If the g is positive, the self-interaction is repulsive, while if g is negative, then we have an attractive interaction. The same goes for the dimensionless coupling strength $\hat{\Lambda}$ (*Lambda_ hat* in the code), which is defined as

$$\hat{\Lambda} = \frac{c^2 g}{4\pi G\hbar^2}. \tag{3}$$

Here is c speed of light in vacuum and G the gravitational constant.

The last change that was made was the generation of a text-file named *Info_ soliton_ solution.txt*. This file solves two issues that the original package had. First, it saves the information, which *Lambda_ hat* was used in the last run of *soliton_ solution*. This is useful since the initial profile file *initial_ fSI.npy* does not tell which self-coupling strength was used. Thus, by the text-file, one can tell what was the last *Lambda_ hat* used, or rather, with which *Lambda_ hat* the current initial soliton profile was calculated. Also, the printed statements from code, that are otherwise lost after closing the console, are saved in this file.

The second issue that this text-file solves is the β-parameter and $\hat{\Lambda}$. The β-parameter is a constant that is needed for the initialization of the solitons in the main code. However, this constant depends on *Lambda_ hat* and must therefore be changed before each run with the main code. Since the original package did not include self-interaction, the *beta*-parameter was hard coded by setting a variable to the value for *Lambda_ hat* $= 0$. If one wished to change *Lambda_ hat* one had to change *beta* as well. The text-file, however, saves the *beta*-parameter and *Lambda_ hat*, which can then be read by the main code, thus avoiding any more input from the user after the usage of *soliton_ solution.py*.

Keep in mind, that after each run of *soliton_ solution.py* the initial profile file *initial_ fSI.npy* as well as the information text-file *Info_ soliton_ solution.txt* are overwritten.

PyUltraLight_SI

Small explanation

This is the main Python code, which handles everything, besides the initial soliton profiles and the radial density profiles. Here, the user specific inputs were combined with the initial soliton profile, which is loaded from the *Soliton Profile Files*-directory. This then get evolved by using a symmetric pseudospectral Fourier-split-step-algorithm. The results of that are saved and displayed in the notebook, based on the input the user gave.

Changes

One change that has to be explained is the name. The name was inspired by a paper by Noah Glennon and Chanda Prescod-Weinstein from 2021. They also developed a version of PyUltraLight with self-interaction. Their version was called PySiUltraLight, but since the version described here is not the same, but the main compartments of the main algorithm the same, we decided to give it a different but similar name. Hence, PyUltraLight_SI.

Another change is the automatic adaption of the *beta*-parameter and *Lambda_hat* (*Lambda_hat* is, of course, completely new). As mentioned above, the *beta*-parameter was initially set to fix value, which corresponds to the case $Lambda\_hat = 0$. The updated version now has some additional lines of code, which read the *beta*-parameter and *Lambda_hat* from the text-file in the *Soliton Profile Files*-directory, which was created by *soliton_solution.py*. Thus, a manual change of both variables is not necessary anymore.

The only other changes that were made, concern the self-interaction in the equation. Since the code reads the dimensionless self-interacting coupling strength *Lambda_hat* from the text-file, *Lambda_hat* is set after a run of *soliton_solution.py*. The lines of code that read *beta* and *Lambda_hat* are at the top of the script under the section for defining constants. *Lambda_hat* is an important parameter for the simulations, thus it was also included as information for the *config.txt*-file, which saves the current simulation parameters of each run.

