User guide to the Multiple Stellar Evolution (MSE) code

Adrian Hamers (hamers@mpa-garching.mpg.de)
https://mse-group.net

https://github.com/hamers/mse
October 2, 2020

Contents

1 Introduction

This is the user guide to the Multiple Stellar Evolution (MSE) code. MSE models the long-term evolution of hierarchical multiple-star systems (binaries, triples, quadruples, and higher-order systems) from the main sequence until the remnant stages. It takes into account gravitational dynamical evolution, stellar evolution (using the SSE tracks), and binary interactions (such as mass transfer and common-envelope evolution). It includes routines for external perturbations from flybys in the field, or (to limited extent) encounters in dense stellar systems such as galactic nuclei.

2 Installation

MSE is written primarily in C++ but also links to some Fortran routines (from SSE/BSE). A Python interface is included for convenience. The source code is available online[1] at https://github.com/hamers/mse. To download it, you can use the following command (it is assumed that you have `git` installed; see, e.g., this url for general installation instructions for `git`):

```
git clone https://github.com/hamers/mse.git
```

This will create a new directory `mse` within your current directory.

To install (i.e., compile) MSE, C++ and Fortran compilers are required, as well as[2] Python 3 in order to use the Python interface. A `Makefile` is included which should be used for compilation. By default, it assumes that `g++` and `gfortran` are available on the system. To compile, simply type

```
make
```

If `g++` and/or `gfortran` are not available, you will need to modify the lines

```
CXX = g++
FC = gfortran
```

in the `Makefile` accordingly to correspond to the C++ and Fortran compilers that you would like to use.

The compilation will typically take a few seconds. If compiled successfully, you should see the message

```
The shared library mse has been created successfully.
```

In order to use the Python interface, the `numpy` package is required. Also, `Matplotlib` is needed to produce plots in some of the scripts provided with the code.

> To remove previously compiled binary files, type
>
> ```
> make clean
> ```
>
> This is usually necessary when copying a compiled version of the code to another system. Also, it can be useful when mixed compilers were accidentally used, and errors occur.

[1] At the moment of writing, the `GitHub` repository is private. Access will be given on a per-request basis (please contact the author). In the future, the repository will be made public.

[2] The code is tested with Python 3; there is no guaranteed compatibility with older versions of Python.

3 Testing the installation

To make sure that the code has been properly installed, you can use a PYTHON test script which is included. In the main directory of MSE, simply type

```
python3 test_mse.py
```

This will run a number of tests, producing some output. If all tests ran successfully, you should see the message

```
All tests passed!
```

> Specific tests (see `test_mse.py` for details) can be initiated by passing the `--t X` command line argument to `test_mse.py`, where `X` is the number of the test to run. In addition, some tests have a 'detailed' mode which is more extensive, but takes significantly longer to run. The detailed mode can be activated by adding the command line argument `--m 1`.

4 Code usage within PYTHON

MSE can be used with the PYTHON interface (recommended for most users), but also directly within C++ (recommended for advanced users only)[3]. In this section, we describe usage within PYTHON.

4.1 Basic

4.1.1 From the command line: `run_system.py`

The easiest way to run a system in MSE is by using the included `run_system.py` script. This script has the convenience of being fully accessible within the command line. It produces basic plots with the time evolution of some quantities, mobile diagrams, and positions of the stars/bodies. It currently supports two types of hierarchical configurations: 'fully nested' systems, in which the number of levels in the system is maximised, and '2+2 quadruple' systems (see Fig. 1).

For example, a hierarchical triple can be simply run with the following command shown in Code snippet 1.

```
python3 run_system.py --configuration "fully_nested"
--masses 40 10 2 --metallicities 0.02 0.02 0.02
--smas 15 120 --es 0.1 0.2 --is 0.001 1.4
--LANs 0.01 0.01 --APs 0.01 0.01
--tend 2e7 --Nsteps 2000 --plot_filename "figs/mytriple"
```

Code snippet 1: Example usage of `run_system.py` for a triple.

