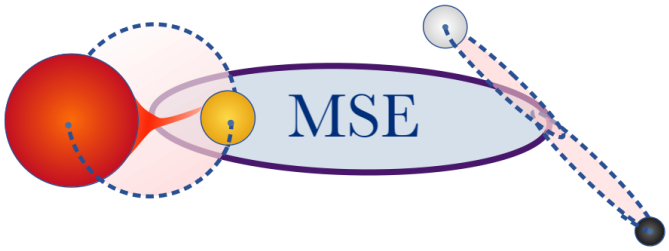
User guide to the Multiple Stellar Evolution (MSE) code

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

https://github.com/hamers/mse
April 21, 2022



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

2.1 General

MSE is written primarily in C and C++ but also links to some Fortran routines (from SSE, and a small number from 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 of `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, 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 `gcc`, `g++`, and `gfortran` are available on the system. To compile, simply type in the main directory (which contains the `Makefile`)

```
make
```

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

```
CC = gcc
CXX = g++
FC = gfortran
```

in the `Makefile` accordingly to correspond to the compilers that you would like to use. This can vary from operating system (see below for Mac-specific instructions).

> The default `CPPFLAGS` line in the `Makefile` includes optimisation. When using a debugger (e.g., `gdb`), the alternative `CPPFLAGS` line should be activated (this will make the code run slower).

The compilation will typically take $\sim$10-20 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.

[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.

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 compilation errors occur.

2.2 Mac users

MSE is known to work on macOS 10.15.7 (Intel x86-64) and macOS 12.2 (M1 arm-64). In both cases, you will first need to install XCode (free for download at the Apple Store) and Apple Command Line Tools. The latter can be installed via the terminal command

```
xcode-select --install
```

Next, GCC (version 11 is known to work) and PYTHON3 will need to be installed. The recommended method is via HOMEBREW; installation instructions can be found here: https://brew.sh. Remember to install at least NUMPY together with PYTHON3; it is recommended to also install `Matplotlib`.

2.2.1 Intel-mac

WARNING: Using GCC as the C compiler is known **not** to work on Intel-mac. Instead, use the default Apple CLANG, i.e., the compiler options should be (assuming gcc version 11)

```
CC = clang
CXX = g++-11
FC = gfortran-11
```

Be aware that g++-11 is used here as the C++ compiler: this refers to G++ version 11 as installed via HOMEBREW (it could be that the terminal command G++, without the '-11', is linked to CLANG, the default Apple compiler, and the latter is known **not** to work as the C++ compiler).

2.2.2 Mac arm-64

On (newer) mac devices with the arm-64 architecture (e.g., M1 chip), different compiler settings are required. First, install HOMEBREW as for Intel-mac and GCC (version 11 is known to work), but do not install PYTHON3 via HOMEBREW. Instead, install CONDA-MINIFORGE (see also this web page; 'Option 2'):

```
brew install miniforge
```

At the moment of writing, MINIFORGE has the large advantage of being natively compiled for the arm architecture (the same applies to GCC available via HOMEBREW; note that the versions of GCC and PYTHON used should be architecture compatible with each other).

Next, create a virtual environment (here named 'test_env' for illustration, and assuming PYTHON version 3.10):

```
conda create --name test_env python=3.10
```

The required NUMPY package can now be installed via

```
conda install numpy
```

Other packages (e.g., MATPLOTLIB) work similarly. Note that, in a new terminal, you will need to activate this virtual environment:

```
conda activate test_env
```

(assuming the name of the environment you intend to use for MSE is 'test_env'; it can be convenient to add this to your bash profile).

Finally, the compiler options for the arm-64 installation as described above are

```
CC = gcc-11
CXX = g++-11
FC = gfortran-11
```

3 Testing the installation

To make sure that the code has been properly installed and is running correctly, 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!
```

> Include the argument '`--verbose`' for more verbose output, and '`--plot`' to show plots when applicable. 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`'. Note that the test numbered 100 is actually a suite of unit tests specific to C and C++-related parts of the code. The latter are described in SRC/TEST.CPP.

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). 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, positions of the stars/bodies, and an HR diagram.

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
```

