Modelling solar evolution with MESA and EVtwin: Appendices

Alexander Olza Rodriguez

June 19, 2020

Contents

A Reproducibility

All files used in this work are listed below. They must be in the same directory, and with their original names.

- `struct_GS98.dat`
- `struct_AGS.dat`
- `GS98.csv`
- `AGS.csv`
- `amuse_tutorial.py`
- `tutorial_plots.py`
- `functions.py`
- `run_mesa.py`
- `run_evtwin.py`
- `run_alternative.py`
- `make_graphs_both.py`
- `make_graphs_alternative.py`

The first two files contain the benchmarks by Vinyoles et al., and they are publicly available at this site[1]. The third and fourth contain the chemical composition, available here[2]. The rest constitute the source code of this work, distributed as complementary material.

To reproduce the results, follow the instructions below:

1. Install AMUSE (version 12)[3] with MESA

[1] https://www.ice.csic.es/personal/aldos/Solar_Data.html

[2] http://stev.oapd.inaf.it/ webmaster/aesopus_1.0/HELP/aesopus_help/node1.html

[3] If you prefer to use the latest AMUSE, note that the most up-to-date versions do not offer support for Python 2.x. Thus, you will need to translate the source code to Python 3. A tool such as the translator `2to3` may be handy.

2. Run `amuse_tutorial.py`. As output, you will get figures 2 and 3.

3. Run `run_evtwin.py`. The program will ask you to choose a composition. Type 'AGS' or 'GS98'. After around an hour the calculations will become too slow. Then the program will give you the option to terminate the execution: do so. Then run the script again with the other composition. As output, you will get two new directories (`evtwin_AGS_results` and `evtwin_GS98_results`) full of csv files. They contain all of the saved stellar models.

4. Repeat the process above with `run_mesa.py`.

5. Run `make_graphs_both.py`. When prompted, choose the composition GS98. As output, you will get a new directory named `figures` with figures 5, 6 and 7. Run it again, now choosing AGS. This will generate figures 8, 9 and 10.

6. Run `run_alternative.py`. As output you will get a directory `mesa_alternative_results` containing the set of stellar models as csv files.

7. Run `make_graphs_alternative.py`. This will generate figures 11, 12 and 13.

B Notes on the source code

This appendix contains the documentation of the software listed in Appendix A. We will not describe the graphing programs because they only contain Matplotlib instructions.

B.1 AMUSE tutorial

The two scripts `amuse_tutorial.py` and `tutorial_plots.py` constitute a tutorial to get started with AMUSE. `amuse_tutorial.py` calculates the orbits of the planets in the Solar System using the NBody integrator `ph4`. It is carefully explained in Section 2. `tutorial_plots.py` contains common instructions to plot data with the Python library Matplotlib. These programs generate figures 2 and 3.

B.2 Stellar evolution simulations: Auxiliary functions.

Now we will describe all of the auxiliary functions used in the simulations. They are grouped all together in the file `functions.py`.

B.2.1 The function `uniform_star`

- **Input:** `code,code_name,mass,Z,H,composition_file`
  - `code`: A code instance. For example, `code=MESA()`. This object is responsible for the communication between the user script and the independent process running the code MESA in the background.
  - `code_name`: A string. It should be "mesa" or "evtwin".
  - `mass`: A quantity. In our case, `mass= 1|units.MSun`.
  - `Z` and `H`: Two floats. They represent the metallicity and the Hydrogen mass fraction. They are different for each composition.
  - `composition_file`: A string. The name of the composition file we will read.
- **Output:** A particle that can be evolved by the `code` given as input. It represents a star with the desired uniform composition.

Description: First, we read the desired composition file. In this work we have used [GS98] and [AGS07] compositions, extracted from ÆSOPUS[4]. The first 3 lines of the composition file are a header, so we skip them. We have only read the elements with a mass fractions greater than 10^{-4}.

Now we create a star with a default structure to copy the default r, P, L, T profiles into our custom-made star. This process is different with MESA and EVtwin because of the way they are implemented into AMUSE.

For EVtwin, the stellar structure is contained in an AMUSE-specific data structure, called a Grid, of dimension 199. We query it with the AMUSE built-in function `get_internal_structure()` that applies to particles added to an EVtwin instance. We declare a new Grid and we initialize it with the default structural profiles and with the custom mass fractions.

For the case of MESA, the structure is contained in a collection of arrays with units. Each profile can be obtained separately, with methods such as:

Listing 1: Obtaining the cumulative mass profile of a MESA star.

```
1
2 default_star=code.particles.add_particle(Particle(
     mass=1|units.MSun))
3 zones = default_star.get_number_of_zones()
4 mass=default_star.get_cumulative_mass_profile(
5         number_of_zones=zones) * default_star.mass)
```

B.2.2 The function `core_radius`

- **Input:** `sun, code`
  - `sun`: A particle that must be attached to either MESA or EVtwin. Thus, it must have the corresponding attributes of a star.
  - `code`: A string. The name of the code.
- **Output:** A quantity with units of length. The radius of the first sphere that contains $0.1 M_{\odot}$.

