TRAPPIST-1 N-Body Simulation Code

Last Edit: 14 May 2022

REBOUND based code for TRAPPIST-1 N-body simulations, written as part of Geological Sciences Honors Thesis (Sebastian Perez-Lopez, 2022). Code pre-loaded with TRAPPIST-1 orbital parameters for seven terrestrial planets based on Agol et al. 2021 and Raymond et al., 2022, but can be easily modified to simulate any other exoplanet system with known/estimated orbital parameters. Includes a feature for adding a theoretical planetesimal with any structure (e.g. takes width, start & end locations, # of planetesimals, planetesimal mass, starting orbital parameters, etc. as inputs). Also includes 'timestep' feature that pauses the simulation every X years and stores simulation data. Implemented custom collision resolver (direct mergers) to calculate impact velocities, as well as resonance argument data to track system stability.

Up to three data outputs that can be optionally run: 1) Simulation arhcive, which allows you to re-upload the simulation file and integrate for more time. (In essence, allows you to split up long integration timescales over multiple runs).
2) Transfer start and end system data as well as collisional & resonance data to an excel spreadsheet.
3) Transfer timestep data to an excel spreadsheet (code has feature to pause simulation X times through integration to rip out data for all surviving bodies). Useful for evaluating system evolution throughout integration.

DEFINE FUNCTIONS NECESSARY FOR INTEGRATION:

In [ ]:

```
# Function that codes for power law distribution. Used for randomly assigning initial semimajor axes (a)
# to planetesimals.
def rand_powerlaw(slope, min_v, max_v):
    y = np.random.uniform()
    pow_max = pow(max_v, slope+1.)
    pow_min = pow(min_v, slope+1.)
    return pow((pow_max-pow_min)*y + pow_min, 1./(slope+1.))

# Function that codes for uniform distribution. Used for randomly assigning initial true anomalies (f)
# to planetesimals.
def rand_uniform(minimum, maximum):
    return np.random.uniform()*(maximum-minimum)+minimum

# Function that codes for rayleigh distribution. Used for randomly assigning eccentricities (e) and inclinations (i)
# to planetesimals
def rand_rayleigh(sigma):
    return sigma*np.sqrt(-2*np.log(np.random.uniform()))

# Function that that shifts any degree value to between 0 and 360. E.g. 400 deg > 40 deg
def zeroTo360(val):
    while val < 0:
        val += 2*np.pi
    while val > 2*np.pi:
        val -= 2*np.pi
    return (val*180/np.pi)

# Function that shifts any degree value to -180 / + 180
def min180To180(val):
    while val < -np.pi:
        val += 2*np.pi
    while val > np.pi:
        val -= 2*np.pi
    return (val*180/np.pi)
```

SET UP SIMULATION FUNCTION. CHOOSE COLLISION RESOLVER, INTEGRATOR, TIMESTEP, ETC. INPUT PLANETARY ORBITAL PARAMETERS AND INITIAL PLANETESIMAL DISK:

ORBITAL ELEMENTS IN REBOUND:
d = radial distance from reference
v = velocity relative to central object's velocity
h = specific angular momentum
P = orbital period (negative if hyperbolic)
n = mean motion (negative if hyperbolic)
a = semimajor axis
e = eccentricity
inc = inclination
Omega = longitude of ascending node
omega = argument of pericenter
pomega = longitude of pericenter
f = true anomaly

M = mean anomaly

E = eccentric anomaly

l = mean longitude = Omega + omega + M

theta = true longitude = Omega + omega + f

T = time of pericenter passage

rhill = Hill radius, rhill = a * cbrt(m/3M)