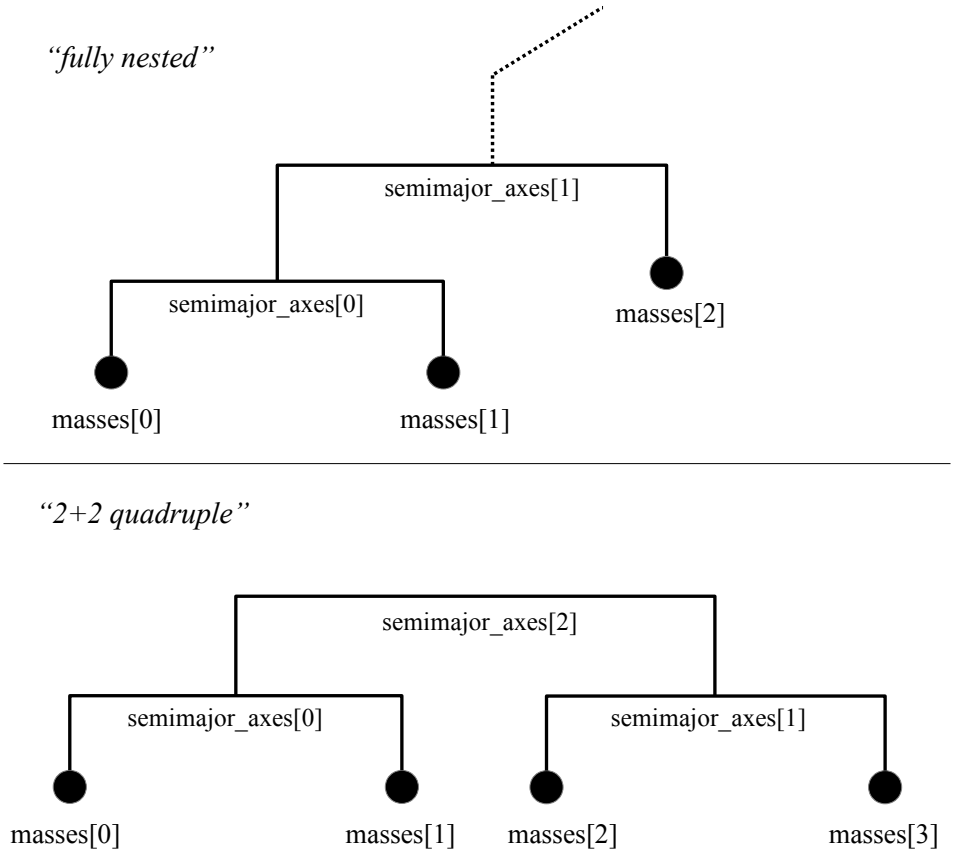


Figure 1: Illustration of the `fully_nested` and `2+2_quadruple` configurations that can be used the `run_system.py` script and the `evolve_system()` wrapper function by specifying the `configuration` parameter. Also shown are designations within the argument lists (displayed are `masses` and `semimajor_axes`, 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. Note that other configurations can also be specified (see Section 4.1.1).

```
--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). Other configurations can also be input (see Section 4.1.1 below).

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.

Some of the arguments in `run_system.py`, for example the metallicities, arguments of periapsis, or the longitudes of the ascending nodes, can be omitted. In that case, default or random values will be assumed. A full summary of the assumed initial conditions is printed when starting the evolution.

By default, the bodies in the system are assumed to be stars. Optionally, evolved stars (using the SSE stellar type integers) can be specified with the argument list `-stellar_types` (with the order corresponding to `--masses`). Note that the starting stellar type should be consistent with the initial mass (it is determined by running the star in isolation until the desired stellar type is reached). For example, it is not possible to set the stellar type to 14 (BH) if the initial mass is, say, $0.8\,M_\odot$, or 7 (stripped He star) since the latter requires binary interaction in MSE. If the specified stellar type is inconsistent, then another initial stellar type will be assigned (depends on the situation).

Planets (and, in the future, other objects) can be specified with the optional argument `--object_types`, which should be a list of integers corresponding to `--masses`; an integer value of 1 (default value) indicates a star, and an integer value of 2 indicates a planet. For a given star the `stellar_type` are not used when `object_type` is not 1.

> 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 start with initially unstable systems (as determined by the stability criterion implemented in the code). 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, four 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.
- `[plot_filename]_HRD.pdf`: Hertzsprung-Russell diagram of all stars in the system.

