Tutorial for Simulating Ultra Compact Dwarf Galaxies with the Astronomical Multipurpose Software Environment

Simon Portegies Zwart
Sterrewacht Leiden
Leiden University
Niels Bohrlaan 2
2333 CA, Leiden
the Netherlands

July 2, 2014

1.1 Getting started

Download AMUSE from `amusecode.org`. You can download the entire source code of the package, or just use the binary version for now. You should familiarize yourself with the `Python` programming language, but for this tutorial you will not need a black-belt. Details about AMUSE can be found in the literature, for example in [1] of a more technical and philosophical description, and in [2] for a more astronomical view. But of course you can also read the syllabus (available at the website) or the online manual.

1.2 Solving the Gravity

A simplest possible script to integrate Newton's equations of motion is given in List. 1.1. This script starts by including the required AMUSE package `lab` and the python option parser.

```
1 from amuse.lab import *
```

The `amuse.lab` incorporates most of AMUSE into your python environment, including most usual n-body codes like `Hermite`, `Huayno`, `Mercury`, `BHTree`, `PhiGRAPE` and `Bonsai`.

In listing 1.2 we present a typical program, with a command line interface and the main calling routines.

The combination of listings. 1.1 and 1.2 together including the AMUSE package makes for a complete program that may be run and used for further experimentation. The listing of this program is called `CDG_gd.py`, and is available in the AMUSE examples directory.

Assignment 1 run the script and measure the time it takes to run 2, 4, 8, 16, 32, 65, 128, 256, 512 and 1024 bodies for 1 Myr. Also measure the energy error as a function of N

At the moment I prefer to use `matplotlib` for plotting, but use whatever package you like. A simple script to visualize you results is `plot_cluster.py`.

Assignment 2 Consider how the computer time scales with N, and if you are happy with the results. Also investigate the error in the energy. Is the energy error sufficiently small for your objected research project? Can you release the energy error, or does your calculation have better energy conservation? You can tune the error made by the code by setting the time-step parameter. For N-body codes that is done via

```
1   gravity.parameters.timestep = 0.03125
```

Several codes have the possibility to set softening parameters, opening angles or other constructs to tune the performance or accuracy of the code.

Assignment 3 So far you have been running with a 4th order Hermite predictor-corrector N-body integrator. This is only one of the many N-body integrators in AMUSE. Change the integrator to a Barnes-Hut tree code, by replacing the Hermite integrator by the BHTree integrator.

The call to the integrator looks like this:

```
1     CDG = Hermite(converter)
```

Now redo the analysis from assignment 2.

```
def main(Ncl, mcl, rcl, W0, t_end, n_steps):
    converter = nbody_system.nbody_to_si(mcl, rcl)
    bodies = new_king_model(Ncl, W0, convert_nbody=converter)

    gravity = Hermite(converter)
    gravity.particles.add_particles(bodies)
    channel_from_gravity_to_framework = gravity.particles.
        new_channel_to(bodies)

    write_set_to_file(bodies.savepoint(0.0 | t_end.unit), "
        nbody.hdf5", "hdf5", append_to_file=False)

    Etot_init = gravity.kinetic_energy + gravity.
        potential_energy

    time = zero
    dt = t_end/float(n_steps)
    while time < t_end:
        Etot_prev = Etot_init
        time += dt

        gravity.evolve_model(time)
        channel_from_gravity_to_framework.copy()
        write_set_to_file(bodies.savepoint(time), "nbody.hdf5",
            "hdf5")

        Ekin = gravity.kinetic_energy
        Epot = gravity.potential_energy
        Etot = Ekin + Epot

        print "T=", time, "M=", bodies.mass.sum(), "E= ", Etot,
            "Q= ", Ekin/Epot,
        print "dE=", (Etot_init-Etot)/Etot, "ddE=", (Etot_prev-
            Etot)/Etot

    gravity.stop()
```

Listing 1.1. Main routine for solving the gravity of an N-body problem

1.3 Adding a mass function

So far, all stars in your simulation had the same mass. This is not terribly realistic. In listing 1.3 you can see how to introduce a mass function to your N-body system.