In [ ]:

```
def setupSimulation():

    # Load REBOUND simulation structure
    sim = rebound.Simulation()

    # Set simulation units. Time = 'year', Distance = 'AU', Mass = 'Solar Mass'
    sim.units = ('yr', 'AU', 'Msun')

    # PICK INTEGRATOR HERE:
        # Standard: mercurius. Other options: ias15, whfast
    sim.integrator = "mercurius"

    # PICK MERCURIUS TIMESTEP HERE:
        # Standard: 0.0000082192
    sim.dt = 0.0000082192

    # PICK IAS15 MINIMUM TIMESTEP HERE:
        # Standard: 1e-6
    sim.ri_ias15.min_dt = (1e-6) # This ensures that close encounters do not stall the integration by setting
                                 # a minimal timestep for ias15.

    # PICK HILL FACTOR HERE:
        # Standard: 3
    sim.ri_mercurius.hillfac = 3 # This determiens how many hill radii away from a planet a planetesimal must be
                                 # for the code to trigger a 'close encounter' and switch to ias15.

    sim.testparticle_type = 1 # Set planetesimals to semi-active mode (only interact w/ active bodies, e.g. planets)

    sim.collision = "direct" # When any two particles overlap, a "collision error" is flagged

    sim.boundary = "open" # Open boundary conditions. Remove particle if they leave the box (are ejected from system)
    boxsize = 1 # AU
    sim.configure_box(boxsize)

    mEarth = 3e-6 # Earth mass in units of Msun
    rEarth = 4.25e-5 # Earth radius in units of AU

    # ASSIGN ORBITAL PARAMETERS HERE:
        # Mass, Radius, a, e, Omega derived from Agol et al., 2021
        # Mean Anomaly (M) and Omega derived from Raymond et al., 2022
            # Omega derived by subtracting omega values (Agol) from longitude of periastron values (Raymond)
    sim.add(m=.0898, hash="star") # Central star
    sim.add(m=1.374*mEarth, r=1.116*rEarth, a=0.01154, e=0.00305473, omega=2.35156489, M = 1.846294629,
            Omega = 2.074808, hash="b") # Trappist1b
    sim.add(m=1.308*mEarth, r=1.097*rEarth, a=0.01580, e=0.00055009, omega=0.01817982, M = 0.958157136,
            Omega = 2.296614, hash="c") # Trappist1c
    sim.add(m=0.388*mEarth, r=.788*rEarth, a=0.02227, e=0.00563298, omega=2.64777152, M = 2.991347207,
            Omega = 0.885749, hash="d") #Trappist1d
    sim.add(m=0.692*mEarth, r=.920*rEarth, a=0.02925, e=0.00632463, omega=0.81670784, M = 0.540630048,
            Omega = 0.098368, hash="e") #Trappist1e
    sim.add(m=1.039*mEarth, r=1.045*rEarth, a=0.03849, e=0.00841547, omega=0.06063985, M = 4.318657885,
            Omega = 2.907161, hash="f") #Trappist1f
    sim.add(m=1.321*mEarth, r=1.129*rEarth, a=0.04683, e=0.00400979, omega=0.32490512, M = 1.523298587,
            Omega = 5.888068, hash="g") #Trappist1g
    sim.add(m=0.326*mEarth, r=0.755*rEarth, a=0.06189, e=0.00365005, omega=0.0054794, M = 2.06968661,
            Omega = 2.999746, hash="h") #Trappist1h


    mTrappistH=0.326*mEarth
    aTrappistH = 0.06189
    mSun=.0898

    hillRadiusH = aTrappistH * ((mTrappistH / (3*mSun)) ** (1/3)) # Hill radius of planet h = .0009511 AU

    sim.N_active = sim.N # Declares amount of active bodies in the system (Central star + 7 planets)
                         # All other bodies added after are semi-active (plantesimals)

    N_pl = 1000 # Number of planetesimals in initial disk
    m_pl = 2*mEarth*(1e-5) # Mass of each planetesimal
    r_pl = 1.00446e-6 # Radius of each planetesimal

    np.random.seed(30) # By setting a seed you will reproduce the same simulation every time (easier for debugging).
                       # REMOVE SEED IF YOU WANT RANDOMIZED SIMULATION RUNS.
```

```
    # CONSTRUCT INITIAL PLANETESIMAL BELT HERE:
    for i in range(N_pl):
        # Disk .01 AU wide, starting at outer edge of planet h's hill sphere and extending for .01 AU
        a = rand_powerlaw(0, aTrappistH + hillRadiusH, aTrappistH + hillRadiusH + .01)
        e = rand_rayleigh(0.04)
        inc = rand_rayleigh(0.005)
        f = rand_uniform(-np.pi,np.pi)
        p = rebound.Particle(simulation=sim,primary=sim.particles[0],m=m_pl,r=r_pl,a=a,e=e,inc=inc,Omega=0,omega=0,f=f)
        # Only add planetesimal if it's far away from the planet
        d = np.linalg.norm(np.array(p.xyz)-np.array(sim.particles[1].xyz))
        planetesimalPeriapse = (1-e)*a
        if d > 0.01:
            sim.add(p)

    sim.move_to_com() # Move system to Center of Mass reference frame

    return sim
```