[4]http://stev.oapd.inaf.it/ webmaster/aesopus_1.0/HELP/aesopus_help/node1.html

B.2.3 The function `save_results`

- **Input:** `sun,filename,composition,code_name`
  - `sun`: A particle that must be attached to either MESA or EVtwin. Thus, it must have the corresponding attributes of a star.
  - `filename`: A string. The name of the file in which we want to store the results.
  - `composition`: A string, 'GS98' or 'AGS'. It will be mentioned in the file header.
  - `code_name`: A string, 'mesa' or 'evtwin'. Depending on its value, `save_results` will acess the stellar structure via `structure_from_star (sun)` for 'mesa' (described below) or `sun.get_internal_structure ()` for 'evtwin'. The name of the code will also appear in the file header.
- **Output:** None.

Description: The function creates a csv file that contains the current stellar model.

This function uses the AMUSE built-in method `.value_in(units)`. The resulting file contains values in a specific set of units, described in the header.

B.2.4 The function `structure_from_star`

- **Input:** `star`. A particle attached to a stellar evolution code (MESA or EVtwin).
- **Output:** A Python dictionary. It contains the structural and chemical profiles. The keys are `radius, density, mass, luminosity, temperature, pressure, composition, species_names`. Each entry is an array of quantities, with dimension equal to the number of zones in the star, except `species_names`, which is an array of strings.

B.2.5 The function `cycle`

This function is responsible for stellar evolution. It consists in a `while` loop that iterates until the input star changes its stellar type.

- **Input:** `i,n,t,sun,path,composition,code,code_name`
  - `i`: An integer. We will increment it in each iteration, to generate numbered output files with the name `path_i.csv`.
  - `n`: An integer. It is related to the timestep for saving the results (not the internal timestep of the stellar evolution code): $\Delta t = 4.5395 Gyr/n$.
  - `sun`: A particle attached to `code`.
  - `path`: A string. The directory in which we will store the results.
  - `composition`: A string, 'GS98' or 'AGS'.
  - `code`: A code instance.
  - `code_name`: A string.
- **Output:** `i,t` described above.

Description: While the stellar type of `sun` remains unchanged, the star evolves in intervals of dt using the AMUSE built-in function `code.evolve_model(t)`. This function connects with the numerical solver running in the background (represented by the instance `code`). Then the background code solves the stellar structure and evolution equations, generating a series of stellar models (that will not be saved) until the indicated time `t` arrives. After that we print the current age of `sun` and we call our function `save_results` with a unique filename.

`cycle` also implements a series of stopping conditions. If the age of the star (obtained with AMUSE, `sun.age`) exceeds a certain value, we print a message warning the user that the calculations have become extremely slow, and we give them the option to terminate or continue with the evolution. This critical age is different for each code and composition, and it has been determined empirically.

B.3 Simulations with MESA: `run_mesa.py`

At the beginning of the execution, we ask the user to type the name of the desired composition. When we obtain a valid answer, we assign the corresponding values to Z and H and run the simulation.

First we create a directory `mesa_{0}_results` where `{0}` is the chosen composition.

Then we declare an instance `mesa=MESA()` and we switch off the stellar wind. We read the corresponding composition file (which must be in the same directory as `run_mesa.py`, under the name 'AGS.csv' or 'GS98.csv'), and we create our custom star by calling `uniform_star`.

After that, we save the initial results and we start the evolution. To do so we call `cycle` twice: Once for the Main Sequence (with `n=100`) and once for the First Giant Branch (with `n=500`). Finally, we terminate the MESA background process (`mesa.stop()`).

Note that, in the simulations performed in this work, the second call to `cycle` was interrupted when the calculations became extremely slow, without finishing the First Giant Branch (except for the alternative composition).

B.4 Simulations with EVtwin: `run_evtwin.py`

This script is exactly analogous to `run_mesa.py`. The only difference is that the stellar wind could not be completely switched off. Although it is only explicit as a comment in the code, the AGB wind used in our simulations is set to its default value, corresponding with Watcher's fitting formula.

B.5 Alternative simulation: `run_alternative.py`

The alternative simulation is performed with MESA. The script is analogous to the one described in section B.3, except that we do not have to choose a composition and we do not call the function `uniform_star`. Instead, we create the uniform star directly in the script with our custom composition. After that we call `cycle` twice, with `n=100` for the Main Sequence and `n=1000` for the First Giant Branch. In this case, the two cycles come to an end without manual interruptions.