If the command line option `--save_data` is given to `run_system.py` (enabled by default; disable with `--no-save_data`), some of the code's output will also be saved to disk as a PYTHON `Pickle` file. This is convenient for later replotting or analysing the same system, without rerunning the system. To only load previously-generated data and generate plots, call `run_system.py` with the command line option `--plot_only` (obviously, the system needs to have run beforehand, so make sure the `plot_filename` is correct and the `Pickle` datafile exists).

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.

Custom configurations Configurations other than 'fully nested' and '2+2 quadruple' can be specified with the `--configuration` command line argument, which should be a string. Below, we explain how to translate a given hierarchical multiple system (as represented in a mobile diagram) into a configuration string.

1. Draw the tree diagram of your hierarchy, as shown in the examples in Fig. 2. As much as possible, try to keep the most number of sub-binaries in the right branches (there can be *only* two branches per node).
2. While initializing the `configuration` parameter, any *binary* (two children) must be enclosed in square brackets. For a single child, no square brackets should be used.
3. If the hierarchy is *fully nested*, the `configuration` parameter should be set to `'N'`, where N is the total number of stars (see Fig. 2(a)). Even if any sub-branch is fully nested, a similar notation can be followed (see Fig. 2(b)).
4. For *almost* fully nested configurations (i.e. fully nested upto a point, after which one branch has a different configuration), use the `'{N+other}'` notation (see Fig. 2(c)). The meaning of this notation is as follows - `'{N+other}'` $\equiv$ `'[1, . . . [1, other] . . . ]'`, where the ... represent a repetition of N times. This notation can also be used multiple times (see Fig. 2(d)), but keep proper bracketing in mind !

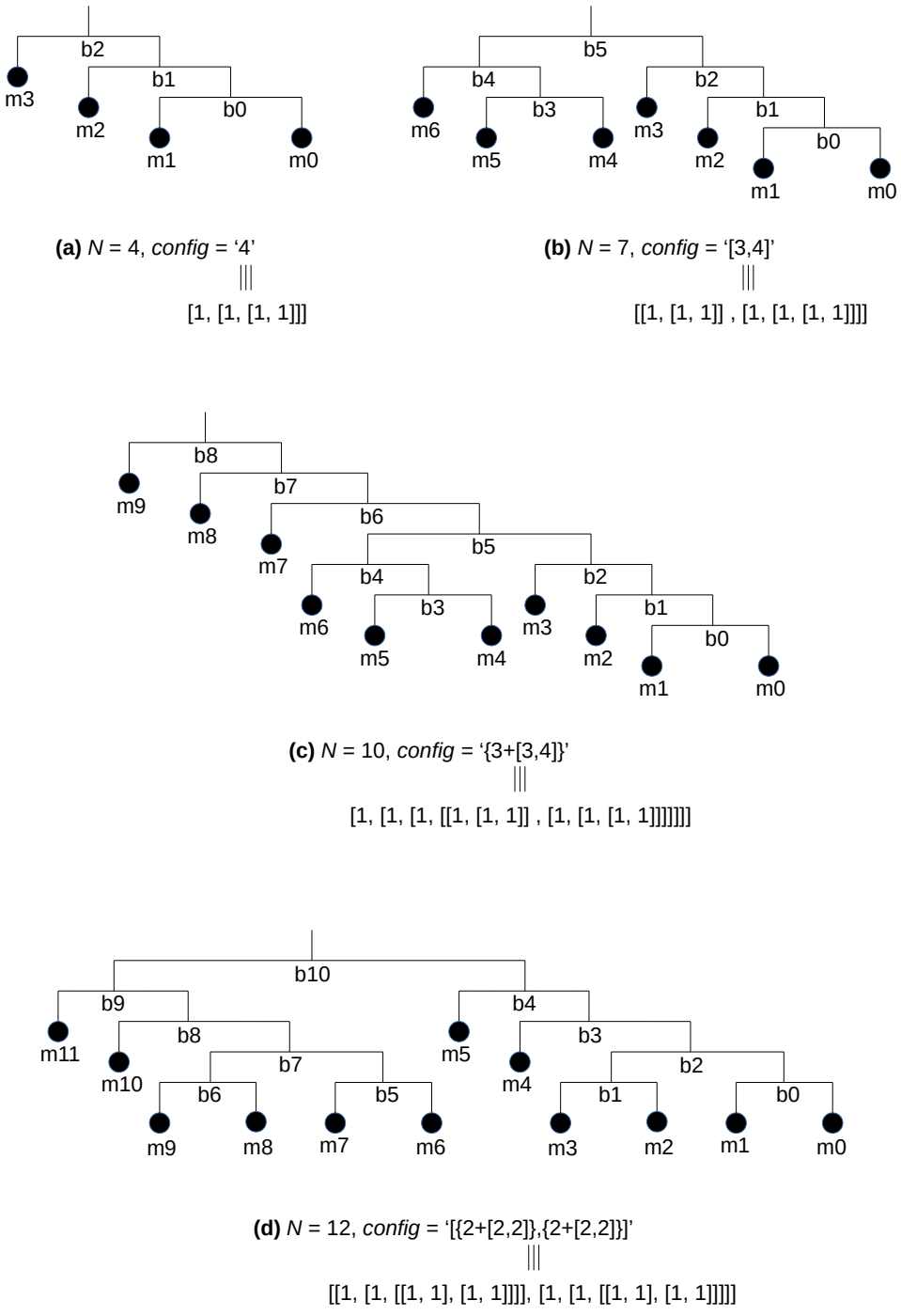


Figure 2: Selected tree diagrams of multiple star system hierarchies to explain the notation. Note the indexing of the masses (mX) and binaries (bX), where X denotes the index. Also, note the verbose configuration list corresponding to each configuration. (a) A fully nested hierarchy. (b) A hierarchy, with two sub-branches, both of which are fully nested. (c) An *almost* fully nested hierarchy. (d) Two *almost* fully nested hierarchies. (note the indexing!)

5. The indexing of masses and binary nodes is done from the bottom right. It is crucial to follow this numbering while initializing the list parameters such as masses, metallicities, semimajor axes, eccentricities, inclinations, longitudes of the ascending node, and arguments of periapsis.

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[3].