LOAD SIMULATION SET-UP FROM ABOVE, CHOOSE INTEGRATION RUN TIME. RUN SIMULATION. THIS CELL WILL OUTPUT LINES FOR EVERY COLLISION (TIME OF COLLISION, BODIES INVOLVED IN COLLISION, UPDATED PLANET MASS, AND IMPACT VELOCITY).

In [ ]:

```
%%time

# Import required packages
import rebound
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt
from rebound import hash as h
import pandas as pd

%matplotlib inline

# Number of plantesimals in the system (make sure it is same as in previous cell)
N_pl=1000

# Set-up simulation function from previous cell
sim = setupSimulation()

# Calculate intial energy of the system. Used to track energy change at end. If energy change is large, likely means
# the integrator is not resolving something correctly.
E0 = sim.calculate_energy()

# Initialize impacts on all bodies (zero at start)
starImpacts = 0; bImpacts = 0; cImpacts = 0; dImpacts = 0; eImpacts = 0; fImpacts = 0; gImpacts = 0; hImpacts = 0

# Set up pointers for all bodies using hashes
star = (sim.particles[h("star")]); b = (sim.particles[h("b")]); c = (sim.particles[h("c")]); d = (sim.particles[h("d")]);
e = (sim.particles[h("e")]); f = (sim.particles[h("f")]); g = (sim.particles[h("g")]); h = (sim.particles[h("h")])

# Initialize columns for dataframes
impactedBodies = []; impactTimes = []; impactVelocity = []; totalImpacts = []
bMass = []; cMass = []; dMass = []; eMass = []; fMass = []; gMass = []; hMass = [];
bEcc = []; cEcc = []; dEcc = []; eEcc = []; fEcc = []; gEcc = []; hEcc = [];
bX = []; cX = []; dX = []; eX = []; fX = []; gX = []; hX = [];
bY = []; cY = []; dY = []; eY = []; fY = []; gY = []; hY = [];
bZ = []; cZ = []; dZ = []; eZ = []; fZ = []; gZ = []; hZ = [];
bAU = []; cAU = []; dAU = []; eAU = []; fAU = []; gAU = []; hAU = [];

# Extract start-system data!
xyStart = np.zeros((N_pl+7, 6)); # np.zeros(all bodies in system, # of parameters of interest)
semimajorAxisStart = []; eccentricityStart = []; massStart = []; xStart = []; yStart = []; zStart = [];

for j,p in enumerate(sim.particles[1:]):
        xyStart[j] = [p.m,p.a,p.e,p.x,p.y,p.z]
        semimajorAxisStart.append(p.a)
        eccentricityStart.append(p.e)
        massStart.append(p.m)
        xStart.append(p.x)
        yStart.append(p.y)
        zStart.append(p.z)

# Create start-system dataframe (startdf) from values extracted above:
startdf = pd.DataFrame({'mass @ start':massStart, 'ecc @ start':eccentricityStart,
                        'semimajor axis @ start':semimajorAxisStart, 'x pos @ start':xStart,
                        'y pos @ start':yStart, 'z pos @ start':zStart})


# Initialize array that will store orbital information for all bodies throughout integration
xy = np.zeros((500, N_pl+7, 14))
```

```
    # 1st # = number of timesteps, 2nd # = # of bodies, 3rd # = # of orbital parameters of interest
times = np.linspace(0.,50000,500) # np.linspace(0, end time, # of frames)
    # Note: When running systems with planetesimals, use maximum ~500 frames or there will be too much data
    # to transfer to a text file.

# Begin integration:
for i,t in enumerate(times):
    # WHILE LOOP CODE THAT GOES THROUGH INTEGRATION, DATA EXTRACTION, AND MERGING
    for j,p in enumerate(sim.particles[1:]):
        xy[i][j] = [p.m,p.a,p.e,p.x,p.y,p.z,p.vx,p.vy,p.vz,p.l,p.Omega,p.omega,p.M,p.d]

    while (sim.t != t): # While you still haven't reached the end of the intended simulation time...
        try: # Continute to integrate to end of simulation time...
            sim.integrate(t)
            print('Timeframe output:', sim.t) # When it reaches end of timeframe, print out for real-time tracking

        except: # Except whenever you run into an error (collision), run the following code:
            collisionTimes = [] # Initialize collision time array

            for particle in sim.particles: # Loop through all particles in the simulation...
                collisionTimes.append(particle.lastcollision) # Gather last collision times (built-in to Rebound)
