User manual for the Multiple Stellar Evolution (MSE) code

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

https://github.com/hamers/mse
September 30, 2020

Contents

1 Introduction

This is the user manual for 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 uses some Fortran routines. A Python interface is included for easy use. The source code is available[1] at https://github.com/hamers/mse (it is assumed that you have `git` installed). To download it, you can use the following command:

```
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 Python (2/3) 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 your C++ and Fortran compilers.

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.

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!
```

[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.

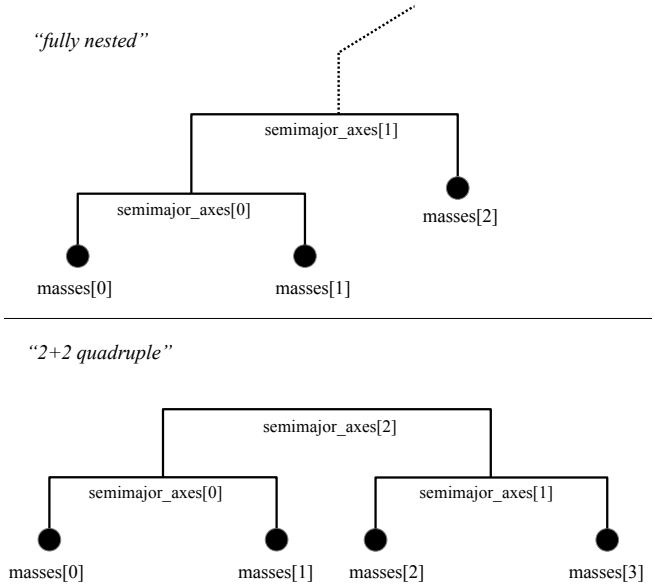

Figure 1: Illustration of currently supported configurations within the `run_system.py` script and the `evolve_system()` wrapper function. Also shown are designations within the argument lists (displayed are `masses` and `semimajor_axes`, but the same orders apply to other properties of bodies/orbits).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

> 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 – Python

MSE can be used with the Python interface (recommended for most users), but also directly within C++ (recommended for advanced users only)[2]. 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 accessible fully within the command line. It produces basic plots with the time evolution of some quantities, mobile diagrams, and positions. 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).

[2] At the moment of writing, MSE has been most tested with the Python interface. Usage within C++ is less tested.

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

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

```
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"
```

Note that arrays are to be input using spaces to separate values. All masses should be given in $\mathrm{M}_\odot$, 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).

In the above example Code snippet 1, three masses are given, so the system is assumed to be a hierarchical triple. The order of the arrays 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 $\mathrm{M}_\odot$, respectively, and the tertiary mass is 2 $\mathrm{M}_\odot$. The metallicities are 0.02 for all stars. The inner and outer orbital semimajor axes are 15 and 120 AU, respectively, and the eccentricities are 0.1 and 0.2 for the inner and outer orbits, respectively. The inclinations of the inner and outer orbits are 0.001 and 1.4 rad, respectively. The longitudes of the ascending node (`--LANs`) and the arguments of periapsis (`--APs`) 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 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 in this case). 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 above within a user-created Python script (not using the command line directly). In fact, as can be immediately inferred by inspecting `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 could be used as shown in Code snippet 2.

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

```
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")
```

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 AMUSE[3].

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

```
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)
```

[3] amusecode.org

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

We give a simple example in Code snippet 3, initialising and running the same system that was used in Section 4.1. The function `create_nested_multiple` from the `Tools` class is used (lines 12–15) to generate the `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 made (line 17), and the `particles` are added to the code (line 18).

Lines 25–34 represent a time loop in which the code is called repeatedly until reaching the 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 depend on `Nsteps`.

In line 36, the `code.log` is accessed. 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 these by running the high-level time loop (lines 25–34 in Code snippet 3), which is particularly useful for population synthesis. The contents of the code logs are described in detail in Section 7.

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 – C++

Using the code within C++ gives most customisability and minimises any potential overhead losses generated 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.

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

```
ParticlesMap particlesMap;
int N_bodies = 3;
double masses[3] = {40.0,10.0,2.0};
int stellar_types[3] = {1,1,1};
double smas[2] = {15.0,120.0};
double es[2] = {0.1,0.2};
double TAs[2] = {0.01,0.01,0.01};
double INCLs[2] = {0.001,1.4};
```

```
double APs[2] = {0.01,0.01};
double LANs[2] = {0.01,0.01};

create_nested_system(particlesMap,N_bodies,masses,stellar_types, \
    smas,es,TAs,INCLs,APs,LANs);

initialize_code(&particlesMap);

double t=0.0;
double tend = 2.0e7;
int Nsteps = 2000;
double dt = tend/ ((double) Nsteps);
double output_time,hamiltonian;
int state,CVODE_flag,CVODE_error_code;

while (t<tend)
{
    evolve(&particlesMap,t,t+dt,&output_time, \
        &hamiltonian,&state,&CVODE_flag,&CVODE_error_code, \
        &integration_flag);
    print_system(&particlesMap,integration_flag);
    t += dt;
}

clear_particles(&particlesMap);
```

6 Particle sets

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 initial system is assumed to be hierarchical so it should contain orbit particles (if not, the system is assumed to be unbound from the beginning). Due to various processes, orbits can become unbound, and/or new orbits could form, and this is determined by analysing the system after direct N-body integration. The structure of the system is therefore dynamic.

7 Code logs

8 Code parameters

9 Citing the code in scientific publications

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 1: Description of global user-adjustable code parameters in MSE.

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 2: Description of star or orbit-specific user-adjustable code parameters in MSE.