```
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_fully_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
```

[3] amusecode.org

```
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])
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_fully_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 running the code repeatedly.

5 Code usage within C++

Using the code within C++ gives most customisability and eliminates 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.

```
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);
```

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

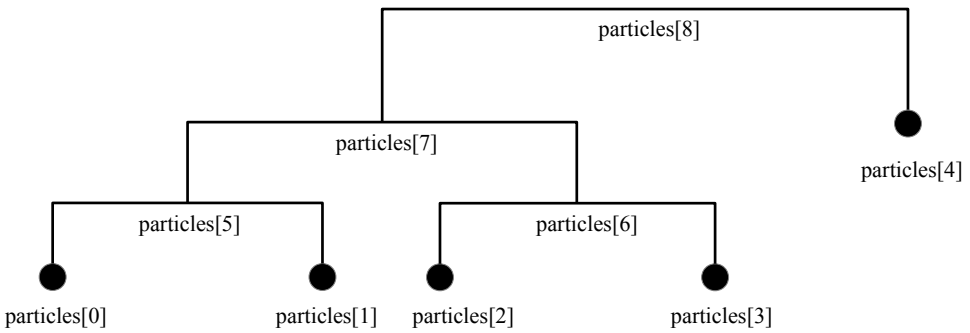


Figure 3: Mobile diagram of a hierarchical quintuple. Indicated are which elements of the particles set the bodies/binaries are assigned to.

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 automatically produce `particles` in Python. In particular, `create_fully_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 are `create_2p2_quadruple_system()`, which generates a 2+2 quadruple system, and `create_hierarchy()`, which can be used to generate *any* nested hierarchical configuration (see Section 4.1.1).

Nevertheless, we here give a description to produce `particles` sets in general, for those users wanting to generate `particles` sets on their own. As mentioned above, any hierarchical configuration is allowed in MSE, 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. 3.

```
1  N_bodies = 5
2  N_binaries = N_bodies-1
3
4  particles = []
5
6  ### Add the bodies ###
7  for index in range(N_bodies):
8      particle = Particle(is_binary=False,mass=masses[index])
9      particles.append(particle)
10
11 ### Add the binaries ###
12 for index in range(N_binaries):
13     if index==0: ### corresponds to particles[5]
14         child1 = particles[0]
15         child2 = particles[1]
16     elif index==1: ### corresponds to particles[6]
```

```
17          child1 = particles[2]
18          child2 = particles[3]
19      elif index==2: ### corresponds to particles[7]
20          child1 = particles[5]
21          child2 = particles[6]
22      elif index==3: ### corresponds to particles[8]
23          child1 = particles[7]
24          child2 = particles[4]
25
26      particle = Particle(is_binary=True,child1=child1,\
27          child2=child2,a=semimajor_axes[index],\
28          e=eccentricities[index],INCL=inclinations[index],\
29          AP=arguments_of_pericentre[index],\
30          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. 3.

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 (object_type equal to 1). Optionally, the metallicity can be specified (particle.metallicity = ...). Initial stellar radii are determined from the SSE tracks. To model a planet, one should set object_type to be 2. The radius should then be set with particle.radius, as well as the tidal evolution parameters.

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 each be a Particle; in the example, they are assigned to previously-created elements of particles in such a manner to produce the structure depicted in Fig. 3. 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. 3, 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
```