The self-interaction is also included as an additional term in the dimensionless Gross-Pitaevskii-equation. This dimensionless form looks like this:

$$i\frac{\partial\hat{\psi}(\vec{x},t)}{\partial t} = -\frac{1}{2}\Delta\hat{\psi}(\vec{x},t) + \hat{\Phi}(\vec{x},t)\hat{\psi}(\vec{x},t) + \hat{\Lambda}|\hat{\psi}(\vec{x},t)|^2\hat{\psi}(\vec{x},t). \tag{4}$$

The hat only means, that these are dimensionless quantities. ψ is the wave function and Φ the gravitational potential. Besides the last term on the right-hand side, the equation is identical to the Schrödinger equation. Thus, only this last term had to be included to upgrade the code to be able to simulate self-interaction. Therefore, the temporal evolution of the differential equations looks like this:

$$\psi(\vec{x},t+h) = \exp\left[-\frac{ih}{2}\Phi(\vec{x},t+h)\right]\exp\left[-\frac{ih\hat{\Lambda}}{2}|\psi(\vec{x},t+h)|^2\right]$$
$$\mathcal{F}^{-1}\exp\left[-\frac{ih}{2}k^2\right]\mathcal{F}\exp\left[-\frac{ih}{2}\Phi(\vec{x},t)\right]\exp\left[-\frac{ih\hat{\Lambda}}{2}|\psi(\vec{x},t)|^2\right], \tag{5}$$

$$\Phi(\vec{x},t+h) = \mathcal{F}^{-1}\left(-\frac{1}{k^2}\right)\mathcal{F}4\pi|\psi(\vec{x},t_i)|^2. \tag{6}$$

Here $\mathcal{F}$ stands for the Fourier transformation, k is the wave number and h is the time step. In the code, however, there is a simplification implemented for the Gross-Pitaevskii equation. Since the last half step of an iteration and the first half step of the next iteration are the same, these steps can be combined:

$$\psi(t+nh) = \exp\left[+\frac{ih}{2}\Phi\right]\exp\left[+\frac{ih}{2}\hat{\Lambda}|\psi(t)|^2\right]\left(\prod^{n}\exp[-ih\Phi]\exp[-ih\hat{\Lambda}|\psi(t)|^2]\exp\left[\frac{ih}{2}\nabla^2\right]\right)$$
$$\exp\left[-\frac{ih}{2}\Phi\right]\exp\left[-\frac{ih}{2}\hat{\Lambda}|\psi(t)|^2\right]\psi(t). \tag{7}$$

Do not get confused by the plus sign for the first term. This sign has to be included so that the mathematical expression above is correct. In the code itself, there is no positive term, since we can add the last half step in the last iteration.[3]

[3] To represent in numbers: first we do a $\frac{1}{2}$-step, then a whole step, which makes a $\frac{3}{2}$-step. Thus, we have to subtract a $\frac{1}{2}$-step to get to 1 again. Since the terms are negative, we add, of course, the half step in the end. In the code, however, we can stop the whole steps before the end and just make a finishing half step at the end, thus, we do not have to add/subtract a half step in the end.

Last note: If one wishes to change *Lambda_hat* or the axion mass, then the axion mass has to be changed here in the main code, while a change of *Lambda_hat* requires another run with *soliton_solution.py*, where *Lambda_hat* can be specified.

DENSITY_PROFILE_V4

This part of the package is completely new. As indicated by the "V4", there were other versions before, specifically V1, which was public. But this version, 4 is the latest and newest one.

It was designed to use the simulation data from the main simulation, transform it into a radial density profile and fit it with appropriate models. This is interesting, since ultralight dark matter is known for solving the core/cusp-problem. However, DENSITY_PROFILE_V4 was specifically created to analyze the radial density profiles of mergers of solitons (a single object) and single solitons. It is not suited for simulations, where two or more solitons are or stay separated in the box.

The code works nearly completely automatically and can be controlled from the notebook, which will be explained in the next section. Only the axion mass has to be adjusted when it changes from the default value of 10^{-22} eV. Other than that, everything can be adjusted in the notebook. It should be noted that the code can only process the three-dimensional density data from the simulations. Therefore, it is necessary for the code, that these data is saved, which can be done in the notebook by setting the variable *save_rho* to *True*.

But since this code is completely new for the package, we will go through the major things the code can and does. As mentioned, the code needs the three-dimensional data. This data is loaded from the last simulation made, but can be changed by using the *loc*-variable to adjust the target directory.