                                                              # for all particles.

            collisionTimes = np.array(collisionTimes)

            # Extract index of latest collision times to find two bodies involved in this condition
            max_idx = np.where(collisionTimes == max(collisionTimes))
                # There are two index values each time. The first value is the planet index, between 1 and 7.
                # The second value is the planetesimal index, between 8 and 1007.
                # We only need to extract these values at each exception and store them in max_idx.
                # Length of max_idx is always 2.

            # Only look at collisions involving at least one planet
                # Ignore fluke planetesimal-planetesimal collisions triggered by chance overlaps raising an error
            if max_idx[0][0] <= 7 or max_idx[0][1] <= 7:
                planet = sim.particles[int(max_idx[0][0])]
                planetesimal = sim.particles[int(max_idx[0][1])]

                # *** BELOW IS CODE TO EXTRACT IMPACT VELOCITIES ***

                # X, Y, Z velocity components of planet involved in collision:
                planetVelocityComp = [planet.vx, planet.vy, planet.vz]
                # X, Y, Z velocity components of planetesimal involved in collision:
                planetesimalVelocityComp = [planetesimal.vx, planetesimal.vy, planetesimal.vz]

                # Relative velocity components (x1-x2)^2, etc.
                relativeVelocityComp = [planetVelocityComp[0]-planetesimalVelocityComp[0],
                                        planetVelocityComp[1]-planetesimalVelocityComp[1],
                                        planetVelocityComp[2]-planetesimalVelocityComp[2]]

                relativeVelocity = sqrt((relativeVelocityComp[0])**2 +
                                        (relativeVelocityComp[1])**2 +
                                        (relativeVelocityComp[2])**2)

                #Convert velocity to relevant units (from AU/year to km/s):

                relativeVelocity = relativeVelocity*(1/365)*(1/24)*(1/60)*(1/60)*(1.496e8) #km/s
                collisionTime = sim.t #Earth years

                print('Time: ', sim.t, 'Bodies :', max_idx[0], planet.m, planetesimal.m, 'Relative Velocity :',
                      relativeVelocity) # Print out impact information for real-time tracking.

                # Update data frame columns for this collision.
                # Extract data for all planets, including ones not involved in  collision.
                impactedBodies.append(int(max_idx[0][0]));
                impactTimes.append(collisionTime);
                impactVelocity.append(relativeVelocity);

                bMass.append(b.m); bEcc.append(b.e); bX.append(b.x); bY.append(b.y); bZ.append(b.z); bAU.append(b.a);
                cMass.append(c.m); cEcc.append(c.e); cX.append(c.x); cY.append(c.y); cZ.append(c.z); cAU.append(c.a);
                dMass.append(d.m); dEcc.append(d.e); dX.append(d.x); dY.append(d.y); dZ.append(d.z); dAU.append(d.a);
                eMass.append(e.m); eEcc.append(e.e); eX.append(e.x); eY.append(e.y); eZ.append(e.z); eAU.append(e.a);
                fMass.append(f.m); fEcc.append(f.e); fX.append(f.x); fY.append(f.y); fZ.append(f.z); fAU.append(f.a);
                gMass.append(g.m); gEcc.append(g.e); gX.append(g.x); gY.append(g.y); gZ.append(g.z); gAU.append(g.a);
                hMass.append(h.m); hEcc.append(h.e); hX.append(h.x); hY.append(h.y); hZ.append(h.z); hAU.append(h.a);

                # *** BELOW IS CODE FOR CUSTOM COLLISION MERGER ***

                cp1 = sim.particles[int(max_idx[0][0])] # Points to first of the two collided particles (planet)
                cp2 = sim.particles[int(max_idx[0][1])] # Points to second of the two collided particles (planetesimal)

                # Merge by conserving mass, volume, and momentum
                invMass = 1.0/(cp1.m+cp2.m) #Inverse sum mass of two collided particles
