```
"""
N_Bodies Gravitational Collapse simulation by Laurence de Bruxelles
"""

import numpy
import matplotlib.pyplot as pyplot

def coroutine(func):
    """coroutine decorator"""
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        cr.next()
        return cr
    return start

@coroutine
def data_buffer(target,bufsize):
    """
    Caches values in a fixed length buffer
    until it is full or None is sent,
    whereupon it sends on
    """
    try:
        buf = bufsize*[None]
        while 1:
            for i in xrange(bufsize):
                buf[i] = (yield)
                if buf[i] == None:
                    target.send(buf[:i])
                    break
            else:
                target.send(buf)
    except GeneratorExit:
        target.close()

@coroutine
def dedup(target,**tol):
    """
    Skip values in a sequence
    that are too similar. Values
    can be tuples, in which case
    all elements of the tuple have
    to be similar, a la numpy.allclose().
    """
    import numpy

    old = numpy.NaN
    while 1:
        new = (yield)
        if new == None:
            target.send(None)
            continue
        if numpy.allclose(new,old,**tol):
            continue
        target.send(new)
        old = new

def read_file(paths):
    """
    Read and process files that contain
    the initial variables for an N bodies simulation.

    Reads the masses, positions, velocities,
    name and plotting style,
    in the order specified by cols.

    Takes a list of paths
    Returns m a numpy array
            x a numpy array
            v a numpy array
            name a list
            style a list
    """
    cols = ('m','x','v','name','style')

    # read lines into a dictionary as per cols
    files = (open(path) for path in paths)
    lines = [line.split() for f in files for line in f]
    data = (dict(zip(cols,line)) for line in lines)

    def field_map(dictseq,name,func):
        # From dabeaz.com/generators-uk/
        for d in dictseq:
            d[name] = func(d[name])
            yield d
    read_vector = lambda l: map(float,l.split(','))

    # do some data conversions
    data = field_map(data,'m',float)
    data = field_map(data,'x',read_vector)
    data = field_map(data,'v',read_vector)

    # append to lists since
    # numpy.append is a Bad Idea
    out = [[] for _ in range(len(cols))]
    out = dict(zip(cols,out))
    for d in data:
        for k in d:
            out[k].append(d[k])

    out['m'] = numpy.array(out['m'])
    out['x'] = numpy.array(out['x'])
    out['v'] = numpy.array(out['v'])

    # return lists in order of
    # the column specification
    return [out[k] for k in cols]

class Energy:
    def __init__(self):
        self.first = True

    def __call__(self,n):
        """
        Calculate the energy of the system,
        and return the percent change in the
        total energy
        """
        self.ke = self.kinetic_energy(n)
        self.pe = self.potential_energy(n)
        self.total = self.ke + self.pe
        self.virial = 2*self.ke + self.pe

        # save the initial energies
        if self.first:
            self.initial_total = self.total
            self.initial_ke = self.ke
            self.initial_pe = self.pe
            self.first = False

        self.percent_total = 100*(self.total/self.initial_total - 1)
        self.percent_ke = 100*(self.ke/self.initial_ke - 1)
        self.percent_pe = 100*(self.pe/self.initial_pe - 1)

        return self.percent_total

    def kinetic_energy(self,n):
        """
        Find the total kinetic energy of an N bodies system.

        Takes an N_bodies instance.
        """
        # align pe and ke by moving velocity a half-step
        v = n.v - 0.5*n.a*n.dt
        ke = n.m[:,numpy.newaxis]*numpy.square(v)
        return numpy.sum(ke)

    def potential_energy(self,n):
        """
        Find the total gravitational potential energy
        of an N bodies system.

        Takes an N_bodies instance
        """
        # Potential U_i=sum(i!=j,-GM_j/|r_ij|)

        U = -n.G*n.m*n.m[:,numpy.newaxis]/n.r_norm
        U[range(n.n),range(n.n)] = 0 # get rid of self-potentials

        return numpy.sum(U)

class Plotter:
    """
    A coroutine like layer above pyplot.
    Draws a graph as and when data arrives,
    so it will be animated
    """
    def __init__(self,plots,axis,**pipeargs):
        """
        Create the main plot. Each line required
        must be declared upfront here, in the form
        of a dictionary, plots. plots.keys() are the
        labels for each line, and plots.values() are
        the style of the line, in the matplot
        abbreviated form. The range of data values
        must also be declared in a tuple axis, where
        axis = [xmin,xmax,ymin,ymax], as in pyplot.

        There are also various options
        controlling how the data is handled.
        """

        pyplot.ion() # enable animations

        self.fig = pyplot.figure()
        self.ax = self.fig.add_subplot(1,1,1) # fill figure

        self.lines = self.ax.plot(*self.read_args(plots),markevery=(-1,1))
        self.lines = self.add_pipeline(self.lines,**pipeargs)
        self.lines = dict(zip(plots.keys(),self.lines))

        self.ax.axis(axis)
        self.ax.set_aspect('equal') # ensure circles appear circular

        pyplot.draw()

        # needed for later
        self.count = 0
        self.bufsize = pipeargs['bufsize']

    def add_subplot(self,subplots,axis,position,**pipeargs):
        """
        Add a floating subplot to the pyplot figure.