Requirements

A few settings must be made for DENSITY_PROFILE_V4 to work. As mentioned above, the *save_rho*-variable has to be set to *True*. Also, the density data must be saved in the *.npy*-Format. Additionally, it is necessary to use dimensionless units, i.e., code units, for all quantities, since DENSITY_PROFILE_V4 in its current form can only convert code units into the units described below. However, it is highly recommended to always use code units, since these are smaller and therefore handier. Furthermore, as a nice side effect, the main code does not have to convert the inputs to code units, which saves some time (to be fair, not relevant enough, such that a human could measure it).

Stage 1: Fitpreperation

If the target directory is not changed, then the data from the last run will be loaded. From this data, the center is found by searching for the maximal density. This is based on the assumption that the highest density is in the center, even with a core profile. Next, the distance to the center is determined. This is necessary because the center of a merger/soliton can be at any grid point in the box. This will cause problems for the averaging of the density data, since there will be more data available in one direction if the center is displaced from the center of the box. By truncating the density data to the maximum that is possible in all directions, we can assure, that we have the same amount of data in all directions. To get the radial distance, we set up vectors from the center, which then will be combined in a radial vector.

To average the density data, it is suitable to use spherical averaging. This means that we average all density data within a shell of a sphere. This is done by bining the density data into bins, which depend on the spatial resolution of the simulation. The higher the spatial resolution, the smaller the bins, thus making the resulting profile smoother. If a grid point (= data point of a specific density) inside a bin (= radial distance), it will be averaged with all the other points that meet the same criterion. By doing this, we get an average density for each bin. Since the bins relate to radial distance, we get, in the end, an average density for each radial position.

It is possible to make the bins smaller, by decreasing the bin size. However, this can lead to a *nan* when the bin size is too small. It is not recommended to decrease the bin size from the standard, but sometimes it can be handy, especially when an accurate mass is needed. For this purpose, a little section is also implemented that cancels all occurring nans out of the final list. Again, this is not recommended because, by using this, data (= information) will get lost. Therefore, this section is deactivated by default, but can be activated by deleting the # and saving the file afterward.

Last but not least, the resulting radial distances in the form of bins and the corresponding averaged density data will be converted into their specific units. This is kpc for the radial density and $M_\odot/\mathrm{kpc}^3$ for the density. These

units were chosen since they enable a good comparison with results from the literature and show the dimension of the mergers/solitons best when code units are used.

Stage 2: Calculate the total mass and R_p

In the next part the total mass and radius, which contain a user-specific percentage of mass, are calculated. Due to the fact that the density profiles go to infinity, the mass can also grow bigger than it should. It will not go to infinity since the density profile converges, but there can be a huge gap between the real total mass and the calculated one. Thus, the calculated total mass is a reference for how accurate the simulation is. Most of the time, it is hard to modify the parameters to do so. But especially for the box size, it is THE key parameter to getting a good total mass.

Of course, the total mass of the simulation is given by the initial conditions in the notebook. This value will be the correct total mass, which will be used as a reference. To get to the total mass as well as R_p the density profile[4] is integrated. DENSITY_PROFILE_V4 offers the choice between two methods. One is the integration with spherical shells, where the radius bins from the preparation, which form spherical shells in 3D, are used. The other method is the trapezoidal rule. It could be shown that, in most cases, the trapezoidal rule delivers better results. The reason for this is that mostly the total mass is overestimated by the calculation, whereas the trapezoidal rule does this less than the integration with spherical shells. In the case where the total mass is underestimated, the spherical shell-method is a little better. But since this is not as much of a case, the trapezoidal rule is better most of the time.

Both methods integrate the existing density profile to estimate the mass. The user can specify in the notebook which percentage of mass he wants the radius to include. The code then integrates the density profile up to this percentage of mass and saves the current radius bin as R_p. After that, it finished the integration up to the end of the profile to estimate the total mass.