There is one caveat here: We have now coupled the stellar masses to the dynamical model, which means that you have lost the freedom to chose an arbitrary total mass of your cluster. Of course, we can easily correct for this, but let's continue with realistic, but somewhat smaller cluster. You could run with more particles (maybe a few 10 millions), but than you probably want to buy a parallel computer and/or a GPU and run with a more optimized code, like `Bonsai`, which we have run with up to 100 billion particles. This will probably not work on your laptop, but if you have access to `Titan` or `LGM` you should surely try it out.

```
def new_option_parser():
    from amuse.units.optparse import OptionParser
    result = OptionParser()
    result.add_option("-N", dest="Ncl", type="int",
           default = 100, help="number of stars [%default]")
    result.add_option("-t", unit=units.Myr, dest="t_end",
           type="float", default = 1|units.Myr,
           help="end time of the simulation [%default]")
    result.add_option("-n", dest="n_steps", type="float",
           default = 100, help="number of output steps [%
              default]")
    result.add_option("-m", unit=units.parsec, dest="mcl",
           type="float", default = 10**7|units.MSun,
           help="cluster mass [%default]")
    result.add_option("-r", unit=units.parsec, dest="rcl",
           type="float", default = 10|units.parsec,
           help="cluster half-mass radius [%default]")
    result.add_option("-W", dest="W0", type="float", default =
       7.0,
           help="King model structure parameter [%default]")
    return result

if __name__ in ('__main__'):
    set_printing_strategy("custom", preferred_units =
        [units.MSun, units.RSun, units.yr], precision = 4,
        prefix = "", separator = " [", suffix = "]")
    o, arguments  = new_option_parser().parse_args()
    main(**o.__dict__)
```

Listing 1.2. command line options and main calling sequence

```
    masses = new_salpeter_mass_distribution(Ncl, 0.1|units.MSun
       , 100|units.MSun)
    converter = nbody_system.nbody_to_si(masses.sum(), rcl)
    bodies = new_king_model(Ncl, W0, convert_nbody=converter)
    bodies.mass = masses
    bodies.scale_to_standard(convert_nbody=converter)
```

Listing 1.3. Making bodies with a mass function

```
    converter = nbody_system.nbody_to_si(mcl, rcl)
    bodies = new_king_model(Ncl, W0, convert_nbody=converter)
```

Listing 1.4. Making the bodies

```
class MilkyWay_galaxy(object):
    def get_gravity_at_point(self, eps, x,y,z):
        phi_0 = self.get_potential_at_point(eps, x,y,z)
        grav = AdaptingVectorQuantity()
        dpos = 0.001*(x**2+y**2+z**2).sqrt()
        phi_dx = self.get_potential_at_point(0,x+dpos,y,z) -
            phi_0
        phi_dy = self.get_potential_at_point(0,x,y+dpos,z) -
            phi_0
        phi_dz = self.get_potential_at_point(0,x,y, z+dpos) -
            phi_0
        return phi_dx/dpos, phi_dy/dpos, phi_dz/dpos

    def disk_and_bulge_potentials(self, x,y,z, a, b, mass):
        r = (x**2+y**2).sqrt()
        return constants.G * mass /\
            (r**2 + (a + (z**2 + b**2).sqrt())**2).sqrt()

    def halo_potential(self, x,y,z, Mc=5.0E+10|units.MSun, Rc
        =1.0|units.kpc**2):
        r=(x**2+y**2+z**2).sqrt()
        rr = (r/Rc)
        return -constants.G * (Mc/Rc)*(0.5*numpy.log(1 +rr**2)
            + numpy.arctan(rr)/rr)

    def get_potential_at_point(self, eps, x, y, z):
        pot_disk = self.disk_and_bulge_potentials(x,y,z,
            0.0|units.kpc, 0.277|units.kpc, 1.12E+10|units.MSun
                )
        pot_bulge = self.disk_and_bulge_potentials(x,y,z,
            3.7|units.kpc, 0.20|units.kpc, 8.07E+10|units.MSun)
        pot_halo = self.halo_potential(x,y,z,
            Mc=5.0E+10|units.MSun, Rc=6.0|units.kpc)
        return pot_disk + pot_bulge + pot_halo
```

Listing 1.5. axi-symmetric model of the galaxy potential.

Assignment 4 Replace the snippet from listing 1.4 by listing 1.3, debug and and run the code again.

1.4 Orbit in the Galaxy

Now we are going to add the Galaxy as a simple background potential. We adopt the relatively simple prescription by [3]. This can be implemented in `Python`, such as is done in listing 1.5.

Remember to include `numpy` as a separate package, otherwise it will not run.