Note that multivalued entries (such as `--masses`) are to be input using spaces to separate values. All masses should be given in $M_\odot$, all semimajor axes in AU, times should be given in yr, and all orbital angles in rad. When `--configuration` is given as `"fully_nested"`, the number of bodies in the system is inferred from the number of values given to the "`--masses`" argument. If, instead, "`--configuration`" is given as `"2+2_quadruple"`, a 2+2 quadruple is assumed (so four stars with two binaries orbiting each other's center of mass).

[3] At the moment of writing, MSE has been most tested with the PYTHON interface. Calling the code directly within C++ is less tested.

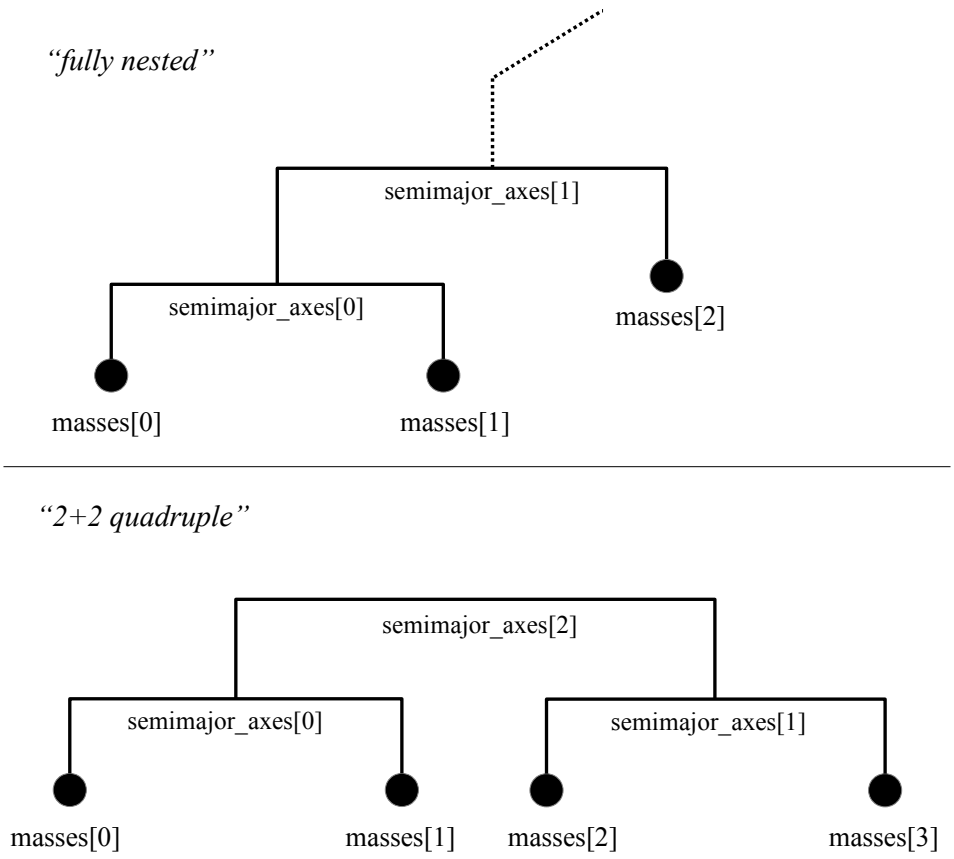

Figure 1: Illustration of the currently supported configurations within the `run_system.py` script and the `evolve_system()` wrapper function. Also shown are designations within the argument lists (displayed are `masses` and `semimajor_axes`, but the same orders apply to other properties of bodies/orbits). Top: a 'fully nested' configuration (explicitly showing the simplest case of a triple). Bottom: a 2+2 quadruple system.

In the above example Code snippet 1, three masses are given, so the system is assumed to be a hierarchical triple. For multivalued entries, the order of the values should be consistent with the order as indicated in Fig. 1. That is, for our example, the initial masses of the two stars in the inner binary are 40 and 10 $M_\odot$, respectively, and the tertiary mass is 2 $M_\odot$. The metallicities (`--metallicities` or `--zs`) are 0.02 for all stars. The inner and outer orbital semimajor axes (`--smas` or `--semimajor_axes`) are 15 and 120 AU, respectively, and the eccentricities (`--es` or `--eccentricities`) are 0.1 and 0.2 for the inner and outer orbits, respectively. The inclinations (`--is` or `--inclinations`) of the inner and outer orbits are 0.001 and 1.4 rad, respectively. The longitudes of the ascending node (`--LANs` or `--longitudes_of_ascending_node`) and the arguments of periapsis (`--APs` or `--arguments_of_pericentre`) are all 0.01 for the inner and outer orbits, respectively.

