Project for Alice Chen: Instability in multi-planet systems

The goal of this project is to familiarize yourself with the orbital integration software you will be using for the summer, to learn to analyze the outputs from those simulations, and to develop some of the tools you will use for the next few months. A good task toward this end is to reproduce Fig. 1 of Chambers et al. 1996, The Stability of Multi-Planet Systems.

Specifics:

To install REBOUND:

The computer you're using (in the lab or your own) will have to have git installed, and have access rights to install programs. If this is not the case, let me or professor Green know and we can help you get it set up. Then:

1) Using a terminal, navigate to the directory in which you want to install rebound (it will make a folder named rebound, so you don't have to make a folder for it ahead of time). Now run the following commands
2) git clone git@github.com:hannorein/rebound.git (this will get the code)
3) cd rebound
4) git checkout –b mikkola origin/mikkola (this will switch to the correct branch of the code)
5) virtualenv venv && source venv/bin/activate (this keeps your installation separate from everything else. If you by any chance get a warning from something called Anaconda, let me know and we have to do something slightly different.)
6) pip install –e ./
7) pip install numpy
8) pip install scipy
9) pip install matplotlib

Now test this by opening python on the command line (by just entering python), and enter 'import rebound'. If this doesn't bring up any errors, the install succeeded.

Note: to use rebound, you will now always have to go to this rebound directory. If you enter ls, you will see that there is a folder there called venv for the virtual environment. In order to access rebound, you will always have to enter

source venv/bin/activate

before trying to execute any code that calls rebound. The command just activates the rebound installation. You should notice that when you do this, your command prompt now has venv in parentheses in front of it.

The setup:

To reproduce this plot, you need to run several simulations with 3 planets, spaced by different amounts. Additionally, you'll have to check at what point things get unstable and record the time it took for this to happen (that goes on the y axis).

Simple first case

Make a simulation of the Earth (at 1 AU) on a circular orbit around the Sun.

Follow the example in the folder where you installed rebound under python_examples/simple/problem.py for how to change units, add particles, and to integrate

To work in units of years, AU (distance from Earth to Sun), and solar masses, you want to set G = 4*PI^2

Output the coordinates every few timesteps (check python_examples/outersolarsystem/problem.py for how to do this)
Make a plot of x vs y to make sure you get a circle (when you plot in matplotlib, figure out how to set aspect ratio to 1)

Make a plot of a vs t and e vs t to make sure you get a flat line. To do that, for each output you would call, e.g.

```
o = particle[1].get_orbit()
print(o.a, o.e)
```

Single Integration

Write a function that takes a tuple of parameters for the integration, and then call it, i.e.

```
def integration(parameters):
        mass, separation, tmax = parameters
        …

parameters = (10**(-4), 2.5, 1.e4)
tinst = integration(parameters)
```

Use "ias15" for the integrator. Integrate up to tmax years. Make sure you call rebound.reset() in the integration function, set G as above, timestep to 0.01 (again see examples for how to do this—timestep=dt).

Write a function that calculates the mutual Hill radius from the masses of the planet and star—see the equation in Chambers et al. 1996.

Intialize particles in integration() to semimajor axes of 1 AU, 1 + separation*Hill radius, 1 + 2*separation*Hill radius (so separation is the number of Hill radii you want to separate by).

Write a function that checks if any of the semimajor axes change by more than 20% (a proxy for the system going unstable). Write this function and integration() such that if this condition is met, integration() returns the current time, otherwise (if you reach the end of the simulation at tmax) return tmax.

For this simulation, use the parameters in the snippet of code above.

Doing the problem

Having written a function as above, you can now easily parallelize doing many calls to integration. You can follow the setup of python_examples/2body/problem.py, which also sets up a similar simulation function, then makes a list of, in this case, timestep values and eccentricity values (dt and e0)—you would instead just make a list of separation values, e.g. parameters would look something like ((10**(-4),2.0,1.e4,), (10**(-4, 2.1, 1.e4), …). Then in the for loop (you woulnd't need a for loop), it makes a list of parameter tuples. The next few calls to pool and pool.map then will do the calculations on as many processors as available (change InterruptiblePool(12) to InterruptiblePool()). You can also get some ideas there about how to plot things in a nice way.

For all of them, always use mass = 10**(-4), tmax = 1.e4, and then vary the separation by the range in Fig. 1 of Chambers et al. 1996. Then make a log-log plot like they do, and fit a straight line to the data.

Experiment with the number of values you scan in separation, as well as with increasing tmax to longer times. Once your plot stops changing as you increase tmax, then you've chosen a sufficiently long tmax. (don't go longer than 10^5).

You should see a roughly straight line in a log-log plot, but you should also see disp at some distances beyod a separation of 3.5 Hil radii.

If you have trouble, try looking online (it's always best if you can teach it to yourself), but definitely let me know if you get stuck and I'll be happy to help. If you have time, feel free to mess around with whatever seems interesting to you!