Stage 3: Fitting

The last thing the code does is to fit the resulting density profile. For all fitting procedures, the tool *cure_fit* from *scipy.optimize* is used. First, there are two possible models for the core of the soliton/merger: A Gaussian or a Schive-profile. The Gaussian model is motivated as a solution of the Schrödinger-Poisson/Gross-Pitaevskii-Poisson-system, while the Schive-profile comes from numerical simulations of soliton-mergers. To cover both models, both were used to fit the data. In this case, the whole data points are fitted with the Gauß- and the Schive-profile. The functions for both models are written down in equations 8 and 9.

$$\text{Gauß}(r) = a e^{\frac{-(r^2)}{2\sigma^2}} \tag{8}$$

$$\text{Schive}(r) = \frac{\delta_s}{\left(1 + \left(\frac{r}{r_{sol}}\right)^2\right)^8} \tag{9}$$

Here a for the Gauß and δ_s for Schive stand for the central density of the profile, while σ for Gauß and r_{sol} for Schive are a measure of the size of the core. For the Gauß the typical r_0 has been set 0, because the center of the Gauß should be at a radius of 0. As initial conditions for the unknown parameters, the maximum density, i.e., the density at the first radius bin and the first radius-point (= first radius bin), were given. A point system is then used to decide which model fits the core better. This point system rates according to the following three criteria:

- smallest residual
- standard deviation of the parameter for central density
- standard deviation of the parameter, which defines the width of the model (e.g., σ for Gauß/r_{sol} for Schive).

The fits are compared based on these criteria. If one of the fits is better than the other in one of these criteria, it gets a point. The fit that has more points at the end is selected as the core fit for those data. Due to construction, the point system will always lead to a "winner", so the code will always choose a model.

[4]For further documentation, we will call this density profile, even though it is not a profile, but rather many data points. But to differentiate from the initial loaded in density data, we call the state, where we averaged the data and gave each density data point its radial location "density profile".

Next up, the code finds the connecting-point between the core and envelope. For that, a very simple method is used. It could be shown in various papers, that the size of a soliton core, which has a Gaussian density profile, is well described by the standard deviation of the Gaussian model times a factor:

$$\text{size of soliton core} \approx 2.575829\sigma. \tag{10}$$

This approach is used no matter which core-profile is chosen, since it as it has worked the most consistently and given meaningful results. However, if we only have one soliton, we do not need the connection point, because one soliton has no envelope (at least we do not expect one soliton to have one). Thus, the further connection point determination is skipped, if only one soliton is simulated. For two or more solitons, the radius bin, which is the closest to equation 10, will then be selected as the connection point.

This connection point is then used to split the radius at the connection point, so that the models for the core and envelope can be plotted. In the end, an NWF-Fit is applied to the envelope of the halo. The function for the NFW-model can be found in equation 11.

$$\text{NFW}(r) = \frac{\delta_{NFW}}{\left(\frac{r}{r_s}\right)\left(1+\left(\frac{r}{r_s}\right)\right)^2} \tag{11}$$

Here δ_{NFW} stands for the density at the connection point and r_s is the radius-parameter for the NFW-fit. As initial conditions for the unknown parameters, the values from the connection point were given. To limit the possibilities for the fitting-tool, some parameter-bounds in terms of the connection-point-values were also applied.

Notebook

Small explanation

The notebook is the main user interface for controlling the package. Here, the simulation setting and initial conditions can be chosen and the results are displayed. Even though most of this can be done here, as mentioned above, some parameters have to be altered in the respective files. Also, the notebook is not completely free of code, since the plotting section is coded in the notebook to ensure that one can always adjust the results to the desired look. For further clarifications, it is also useful to look at the comments in the notebook.

The first section is only as a visual to show the user which axion mass is/was used. Other than that, it has no purpose.

The second section controls the simulation parameters. Here the input units for the initial conditions and duration can be set, as well as the box size and resolution, which are the most important simulation parameters, since these influence the spatial resolution. Also, the *save_rho*-variable and the option to save the data in the *.npy*-format can be found here. For the last parameters that can be set here, only the *save_number* is relevant, since this parameter tells one the number of frames, that are saved/displayed.