In `run_system.py`, orbital angles are used to describe the initial orbital orientations (internally, the code uses orbital vectors). See, e.g., https://mitpress.ublish.com/ereader/9943/?preview#page/27 for a brief introduction to orbital angles (if the link in this document does not work, please try pasting it directly into your browser). In particular, note that the *relative* or *mutual* inclination between two orbits, $i_{\rm rel}$, depends on both the individual inclinations and longitudes of the ascending node of those two orbits. Specifically, using i_k and Ω_k to denote the inclination and longitude of the ascending node of an orbit k, the mutual inclination between two orbits labeled '1' and '2' is given by

$$\cos(i_{\rm rel}) = \cos(i_1)\cos(i_2) + \sin(i_1)\sin(i_2)\cos(\Omega_1 - \Omega_2). \quad (1)$$

The main use case of MSE is to model initially hierarchical and dynamically stable systems. However, it is possible to enter initially unstable systems. In the latter case, MSE will immediately switch to direct N-body integration.

The duration of the integration is specified with the `--tend` argument. The number of output steps (this is not the same as the number of steps taken internally within MSE) is given by `--Nsteps`. After completing the simulation, three plots are saved using filenames based on the `plot_filename` argument (note that the directory `figs` should exist for the example in Code snippet 1). The three plots are:

- `[plot_filename].pdf`: time evolution of the masses (top panel), orbital separations and radii (middle panel), and stellar types (bottom panel).
- `[plot_filename]_mobile.pdf`: mobile diagrams of the system during important stages of the evolution.
- `[plot_filename]_pos.pdf`: positions of all bodies in the (X, Y)-plane.

Choosing a very large `--Nsteps` (say, $\gg 10^4$) will produce very smooth-looking plots, but will tend to slow down the integration because communication between PYTHON and C++ is involved for each output step. In practice, it is best to find a balance between quality of presentation, and speed.

4.1.2 The `evolve_system()` wrapper function

The code can also be used in a similar fashion as described above within a user-created PYTHON script (i.e., not using the command line directly with `run_system.py`). In fact, as can be readily inferred from inspection of `run_system.py`, the latter script simply parses the command line arguments and passes them to the `evolve_system()` function which is part of the `Tools` class (provided with MSE). Usage of `evolve_system()` is, therefore, very similar to the above. For example, to run the same example system, the PYTHON code as shown in Code snippet 2 could be used. Note that the order of the arguments given to `evolve_system()` is important.

```
Tools.evolve_system("fully_nested"
    ,3,[40,10,2],[0.02,0.02,0.02],\
     [15,120],[0.1,0.2],[0.001,1.4],[0.01,0.01], [0.01,0.01],\
     2.0e7,2000,plot_filename="figs/mytriple")
```

Code snippet 2: Example usage within Python of `evolve_system.py` for a triple.

4.2 More advanced: custom usage

Although the basic usage options (Section 4.1) can be convenient, they do not give much control over the simulation. If more customisation is needed, it is possible to call MSE more directly. The usage of the code in this manner is inspired by the syntax used in the AMUSE framework[4].

```
1  from mse import MSE,Tools,Particle
2
3  N_bodies=3
4  masses = [40,10,2]
5  metallicities = [0.02,0.02,0.02]
6  semimajor_axes = [15 ,120]
7  eccentricities = [0.1,0.2]
8  inclinations = [0.001,1.4]
9  arguments_of_pericentre = [0.01,0.01]
10 longitudes_of_ascending_node = [0.01,0.01]
11
12 particles = Tools.create_nested_multiple(N_bodies,masses,\
13     semimajor_axes,eccentricities,inclinations,\
14     arguments_of_pericentre,longitudes_of_ascending_node,\
15     metallicities=metallicities)
16
17 code = MSE()
18 code.add_particles(particles)
19
20 Nsteps = 2000
21 tend = 2.0e7
22 t = 0.0
23 dt = tend/float(Nsteps)
24
25 while t<tend:
26     t+=dt
27     code.evolve_model(t)
28
29     particles = code.particles
30     orbits = [x for x in particles if x.is_binary==True]
31     bodies = [x for x in particles if x.is_binary==False]
32
33     print( 't/Myr',t*1e-6,'es',[o.e for o in orbits],'smas',\
34          [o.a for o in orbits])
```

[4] amusecode.org

```
35
36 print("log",code.log)
37 code.reset()
```

Code snippet 3: Custom usage example within Python for a triple.