```

```
                #Conserve momentum
                cp1.vx = (cp1.vx*cp1.m + cp2.vx*cp2.m)*invMass
                cp1.vy = (cp1.vy*cp1.m + cp2.vy*cp2.m)*invMass
                cp1.vz = (cp1.vz*cp1.m + cp2.vz*cp2.m)*invMass

                cp1.x = (cp1.x*cp1.m + cp2.x*cp2.m)*invMass # ??
                cp1.y = (cp1.y*cp1.m + cp2.y*cp2.m)*invMass # ??
                cp1.z = (cp1.z*cp1.m + cp2.z*cp2.m)*invMass # ??

                #Conserve mass
                cp1.m = cp1.m + cp2.m

                #Conserve volume
                cp1.r = np.cbrt(cp1.r*cp1.r*cp1.r + cp2.r*cp2.r*cp2.r)

                #Remove particle with higher index
                    # This will be the planetesimal, therefore keeping planet hash values throughout integration
                sim.remove(index = int(max_idx[0][1]))

                sim.N_active = 8

# Create data frame of all stored values extracted at each collision
collisiondf = pd.DataFrame({'Impacted Body':impactedBodies, 'Collision Time (year)':impactTimes,
                            'Impact Velocity (km/s)':impactVelocity,
                            '1b Mass':bMass, '1c Mass':cMass, '1d Mass':dMass, '1e Mass':eMass, '1f Mass':fMass,
                            '1g Mass':gMass, '1h Mass':hMass, '1b Ecc':bEcc, '1c Ecc':cEcc, '1d Ecc':dEcc,
                            '1e Ecc':eEcc, '1f Ecc':fEcc, '1g Ecc':gEcc, '1h Ecc':hEcc, '1b x pos':bX,
                            '1c x pos':cX, '1d x pos':dX, '1e x pos':eX, '1f x pos':fX, '1g x pos':gX,
                            '1h x pos':hX, '1b y pos':bY, '1c y pos':cY, '1d y pos':dY, '1e y pos':eY,
                            '1f y pos':fY, '1g y pos':gY, '1h y pos':hY, '1b z pos':bZ, '1c z pos':cZ,
                            '1d z pos':dZ, '1e z pos':eZ, '1f z pos':fZ, '1g z pos':gZ, '1h z pos':hZ,
                            '1b AU':bAU,'1c AU':cAU,'1d AU':dAU,'1e AU':eAU,'1f AU':fAU,'1g AU':gAU,'1h AU':hAU})

# GET END SET-UP DATA!
xyEnd = np.zeros((N_pl+7, 6)); # First value = number of timesteps, Third value = # of orbital parameters
semimajorAxisEnd = []; eccentricityEnd = []; massEnd = [];
xEnd = []; yEnd = [];zEnd = [];
for j,p in enumerate(sim.particles[1:]):
        xyEnd[j] = [p.m,p.a,p.e,p.x,p.y,p.z]
        semimajorAxisEnd.append(p.a)
        eccentricityEnd.append(p.e)
        massEnd.append(p.m)
        xEnd.append(p.x)
        yEnd.append(p.y)
        zEnd.append(p.z)

# Create end-system dataframe (enddf)
enddf = pd.DataFrame({'mass @ start':massEnd, 'ecc @ start':eccentricityEnd,
                      'semimajor axis @ start':semimajorAxisEnd, 'x pos @ start':xEnd,
                      'y pos @ start':yEnd, 'z pos @ start':zEnd})

# Initialize orbital parameters needed to track resonance arguments
bLongitudes = []; cLongitudes = []; dLongitudes = []; eLongitudes = []; fLongitudes = []; gLongitudes = [];
hLongitudes = []; bOmegas = []; cOmegas = []; dOmegas = []; eOmegas = []; fOmegas = []; gOmegas = []; hOmegas = [];
bomegas = []; comegas = []; domegas = []; eomegas = []; fomegas = []; gomegas = []; homegas = []; bM = []; cM = [];
dM = []; eM = []; fM = []; gM = []; hM = [];

# Populate arrays with data stored from integration:
for i in range(len(xy)):
    bLongitudes.append(xy[i][0][9]); cLongitudes.append(xy[i][1][9]); dLongitudes.append(xy[i][2][9]);
    eLongitudes.append(xy[i][3][9]); fLongitudes.append(xy[i][4][9]); gLongitudes.append(xy[i][5][9]);
    hLongitudes.append(xy[i][6][9]);

    bOmegas.append(xy[i][0][10]); cOmegas.append(xy[i][1][10]); dOmegas.append(xy[i][2][10]);
    eOmegas.append(xy[i][3][10]); fOmegas.append(xy[i][4][10]); gOmegas.append(xy[i][5][10]);
    hOmegas.append(xy[i][6][10]);

    bomegas.append(xy[i][0][11]); comegas.append(xy[i][1][11]); domegas.append(xy[i][2][11]);
    eomegas.append(xy[i][3][11]); fomegas.append(xy[i][4][11]); gomegas.append(xy[i][5][11]);
    homegas.append(xy[i][6][11]);