In the third section the initial conditions are set. This has to be done in the following way for each soliton that one wishes to simulate:

$$soliton = [mass, [x, y, z], [v_x, v_y, v_z], Phase]. \tag{12}$$

At the end, all solitons have to be combined in the *solitons*-list. An example should be available in the notebook.

In the fourth section, only the main algorithm function is used. Here, the current status of the simulation is displayed.

In the fifth section, the visualizations can be customized. This is done by choosing the animation type and the saving type, as well as the various font sizes and figure sizes.

Changes

Besides the bug fixing and the customization of the visualizations, there is not much difference in the first five sections (Set Axion Mass, Set Simulation Parameters, Set Initial Conditions, Run, and Visualisations). In fact, excluding the change of name of the package due to the inclusion of the self-interaction, nothing is different in the first four sections. In the "Visualisations"-section, only three things were altered:

1. In the section where the plotting for animate-options 1 and 3 is handled, there is an if-statement where the figure sizes can be adjusted to the personal liking.

2. A symmetrical axis was included such that 0 is at the center of the box, or rather at the center of the plane (2D)/axis (1D). This axis is in code units.

3. In the animate-function, the labels for title and axes were included. If one wishes to change these or the font sizes, this can be done here.

The biggest change, of course, is the section "Density profile", since it is completely new. There are two subsections in this section: One for the user input and the plotting section. In the first one, the user can choose the state (= snapshot) of the simulation, from which the density profile will be created. As already mentioned, it is not meaningful to use a state, where the solitons are too far apart. Also, the percentage of the mass, that R_p includes can be set here. Keep in mind that p gets multiplied by 100 in the code. Thus, a number between 0 and 1 has to be given. Last but not least, the method for the mass and R_p-determination can be set. Additionally, the user can decide to save the upcoming density profile by setting the *save_figure*-variable to *'y'* (yes).

After that, only the call of the relevant function is done. Under the first section, the code will print four points:

- The method used for calculating the total mass and R_p.
- The total mass, which was determined by the initial conditions.
- The calculated total mass, which determined by the given method.
- The numerical value for R_p and the percentage of mass it includes.

In the last section of the notebook, the plotting section for the density profile, only the plot itself can be altered. This includes, the names of the axes, the figure itself, font size and other things. The only thing to note is, that for single-soliton simulation the limitation that is done at the end is necessary, since the core fit would otherwise dominate the whole plot and the density profile will be only a flat line of points. If the user wishes, the saving directory and name of the saved plot can be altered here. By default, this is set to the directory where the density data is kept.

How to use the package

This section should give instructions on how to use the package. Keep in mind that some things will be repeated from above.

To start, one must first create the initial soliton profile with *soliton_solution* for the desired *Lambda_hat*, which has to be set in the code. Since that takes about 2 minutes anyway, one can change the axion mass in the main code PyUltraLight_SI, and the axion mass in density_profile_V4. If the calculation of the initial soliton profile is finished, the initial profile file *initial_fSI.npy* and the text file *Info_soliton_solution.txt* are created/overwritten (in the case they already exist). Do not forget to save PyUltraLight_SI and density_profile_V4 after the changes.

Then one can turn to the Jupyter notebook, which is responsible for the rest of the simulation configurations. There, the axion mass should be entered again (only for visualization), as well as the units in which the code should output the results. If one wants to use density_profile_V4 then the dimensionless units have to be used, i.e., ' ' for all quantity unit variables. In addition, one can specify the duration of the simulation plus time unit, the size of the box and the resolution. Here, the saving options for the data are also chosen. For density_profile_V4 it is necessary to save the 3D-data with the variable *save_rho* in the *.npy*-format. This can be done by setting the according variable to *True* and the others to *False*. If one wishes to see the 2D/1D-animation, the variables for *save_plane* and *save_line* have to be set to *True*. The same goes for the energies. It is possible to manipulate the step size, but this is not necessary because the code decides the "ideal" time step size itself. Also, the save number, which is nothing but how many snapshots you want to have, is specified here. The more that is chosen, the better the temporal resolution. But keep in mind, that many snapshots will take up a lot of saving space. One also has the option to include a central potential in the simulation by specifying a central mass.