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)
12	Beginning of WD kick event
13	Beginning of WD kick event
14	Beginning of triple CE event
15	End of triple CE event
16	Formation of millisecond pulsar
17	Final state

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

The `log` is a Python `list` with each element representing an event. The elements are 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 1 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.
- `particles`: the current `particles` set. Note that it contains several key parameters from the particles set used internally in the code (in C++), but not all of them.

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 the form of several tables.

9 Error codes and maximum wall time

If an error occurs during the integration, the code will stop prematurely. Its `error_code`, accessed in Python as `code.error_code`, will then be non-zero. See `src/parameters.cpp` for a description of the different error codes.

Among the error codes are 35 and 36, which are not errors in the strictest sense, but indicate that the wall time exceeded the maximum allowed value. The latter is controlled by the parameter `max_wall_time_s`, i.e., the maximum wall time in seconds (default value: 1.8×10^4 s, or 5 hr).

10 Citing the code in scientific publications

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

- Hamers, A.S., Rantala, A., Neunteufel, P., Preece, H., Vynatheya, P. *Multiple Stellar Evolution: a population synthesis algorithm to model the stellar, binary, and dynamical evolution of multiple-star systems*. 2021, MNRAS, Volume 502, Issue 3.

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*. 2000, 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*. 2002, 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*. 2020, 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.

11 Q&A

The code is slow! What can I do?

Be more patient :-). More seriously: MSE can be quite 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. This is contrast to binary population synthesis codes such as BSE, which use a simplified/approximated method to solve the ODEs (e.g., describing orbital evolution due to gravitational wave radiation), and obviously do not include direct N-body integration.

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.

The code crashes!

Unfortunately, due to the complexity of the code, it is possible that hard crashes occur due to bugs (e.g., PYTHON giving an error, or, worse, a segmentation fault). In most cases, it is expected that the code will return a non-zero error code (`code.error_code`; see `src/parameters.cpp` for a description of the different codes).

If you encounter a bug, please report the problem, preferably by creating an *Issue* ticket on `GitHub`, or by sending an email. Please describe accurately the initial conditions used, the version of the code used, and give the error code (if available) and print output prior to the error. Additional verbose output from the code can be produced by changing the `verbose_flag` code parameter accordingly (see Table 3).

The `error_code` in the case of segmentation faults should be -1, and control should be returned to Python (without the code hard crashing). If for some reason this does not occur and the code hard crashes due to a segmentation fault, it would be helpful if you used a debugger to get more information and to aid the debugging process. First, in order to be able to use a debugger, the code should be compiled with the compiler flag `CPPFLAGS` in the `Makefile` set to the second option (where it is indicated that it supports debugging). Note that the outcome of individual systems can be different when using the different `CPPFLAGS` (this is related to slight changes in the way calculations are carried out with compiler optimisations enabled; the latter are excluded in debugging mode).

Assuming Python usage and that you have `gdb` installed:

```
gdb
(gdb) file python3
(gdb) run X
```

Here, 'X' is the name of your Python program (including command line arguments). If you are using `lldb`, the syntax would be:

```
lldb -f python3
(lldb) run X
```

I am getting different results when running exactly the same system on different machines/environments!

This is a known phenomenon. When running on different machines/environments (or even using different compiler options), results can differ because of the way that floating points are represented and rounded. Although these effects are tiny for single operations, they can add up and become significant when the number of floating point operations is very large, which typically happens when a system is integrated in MSE for a relatively long time. For example, if a triple system undergoes a very large number of ZLK oscillations, then the exact phase of the ZLK cycle in the inner binary at an arbitrary time can differ from machine to machine. This can make the difference whether or not a star in the inner binary will fill its Roche lobe as it evolves, and such a difference will propagate further in the evolution of the entire system. This effect is compounded by the often chaotic nature of the dynamics of >2-body-systems.