We give a simple example in Code snippet 3, initialising and running the same system that was given in Section 4.1. The function `create_nested_multiple()` from the `Tools` class is used (lines 12-15) to generate `particles`. Here, `particles` is a Python `list` containing `Particle` objects; the `particles` represent the entire system (see Section 6 for more information). An instance of the code is created (line 17), and the `particles` are added to the code (line 18).

Lines 25-34 constitute a time loop in which the code is called repeatedly with `evolve_model()` until reaching the desired end time. In line 29, the `particles` set local to the user is updated from the `particles` set in MSE. The subsequent lines, 30-31, separate the particles out into `orbits`, i.e., particles representing orbits, and `bodies`, i.e., particles representing bodies/stars. In lines 33-34, the eccentricities and semimajor axes of all orbits are printed. Any other time-dependent data from the `particles` can also be accessed at this point.

> Note that MSE determines its own internal time-steps dynamically at runtime. In principle, changing `Nsteps` should not affect the result. However, because systems are often chaotic and because of unavoidable numerical errors, the outcome of a particular system can actually often depend on `Nsteps`.

In line 36, the `code.log` is printed. This contains detailed information on 'interesting' events during the evolution. Its purpose is to be able to track important events, without the risk of missing them when running the high-level time loop (lines 25-34 in Code snippet 3), which is particularly useful for population synthesis studies. The contents of `code.log` are described in detail in Section 7.

> The `particles` set is dynamic and can change as the system is evolving. For example, particles can be removed if stars merge into one. If the structure of the system changed when calling `evolve_model()`, this is indicated with the parameter
>
> ```
> code.structure_change
> ```
>
> which will be `True` (default value `False`). When the structure changes, one should bear in mind for analysis purposes that the `particles` will be different (different number of bodies, different hierarchy, etc.).

Lastly, when finished with the code, care should be taken to call `code.reset()`. Failing to do so can result in unexpected behaviour when using the code repeatedly.

5 Code usage within C++

Using the code within C++ gives most customisability and minimises any potential overhead losses caused by the Python interface. However, we only recommend it for more advanced use cases. We give a simple example for a triple in Code snippet 4.

```
1 ParticlesMap particlesMap;
2 int N_bodies = 3;
3 double masses[3] = {40.0,10.0,2.0};
4 int stellar_types[3] = {1,1,1};
5 double smas[2] = {15.0,120.0};
6 double es[2] = {0.1,0.2};
7 double TAs[2] = {0.01,0.01,0.01};
```

```
8  double INCLs[2] = {0.001,1.4};
9  double APs[2] = {0.01,0.01};
10 double LANs[2] = {0.01,0.01};
11
12 create_nested_system(particlesMap,N_bodies,masses,stellar_types,
       \
13     smas,es,TAs,INCLs,APs,LANs);
14
15 initialize_code(&particlesMap);
16
17 double t=0.0;
18 double tend = 2.0e7;
19 int Nsteps = 2000;
20 double dt = tend/ ((double) Nsteps);
21 double output_time,hamiltonian;
22 int state,CVODE_flag,CVODE_error_code;
23
24 while (t<tend)
25 {
26     evolve(&particlesMap,t,t+dt,&output_time, \
27         &hamiltonian,&state,&CVODE_flag,&CVODE_error_code, \
28         &integration_flag);
29     print_system(&particlesMap,integration_flag);
30     t += dt;
31 }
32
33 clear_particles(&particlesMap);
```

Code snippet 4: Example usage within C++ for a triple.