    bM.append(xy[i][0][12]); cM.append(xy[i][1][12]); dM.append(xy[i][2][12]); eM.append(xy[i][3][12]);
    fM.append(xy[i][4][12]); gM.append(xy[i][5][12]); hM.append(xy[i][6][12]);

# Initialize arrays for 2-body resonance arguments (1 and 2)
theta1BC = []; theta1CD = []; theta1DE = []; theta1EF = []; theta1FG = []; theta1GH = [];
theta2BC = []; theta2CD = []; theta2DE = []; theta2EF = []; theta2FG = []; theta2GH = [];

# Initialize arrays for 3-body resonance arguments
thetaBCD = []; thetaCDE = []; thetaDEF = []; thetaEFG = []; thetaFGH = [];

# Calculate resonance arguments and store:
```

```
for j in range(len(bLongitudes)):
    theta1BC.append(min180To180(8*cLongitudes[j]-5*bLongitudes[j]-3*(bOmegas[j]+bomegas[j])));
    theta1CD.append(min180To180(5*dLongitudes[j]-3*cLongitudes[j]-2*(cOmegas[j]+comegas[j])));
    theta1DE.append(min180To180(3*eLongitudes[j]-2*dLongitudes[j]-1*(dOmegas[j]+domegas[j])));
    theta1EF.append(min180To180(3*fLongitudes[j]-2*eLongitudes[j]-1*(eOmegas[j]+eomegas[j])));
    theta1FG.append(min180To180(4*gLongitudes[j]-3*fLongitudes[j]-1*(fOmegas[j]+fomegas[j])));
    theta1GH.append(min180To180(3*hLongitudes[j]-2*gLongitudes[j]-1*(gOmegas[j]+gomegas[j])));

    theta2BC.append(min180To180(8*cLongitudes[j]-5*bLongitudes[j]-3*(cOmegas[j]+comegas[j])));
    theta2CD.append(min180To180(5*dLongitudes[j]-3*cLongitudes[j]-2*(dOmegas[j]+domegas[j])));
    theta2DE.append(min180To180(3*eLongitudes[j]-2*dLongitudes[j]-1*(eOmegas[j]+eomegas[j])));
    theta2EF.append(min180To180(3*fLongitudes[j]-2*eLongitudes[j]-1*(fOmegas[j]+fomegas[j])));
    theta2FG.append(min180To180(4*gLongitudes[j]-3*fLongitudes[j]-1*(gOmegas[j]+gomegas[j])));
    theta2GH.append(min180To180(3*hLongitudes[j]-2*gLongitudes[j]-1*(hOmegas[j]+homegas[j])));

    thetaBCD.append(min180To180(2*bLongitudes[j]-5*cLongitudes[j]+3*dLongitudes[j]));
    thetaCDE.append(min180To180(cLongitudes[j]-3*dLongitudes[j]+2*eLongitudes[j]));
    thetaDEF.append(min180To180(2*dLongitudes[j]-5*eLongitudes[j]+3*fLongitudes[j]));
    thetaEFG.append(min180To180(eLongitudes[j]-3*fLongitudes[j]+2*gLongitudes[j]));
    thetaFGH.append(min180To180(fLongitudes[j]-2*gLongitudes[j]+hLongitudes[j]));

# Create resonance dataframe (resonantdf)
resonantdf = pd.DataFrame({'times (years)':times, 'bLongitudes':bLongitudes, 'cLongitudes':cLongitudes, 'dLongitudes':dLongi
                           'eLongitudes':eLongitudes, 'fLongitudes':fLongitudes, 'gLongitudes':gLongitudes,
                           'hLongitudes':hLongitudes, 'bOmegas':bOmegas, 'cOmegas':cOmegas, 'dOmegas':dOmegas,
                           'eOmegas':eOmegas, 'fOmegas':fOmegas, 'gOmegas': gOmegas, 'hOmegas':hOmegas,
                           'bomegas':bomegas, 'comegas':comegas, 'domegas':domegas, 'eomegas':eomegas,
                           'fomegas':fomegas, 'gomegas':gomegas, 'homegas':homegas, 'b M':bM, 'c M':cM,
                           'd M':dM, 'e M':eM, 'f M':fM, 'g M':gM, 'h M':hM, 'theta1 b:c':theta1BC,
                           'theta1 c:d':theta1CD, 'theta1 d:e':theta1DE, 'theta1 e:f':theta1EF, 'theta1 f:g':theta1FG,
                           'theta1 g:h':theta1GH, 'theta2 b:c':theta2BC, 'theta2 c:d':theta2CD, 'theta2 d:e':theta2DE,
                           'theta2 e:f':theta2EF, 'theta2 f:g':theta2FG, 'theta2 g:h':theta2GH, 'theta bcd':thetaBCD,
                           'theta cde':thetaCDE, 'theta def':thetaDEF, 'theta efg':thetaEFG, 'theta fgh':thetaFGH})

# Calculate final energy of the system. Used to track energy change at end. If energy change is large, likely means
# the integrator is not resolving something correctly.
EF = sim.calculate_energy()

# Print initial, final, and delta energy
print(E0, EF, E0-EF)

# IF YOU WANT TO SAVE SIMULATION ARCHIVE TO CONTINUE INTEGRATION LATER, TOGGLE THIS ON:
# sim.simulationarchive_snapshot("NarrowBelt_2xPlMass_50k.bin", deletefile=True)

## REMEMBER TO TAKE DOWN WALL TIME, AND BELT ARCHITECTURE (what parameters for a, e, i used to assign pls)
```