In practice, this implies that the exact outcome of one specific system should not be given too much emphasis or weight. MSE is meant to be used as a population synthesis code; uncertainties due to numerical difficulties as described above are expected to average out when a large enough sample of systems is considered (what constitutes 'large enough' should be investigated on a case-by-case basis, by considering convergence of the statistical properties of interest with respect to the number of included systems). We furthermore remark that, apart from the numerical difficulties described above, the exact outcome of a specific system also sensitively depends on physical processes included in the code, many of which are uncertain (for example, the precise way that an eccentric orbit responds to mass transfer).

12 Release notes

- v0.86 – minor bug fixes; first public release.
- v0.85 – Changed default parameter binary_evolution_mass_transfer_model in mse.py from 1 (modified Sepinsky) to 0 (Hamers & Dosopoulou 2019).

- v0.84 – Bug fix in stable triple mass transfer evolution (accretors were not aged correctly). Minor memory leak bug fix.
- v0.83 – Bug fixes in triple CE evolution; changed default value of `effective_radius_multiplication_factor_for_collisions_stars` to 1 (previously: 3).
- v0.82 – Numerous bug fixes; implemented a model for NS spinup/down (disabled by default); changed calculation of orbit-averaged mass transfer rate to be eccentricity dependent; changed the way the code handles stellar type changes after stopping conditions; implemented secular flybys (disabled by default).
- v0.81 – further bug fixes.
- v0.80 – a significant number of bug fixes, including a patched memory leak.
- v0.79 – first 'official' release.

Name	Symbol	Description	Default value
ODE integration parameters			
`relative_tolerance`		Relative tolerance used in ODE integration.	10^{-10}
`absolute_tolerance_eccentricity_vectors`		Absolute tolerance used for eccentricity vectors in ODE integration.	10^{-8}
`absolute_tolerance_angular_momentum_vectors`		Absolute tolerance used for angular momentum vectors in ODE integration (in code units).	10^{-2}
`absolute_tolerance_spin_vectors`		Absolute tolerance used for spin vectors in ODE integration (in code units).	10^{-4}
`include_quadrupole_order_terms`		Include pairwise quadrupole-order terms for secular dynamics.	`True`
`include_octupole_order_binary_pair_terms`		Include pairwise octupole-order terms for secular dynamics.	`True`
`include_octupole_order_binary_triplet_terms`		Include triplet octupole-order terms for secular dynamics.	`True`
`include_hexadecupole_order_binary_pair_terms`		Include pairwise hexadecupole-order terms for secular dynamics.	`True`
`include_dotriacontupole_order_binary_pair_terms`		Include pairwise dotriacontupole-order terms for secular dynamics.	`True`
`verbose_flag`		Specifies how much terminal output the code will give (from within C++; does not strictly pertain to ODE integration). 0: no verbose output; > 0: verbose output, with increasing verbosity (>1 will slow down the code considerably).	0

Table 2: Description of global user-adjustable code parameters related to the ODE integration.

Name	Symbol	Description	Default value
N-body integration 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.	0.05
`nbody_analysis_minimum_integration_time`		Minimum integration time allowed for analysis of systems in N-body integration mode.	10^1 (yr)
`nbody_analysis_maximum_integration_time`		Maximum integration time allowed for analysis of systems in N-body integration mode.	10^5 (yr)
`MSTAR_gbs_tolerance_default`		Tolerance used in the MSTAR code.	10^{-10}
`MSTAR_collision_tolerance`		Tolerance used for collision detection in the MSTAR code.	10^{-10}
`MSTAR_output_time_tolerance`		Tolerance used for output time convergence in the MSTAR code.	10^{-6}
`MSTAR_include_PN_X`		Turn on or off specific PN terms in MSTAR (bool). Here, '`X`' can be '`acc_10`': 1PN acceleration; '`acc_20`': 2PN acceleration; '`acc_25`': 2.5PN acceleration; '`acc_SO`': spin-orbit acceleration; '`acc_SS`': spin-spin acceleration; '`acc_Q`': quadrupole acceleration; '`spin_SO`': spin-orbit terms for spin evolution; '`spin_SS`': spin-spin terms for spin evolution; '`spin_Q`': quadrupole terms for spin evolution.	`True`

Table 3: Description of global user-adjustable code parameters related to direct N-body integration.