6 Particle sets

In MSE, the structure of the system at any time is specified by a set of particles. A particle can represent a body (a star, compact object, or planet), or a binary/orbit. Each orbit has two 'children', which themselves can be bodies or orbits. By linking the children of each orbit to other members of the set of particles, any hierarchical structure consisting of nested orbits can be specified. For example, a hierarchical triple-star system is represented by a set of five particles; three of which represent the three stars, and two of which represent the inner and outer orbits.

The code comes with several functions that can be used to automatically produce `particles` in Python. In particular, `create_nested_multiple()` produces the particles representing a fully nested system (see the top panel in Fig. 1). As with `run_system.py` and `evolve_system()`, care should be taken that the parameters are entered in the correct order. Also included is `create_2p2_quadruple_system()`, which generates a 2+2 quadruple system.

However, MSE is not limited to the configurations mentioned above; any hierarchical configuration is allowed, as long as it consists of binary orbits. As an example, we show in Code snippet 5 how in Python a system can be generated that represents a '(2+2)+1' quintuple system. A simple representation of this system in a mobile diagram is given in Fig. 2.

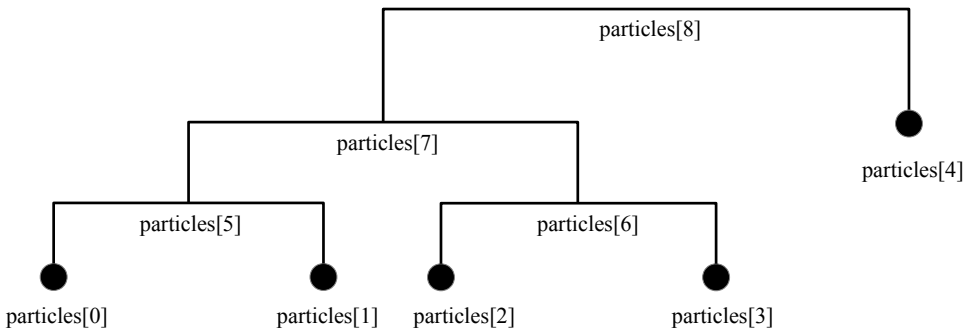

Figure 2: Mobile diagram of a hierarchical quintuple. Indicated are which elements of the particles set the bodies/binaries are assigned to.

```
N_bodies = 5
N_binaries = N_bodies-1

particles = []

### Add the bodies ###
for index in range(N_bodies):
    particle = Particle(is_binary=False,mass=masses[index])
    particles.append(particle)

### Add the binaries ###
for index in range(N_binaries):
    if index==0: ### corresponds to particles[5]
        child1 = particles[0]
        child2 = particles[1]
    elif index==1: ### corresponds to particles[6]
        child1 = particles[2]
        child2 = particles[3]
    elif index==2: ### corresponds to particles[7]
        child1 = particles[5]
        child2 = particles[6]
    elif index==3: ### corresponds to particles[8]
        child1 = particles[7]
        child2 = particles[4]

    particle = Particle(is_binary=True,child1=child1,\
        child2=child2,a=semimajor_axes[index],\
        e=eccentricities[index],INCL=inclinations[index],\
        AP=arguments_of_pericentre[index],\
        LAN=longitudes_of_ascending_node[index])
```

```
31      particles.append(particle)
```

Code snippet 5: Example of how to generate a particle set in Python representing a hierarchical quintuple in the (2+2)+1 configuration. It assumes that the parameters are pre-defined in the variables masses, semimajor_axes, eccentricities, inclinations, arguments_of_pericentre, and longitudes_of_ascending_node. See also Fig. 2.

In lines 7–9 of Code snippet 5, the bodies are added first, and the binaries are added later. Here, we choose the five bodies to be represented by particles[0] through particles[4]. In general, there is no need for any particular ordering of the particles, but it is recommended to follow a set strategy of assigning particles to avoid possible confusion later on.

> A mass should always specified for a body. By default, bodies are assumed to be stars, and the Boolean parameter evolve_as_star is True. Optionally, the metallicity can be specified (particle.metallicity = ...). Initial stellar radii are determined from the SSE tracks. To model a planet, one should set particle.evolve_as_star to be False. The radius should then be set with particle.radius, as well as the tidal evolution parameters (see Table ??).