We prefer not to copy this snipped of code to our earlier script, but rather incorporate it, just as we did with `amuse.lab` or `numpy`. We can save the Galaxy model as a separate script, which we can call `galaxy_model.py`, and then just include it as

```
from galaxy_model import Milky Way_galaxy
```

```
    gravity = bridge.Bridge()
2   gravity.add_system(CDG, (MilkyWay_galaxy(),) )
    dt = t_end/float(n_steps)
4   gravity.timestep = dt
```

Listing 1.6. Bridging the cluster with the Galaxy

Before we can tell the N-body code how to interact with the Galaxy model, we have to give the cluster a position and velocity in the Galaxy. We do this simply by adding the appropriate portion and velocity to all bodies in the cluster.

```
1   bodies.x += Rgal
    bodies.vy += vgal
```

The best place to do this is probably directly after the bodies have received their position in the King model [4], but before the bodies are given to the N-body code.

Now we can initiate the `Bridge` [5]. Bridge relies on the possibility that the combined Hamiltonian of a system can be split in two or more parts. The separate components are subsequently integrated using a 2nd order solver.

For convenience we have renamed the instantiation of the N-body code for solving the equations of motion of the cluster from `gravity` to `CDG`, so that we do not have to make any further changes to the script.

Assignment 5 Incorporate these changes and run a few simulations. Time the code again to investigate how much time is really spend in the `Bridge`. How much time is spend in the `Bridge`, and what is the AMUSE overhead. How does the amount of overhead depend on the number of particles in your simulation?

Assignment 6 Play around with the cluster orbit, for example by changing the number of particles, the orbital velocity or the position of the cluster in the Galaxy. And, of course, have a look at the results with the plotting script.

1.5 Simulating an ultra compact dwarf galaxy

So far our stars did have a mass from a mass function, but the masses and radii of the stars remained the same (in fact our stars were point-masses). This is not terribly realistic, and we can improve on this rather easily by yet starting up another code, in this case a stellar evolution code. For efficiency we will opt for a parameterized stellar evolution code, in this example we use `SeBa` [6].

In order to realize this improvement you have to make two changes to your script. First of all you have to start the stellar evolution code, just like you have started the N-body code. In listing 1.7 we see how the stellar evolution code is initiated. The proper communication between the stellar evolution code and your script require opening a `channel`. This new channel should link the `bodies` to the stars in the memory of the stellar evolution code.

It does not really matter where incorporate listing 1.7 to start the stellar evolution, so long as you have already initialized the stellar masses.

In the event loop we now add the `evolve_model` for the stellar evolution.

```
        stellar.evolve_model(time)
2       channel_from_stellar_to_framework.copy()
```

```
    stellar = SeBa()
    stellar.particles.add_particles(bodies)
    channel_from_stellar_to_framework = stellar.particles.
        new_channel_to(bodies)
    channel_from_stellar_to_framework.copy()
```

Listing 1.7. Starting the stellar evolution

After running this script the cluster will orbit in the potential of the Galaxy and the stars will evolve to. Instead of just showing a movie of the moving stars we can now also present a movie of the HRD of the cluster. It will be a rather boring HRD, because we have not yet incorporated binaries or stellar collisions.

1.6 Final comments

Now you can start simulating compact dwarf galaxies. It would be fun to see what fraction of the mass is lost in the encounter with a the Galaxy, or to what degree the left over bound system is mass segregated. But you can also replace components, such as running with a different N-body code or another stellar evolution code. You can even replace the static galaxy model by a more realistic N-body realization of the Galaxy. You will see that finding a balance between performance, science questions and results are not easy to address.

With the AMUSE framework we hope to achieve two objectives: (1) get away from private software and (2) reduce the development time for scientific production software. I hope that you will enjoy using this little script, and AMUSE.

[1] S. Portegies Zwart, S. L. W. McMillan, E. van Elteren, I. Pelupessy, N. de Vries, *Computer Physics Communications* **183**, 456 (2013).
[2] F. I. Pelupessy, et al., *A&A* **557**, A84 (2013).
[3] B. Paczynski, *ApJ* **348**, 485 (1990).
[4] I. R. King, *AJ* **71**, 64 (1966).
[5] M. Fujii, M. Iwasawa, Y. Funato, J. Makino, *Publ. Astr. Soc. Japan* **59**, 1095 (2007).
[6] S. F. Portegies Zwart, F. Verbunt, *A&A* **309**, 179 (1996).