Name	Symbol	Description	Default value
Prescription parameters			
`eddington_accretion_factor`	$f_{\rm Edd}$	Eddington accretion factor.	10
`nova_accretion_factor`	$\epsilon_{\rm nova}$	Nova accretion factor.	10^{-3}
`binary_evolution_CE_energy_flag`		Normally, the total primary mass is used in the CE prescription. If this flag is 3: use primary core mass instead of total mass.	0
`binary_evolution_CE_spin_flag`		Parameter controlling stellar spins after CE. 0: spins unaffected; 1: alignment and corotation of the spins.	1
`alpha_wind_accretion`	$\alpha_{\rm WA}$	Wind accretion parameter.	1.5
`beta_wind_accretion`	$\beta_{\rm W}$	Wind ejection speed parameter.	0.125
`triple_mass_transfer_primary_star_accretion_efficiency_no_disk`	$\alpha_{\rm TMT,\,no\,disk,\,prim}$	Efficiency at which the primary star accretes mass from the tertiary star during triple mass transfer (no disk forms around inner binary).	0.1
`triple_mass_transfer_secondary_star_accretion_efficiency_no_disk`	$\alpha_{\rm TMT,\,no\,disk,\,sec}$	Efficiency at which the secondary star accretes mass from the tertiary star during triple mass transfer (no disk forms around inner binary).	0.1
`triple_mass_transfer_primary_star_accretion_efficiency_disk`	$\alpha_{\rm TMT,\,disk,\,prim}$	Efficiency at which the primary star accretes mass from the tertiary star during triple mass transfer (disk forms around inner binary).	0.9
`triple_mass_transfer_secondary_star_accretion_efficiency_disk`	$\alpha_{\rm TMT,\,disk,\,sec}$	Efficiency at which the secondary star accretes mass from the tertiary star during triple mass transfer (disk forms around inner binary).	0.9
`triple_mass_transfer_inner_binary_alpha_times_lambda`	$(\alpha\lambda)_{\rm TMT}$	Parameter controlling how the inner orbit responds to accretion from mass from a tertiary star.	5

Table 4: Description of global user-adjustable code parameters related to prescriptions.

Name	Symbol	Description	Default value
common_envelope_alpha	$\alpha_{\rm CE}$	CE α parameter (binary CE evolution).	1.0
triple_common_envelope_alpha		CE α parameter (triple CE evolution).	1.0
common_envelope_timescale	$\tau_{\dot{m},\,\rm CE}$	Mass-loss timescale to shed the envelope during CE evolution.	10^3 (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.	10^3 (yr)
dynamical_mass_transfer_WD_donor_timescale	$\tau_{\dot{m},\,\rm dyn,\,WD}$	Mass-loss timescale for dynamical mass transfer from a WD.	10^3 (yr)
compact_object_disruption_mass_loss_timescale	$\tau_{\dot{m},\,\rm expl}$	Mass-loss timescale for an explosive thermonuclear event.	10^3 (yr)
kick_distribution		SNe kick distribution specification. Between 1-5.	1
kick_distribution_sigma_km_s_NS		SNe kicks – default Maxwellian width for NSs	265 ($\rm km\,s^{-1}$)
kick_distribution_sigma_km_s_BH		SNe kicks – default Maxwellian width for BHs	50 ($\rm km\,s^{-1}$)
kick_distribution_2_m_NS		NS mass used in SNe kick prescription 2.	1.4 ($\rm M_\odot$)
kick_distribution_4_m_NS		NS mass used in SNe kick prescription 4.	1.2 ($\rm M_\odot$)
kick_distribution_4_m_ej		Ejecta mass used in SNe kick prescription 5.	9 ($\rm M_\odot$)
kick_distribution_5_v_km_s_NS		Speed parameter for NSs used in kick prescription 5.	400 ($\rm km\,s^{-1}$)
kick_distribution_5_v_km_s_BH		Speed parameter for BHs used in kick prescription 5.	200 ($\rm km\,s^{-1}$)

Table 5: Description of particle-specific user-adjustable code parameters.