Lines 12–31 describe how particles representing binaries are created. When creating a Particle with is_binary = True, at least two additional arguments should be given: child1, and child2. The latter should be a Particle, and in the example, they are assigned to previously-created elements of particles in such a manner to produce the structure depicted in Fig. 2. The other arguments given to Particle() are the orbital parameters. Note that, in this example, the order of the elements in semimajor_axes etc. should correspond to the orbits in Fig. 2, i.e., from bottom to top, and from left to right.

7 Logs

While running, MSE generates logs of 'interesting' events such as stellar type changes, SNe, Roche lobe overflow, common-envelope evolution, collisions, and the onset of dynamical instability. By generating logs internally when such events happen, there is no risk of them being missed (the latter is possible when running a high-level time loop such as in lines 25–34 in Code snippet 3 and manually analysing the current particles at specific output times). The logs generated by the code are particularly useful for population synthesis studies.

Here, we describe how the logs are accessible through the Python interface. Once a system has run, the log can be retrieved as

```
log = code.log
```

The log is a list with each element representing an event. The elements are Python dictionaries (dict) which contain the following names:

- time: the time of the event (in yr);
- event_flag: an integer number describing what kind of event occurred. See Table ?? for an overview of the meanings of the various event flags.
- index1: if applicable, the particle index of one of the primary (inner) objects involved in the event (for example, one of the bodies involved in a collision);
- index2: similar to index1, but now pertaining to the secondary object;
- binary_index: if applicable, the particle index of the parent object involved in the event (for example, the orbit in which a collision occurred);
- N_particles: the current number of particles in the system.

Event flag	Description
0	Initial system
1	Stellar type change
2	Beginning of SNe event
3	End of SNe event
4	Start of Roche lobe overflow
5	End of Roche lobe overflow
6	Start of common-envelope evolution
7	End of common-envelope evolution
8	Start of collision event
9	End of collision event
10	Onset of dynamical instability in the system
11	Breakdown of the secular equations of motion (will continue using direct N-body)

Table 1: Overview of the meanings of the various event flags that appear in MSE's log output.

8 Code parameters

MSE has a large number of parameters. They can be separated into two types: global parameters that apply everywhere and at all times, and particle-specific parameters that only apply to specific particles (bodies or binaries).

Global parameters can be set as follows. For Python, assuming that `code` is an instance of MSE,

```
code.parameter = value
```

Here, `parameter` is the name of the parameter, and `value` is its value. In C++, global parameters are global variables and can be set by

```
parameter = value;
```

Particle-specific properties are set in a similar way, except that they should be set as attributes to the corresponding particle. In Python,

```
particle.parameter = value
```

for a `particle`. In C++,

```
particle.parameter = value;
```

for a `particle` of the `Particle` class.

A comprehensive overview of the global and particle-specific parameters is given at the end of this document in Tables 2 and 3, respectively.

9 Citing the code in scientific publications

When using MSE for scientific publications, please cite the following paper:

- –To be included!–

If you would like to be thorough, you could also include the following papers that describe codes on which MSE is heavily based:

- Hurley, J.R., Pols, O.R., Tout, C. *Comprehensive analytic formulae for stellar evolution as a function of mass and metallicity*. MNRAS, Volume 315, Issue 3, pp. 543-569.
- Hurley, J.R., Tout, C., Pols, O.R. *Evolution of binary stars and the effect of tides on binary populations*. MNRAS, Volume 329, Issue 4, pp. 897-928.
- Rantala, A., Pihajoki, P., Mannerkoski, M., Johansson, P.H., Naab, T. *MSTAR - a fast parallelized algorithmically regularized integrator with minimum spanning tree coordinates*. MNRAS, Volume 492, Issue 3, p.4131-4148.
- Hamers, A.S., Portegies Zwart, S.F. *Secular dynamics of hierarchical multiple systems composed of nested binaries, with an arbitrary number of bodies and arbitrary hierarchical structure. First applications to multiplanet and multistar systems.* 2016, MNRAS, Volume 459, Issue 3, p.2827-2874.
- Hamers, A.S. *Secular dynamics of hierarchical multiple systems composed of nested binaries, with an arbitrary number of bodies and arbitrary hierarchical structure. II. External perturbations: flybys and supernovae*. 2018, MNRAS, Volume 476, Issue 3, p. 4139-4161.
- Hamers, A.S.. *Secular dynamics of hierarchical multiple systems composed of nested binaries, with an arbitrary number of bodies and arbitrary hierarchical structure. III. Suborbital effects: hybrid integration techniques and orbit-averaging corrections*. 2020, MNRAS, Volume 494, Issue 4, pp.5492-5506.
- Hamers, A.S., Dosopoulou, F. *An analytic model for mass transfer in binaries with arbitrary eccentricity, with applications to triple-star systems*. 2019, ApJ, Volume 872, Issue 2.