OUTPUT DATA FRAMES TO EXCEL SPREADSHEET. STARTDF (START SYSTEM), ENDDF (END SYSTEM), COLLISIONDF (COLLISIONAL DATA), AND RESONANTDF (RESONANCE DATA) WILL EACH HAVE THEIR OWN TAB IN SPREADSHEET:

In [ ]:

```
## UPDATE FILE NAME OR YOU WILL OVERRIDE PREVIOUS DATA!!
with pd.ExcelWriter('/Users/sebastianperezlopez/Desktop/Lissauer Codes/Runs/NarrowBelt_2xPlMass_50k.xlsx') as writer:
    # First tab in excel file will be starting architecture
    startdf.to_excel(writer, sheet_name='Start Set-up')
    # Second tab in excel file will be ending architecture
    enddf.to_excel(writer, sheet_name='End Set-up')
    # Third tab in excel will be collision data // data obtained throughout the simulation at each impact
    collisiondf.to_excel(writer, sheet_name='Collision data')
    # Fourth tab in excel will be resonance data
    resonantdf.to_excel(writer, sheet_name='Resonant Arguments')
```

OUTPUT TIME FRAME DATA TO A TEXT FILE. ALL ORBITAL PARAMETERS CHOSEN TO BE EXTRACTED FOR EVERY BODY AT EVERY TIMSTEP. WILL OUTPUT TO A .TXT FILE THAT CAN BE CONVERTED TO EXCEL TO TRACK SYSTEM EVOLUTION:

In [ ]:

```
f = open("/Users/sebastianperezlopez/Desktop/Lissauer Codes/Runs/TimeframeData_NarrowBelt_2xPlMass_50k.txt", "a")
i = -1
for timestep in xy:
    i += 1
    j = -1
    for planetesimal in timestep:
        print(times[i], str(j+1), str(planetesimal[0]), str(planetesimal[1]), str(planetesimal[2]),
                str(planetesimal[3]), str(planetesimal[4]), str(planetesimal[5]), str(planetesimal[6]),
                str(planetesimal[7]), str(planetesimal[8]), str(planetesimal[9]), str(planetesimal[10]),
                str(planetesimal[11]), str(planetesimal[12]), str(planetesimal[13]), file=f, end = '\n')
        j += 1
f.close()
```

IF RE-UPLOADING A PREVIOUS SIMULATION ARCHIVE TO START RUNNING AGAIN, USE THIS CELL. DO NOT RUN THIS MORE THAN TWICE — IT WILL KEEP APPENDING AND GO UNSTABLE. CODE WON'T RUNA ND YOU'LL HAVE TO RE-RUN THE FIRST PART OF THE SIMULATION.

RECOMMENDED TO MAKE COPY OF SIMULATION ARCHIVE BEFORE RE-UPLOADING

In [ ]:

```
def setupSimulation2ndHalf():

    sim2ndHalf = rebound.Simulation("IC0K1.2521e+02mag1.3982e-01.bin") # input archive file path/name here

    # PICK INTEGRATOR HERE:
        # Standard: mercurius
    sim2ndHalf.integrator = "mercurius"

    # PICK MERCURIUS TIMESTEP HERE:
        # Standard: sim.dt = 0.0000082192
    sim2ndHalf.dt = 0.0000082192

    return sim2ndHalf
```