Then one comes to the initial conditions for the simulation. Here the number of solitons, the soliton's mass, position in the box, velocity, and phase are chosen. Keep in mind, that this has to be done for each soliton, and each soliton should be in its own variable. Afterward, the *solitons*-list needs to be updated by including all solitons one wishes to simulate.

The output can be animated (in the case of 2D and 1D) and this animation can be saved. Also, energy plots can be created and saved. This works quite simply by using the given number system. For example, in the code, "0" means no animations, while 1 means the 2D-animation. If one wants to customize the output, this can be done here.

Last but not least, for the density profile, one must specify the state of the simulation/the snapshot of the simulation, from which the radial density profile will be created. For this, also the percentage of mass, which the R_p should include and the integration method have to be set. Additionally, the *save_figure*-variable has to be changed to *'y'* if the user wants to save the density profile-plot.

The customization of the radial density profile plot can be done at the very bottom. Here one can also deactivate certain features of the default settings, e.g., the black line for the R_p and change the saving directory and name of the saved density profile plot.

Ideas for further upgrades

Of course ,this package has a lot of room for improvement. Some ideas are collected here that were not implemented.

Quality evaluation of the fits

One of the biggest problem of the variability of the code is, that the results can also vary as much. This hits the automated and generalized fitting-routine hard, resulting in low-quality fits some time. To avoid deactivating the fits in the plotting section, an automated code section could be implemented, that ignores the fitting results, if they are not good enough. One feasible choice could be the use of the determination coefficient in combination with a limit. For example the fits could be discarded, if the determination coefficient is below a certain threshold.

The implementation of this would be suitable in the fitting-routine-function of DENSITY_PROFILE_V4. Of course, some lines in the later section would also must need to be altered.

One benefit of the determination coefficient could be to replace the existing point system. Therefore, the determination coefficient could give the code information about, which core fit is used.

Boundary conditions

The currently used periodic boundary conditions make the use of the package sometimes difficult. Extreme care has to be taken, when solitons or parts of solitons come close to the edge of the box, since they could exit and reenter from the other side, falsifying any results.

One way to solve this issue is to install an absorbing sponge, that swallows any matter that may exit the simulation box. Many papers have already installed such a feature in their code. We decided not to install one since, for our purposes, it was not needed. However, for others, the sponge could be a welcome help.

Change of axion mass

Currently, the axion mass is the only parameter, that has to be changed in all the relevant files (except for *soliton_solution.py*).

A neat upgrade would be to adjust the axion mass in the Jupyter notebook and this change would be communicated to the respective files. With this, the package would be even more user-friendly and could represent a simplification for future research.

Turn off/on of density profiles

If the whole notebook is executed, the section for density profiles is executed as well. However, for some simulations density profiles are not wanted or would not make sense. Thus, the user has to run each section separately up to the point of the visualization/animation of the simulation.

However, for simulations, where this feature is not needed, it would be good to deactivate it. For this reason, a switch in the form of a variable could be implemented, that turns the section on or off. Probably a *number system*, *True/False* or *yes or no* would be the easiest way and fit in with the rest of the notebook.

Turn off/on of fits

Like for the density profiles, it could also be useful if a switch for the fits in general is implemented. If it is known, that for a specific simulation the fits will not be useful or would turn out bad, the fits are not meaningful. Thus, an on-off-switch could save a lot of time and therefore increase the comfort of the user.

Animation of the density profiles

The current package is only able to show one state of the simulation and the relevant section in the notebook has to be executed multiple times to see the density profiles of all states. Also, the state-variable has to be changed each time.

Especially if the temporal evolution of the radial density profile is required, an animation like in the Visualisation-section would be a plus. In this case, the ability to choose the time frame one wants to see would also be a good thing.

For the implementation, the basic set-up from the Visualisation-section could be used.

ETC.

As said, there is still a lot of room for improvement, and many things, that could improve the package are not listed here, since we only included the ones that we had during the upgrade of the package. It is up to the reader to come up with further ideas that will further increase the usability of the package.