10 Q&A

The code is slow! What can I do?

Be more patient :-). More seriously, MSE can be computationally demanding depending on the system, and the duration of the integration. For example, secular evolution could occur on a very short time-scale in a tight (but dynamically stable) triple, requiring many internal steps to be taken. Or the system could be dynamically unstable for a long time meaning that direct N-body integration is used continuously, and the latter tends to be significantly slower than the secular approach. If a system takes a very long time to run, one could try adjusting the code's tolerances. The `relative_tolerance`, used for secular integration, could be increased up to, say, 10^{-5} (ideally not larger). Alternatively, if the bottleneck lies with direct N-body integration, the N-body tolerance `mstar_gbs_tolerance_default` could be increased to up to, say, 10^{-4}.

It is also possible that the code completely stalls, and this is likely due to a bug. If you suspect that this is the case, please contact the author and describe the problem in detail.

More Q's to be added

Name	Symbol	Description	Default value
Physical constants			
`CONST_G`	$\mathcal{G}$	Gravitational constant	$4\pi^2$
`CONST_C_LIGHT`	c	Speed of light	63239.72638679138
Algorithm/prescription parameters			
`nbody_analysis_fractional_semimajor_axis_change_parameter`	$\alpha_{N,a}$	Largest allowed fractional semimajor axis change for switching from N-body to secular integration.	0.01
`nbody_analysis_fractional_integration_time`	$\alpha_{N,\Delta t,\mathrm{an}}$	N-body analysis timestep multiplication factor.	0.1
`eddington_accretion_factor`	f_{Edd}	Eddington accretion factor (cf. Equation ??).	10
`nova_accretion_factor`	ϵ_{nova}	Nova accretion factor.	10^{-3}
`binary_evolution_CE_spin_flag`		Parameter controlling stellar spins after CE. 0: spins unaffected; 1: alignment and corotation of the spins.	0
`alpha_wind_accretion`	α_{WA}	Wind accretion parameter.	1.5
`beta_wind_accretion`	β_{W}	Wind ejection speed parameter.	0.125

Table 2: Description of global user-adjustable code parameters.

Name	Symbol	Description	Default value
common_envelope_alpha	$\alpha_{\rm CE}$	CE α parameter.	1.0
common_envelope_timescale	$\tau_{\dot{m},\,\rm CE}$	Mass-loss timescale to shed the envelope during CE evolution.	10^2 yr
dynamical_mass_transfer_low_mass_donor_timescale	$\tau_{\dot{m},\,\rm dyn\,MS}$	Mass-loss timescale for dynamical mass transfer from a low-mass MS star.	10^2 yr
dynamical_mass_transfer_WD_donor_timescale	$\tau_{\dot{m},\,\rm dyn\,WD}$	Mass-loss timescale for dynamical mass transfer from a WD.	10^2 yr
compact_object_disruption_mass_loss_timescale	$\tau_{\dot{m},\,\rm expl}$	Mass-loss timescale for an explosive thermonuclear event.	10^2 yr
kick_distribution_sigma_km_s_NS	$\sigma_{\rm kick,\,NS}$	Width of Maxwellian NS kick speed distribution (in units of km/s)	$265\,{\rm km\,s^{-1}}$
kick_distribution_sigma_km_s_BH	$\sigma_{\rm kick,\,BH}$	Width of Maxwellian BH kick speed distribution (in units of km/s)	$50\,{\rm km\,s^{-1}}$

Table 3: Description of particle-specific user-adjustable code parameters.