        The function call form is similar to that of
        the Plotter constructor, with the addition of
        the list position, which defines the position
        and size of the subplot, in units from 0 to 1,
        such that position = [bottom,left,width,height]
        """
        subax = self.fig.add_axes(position)

        sublines = subax.plot(*self.read_args(subplots))

        sublines = self.add_pipeline(sublines,**pipeargs)
        sublines = zip(subplots.keys(),sublines)
        self.lines.update(sublines)
        subax.axis(axis)

        return subax

    def send(self,d):
        """
        This adds data to the lines in the plot.
        Data should be passed in the form of a
        dictionary, with values having keys
        corresponding to the keys defined at
        instantiation. The name of this method
        is send, to allow it to be used as a
        consumer in a coroutine pipeline
        """
        for key,value in d.iteritems():
            self.lines[key].send(value)

        # redraw after bufsize calls
        self.count += 1
        if self.count == 100:
            self.flush()
            self.count = 0
            pyplot.draw()


    def flush(self):
        """
        Add any data currently residing
        in pipelines to plot lines
        """
        for line in self.lines.values():
                line.send(None)

    @staticmethod
    def add_pipeline(lines,maxpoints=None,bufsize=100,**tol):
        """
        Helper method to add a reciever and a pipeline
        for each plot line. The pipeline created depends
        on the options given
        """
        # this actually adds the data to a line
        lines = [Plotter.line_consumer(l,maxpoints) for l in lines]
        if bufsize: # if we want buffered data
            lines = [data_buffer(l,bufsize) for l in lines]
        if tol: # if we want to discard similar values
            lines = [dedup(l,**tol) for l in lines]

        return lines

    @staticmethod
    @coroutine
    def line_consumer(line,maxpoints=None):
        """
        This helper coroutine does the actual
        work of adding data to the pyplot
        Line2D object for each line; one per
        line is created in __init__ and
        associated with the line labels
        """
        while 1:
            data = line.get_xydata()
            new = (yield)
            if len(new) == 0: continue
            if maxpoints:
                lim = maxpoints - len(new)
                data = data[-lim:]
            new = numpy.append(data,new,0)
            line.set_data(new[:,0],new[:,1])

    @staticmethod
    def read_args(args):
        """
        Helper function to read the line specification
        dictionary into a form useable by pyplot.plot()
        """
        args = [(None,None,style) for (name,style) in args.iteritems()]
        args = [item for tup in args for item in tup] # flatten
        return args

class N_bodies:
    """
    Simulates the time evolution of a system
    of N particles which interact with each
    other through gravitational forces.

    After creating a N_bodies simulation,
    step forward using N_bodies.leapfrog()
    """
    G = 6.67384e-11 # gravitational constant

    def __init__(self,mass,position,velocity,dt,softening):
        """
        Create the simulation, giving initial values
        of particle masses, positions and velocities.

        Masses can be a 1D array or a scalar; if a scalar
        all particles will have the same mass. Position
        and velocity must be numpy arrays of the same shape;
        normally the last axis will have length 3, to give a
        list of vectors. Each particle is defined by the same
        index across all three arrays.

        dt is the time step in seconds between each simulation
        step.
        softening is a constant that reduces the gravitational
        force for particles close to each other.
        """
        self.m = numpy.array(mass)
        self.x = numpy.array(position)
        self.v = numpy.array(velocity)
        self.dt = dt
        self.s = softening

        # make some definitions up front to allocate
        # memory and to have a clear definition of
        # expected array shapes
        self.n = self.x.shape[0] # since m may be a scalar
        self.r = numpy.empty((self.n,self.n,3))
        self.r_norm = numpy.empty((self.n,self.n))
        self.F = numpy.empty((self.n,self.n,3))
        self.a = numpy.empty((self.n,self.n,3))

        # current run time in seconds
        self.t = 0

    def find_r(self):
        """
        Creates a matrix with ij elements x_i - x_j;
        ie a matrix of displacement vectors between
        particles i and j
        """
        # x[:,numpy.newaxis,:] transposes and extends x;
        # put another way x.shape is changed from
        # (n,3) to (n,newaxis,3), and then the subtraction
        # of x broadcasts newaxis to have size n
        self.r = self.x[:,numpy.newaxis,:]-self.x
        # magnitude of each displacement vector
        self.r_norm = numpy.apply_along_axis(numpy.linalg.norm,2,self.r)
        return self.r,self.r_norm

    def find_a(self):
        """
        Creates an array of acceleration vectors for each particle
        """
        # simple Newtonian gravitational force
        self.F = -self.G*self.r
        self.F *= (self.m/(self.r_norm**2 + self.s**2)**1.5)[:,:,numpy.newaxis]
        self.F = numpy.nan_to_num(self.F) # div by zero gives NaN
        # F not true force since does not include m_i
        # sum 'forces' to get acceleration
        self.a = numpy.apply_along_axis(numpy.sum,1,self.F)

        return self.a

    def update_v(self):
        """
        Update the velocities using the accelerations found
        """
        self.v += self.a*self.dt
        return self.v

    def update_x(self):
        """
        Update the positions using the velocities found
        """
        self.x += self.v*self.dt
        return self.x

    def v_correction(self):
        """
        Helper function to get the positions and velocities
        1/2 dt out of time with each other. The leapfrog
        algorithm requires this
        """
        self.find_r()
        self.find_a()
        self.v -= self.a*self.dt/2.0

        return self.v

    def leapfrog(self):
        """
        Use the leapfrog algorith to step the state of the
        simulation dt seconds forward in time. This is the
        primary function of the N_bodies class
        """
        self.find_r()
        self.find_a()
        self.update_v()
        self.update_x()

        self.t += self.dt

        return self.x,self.v
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Run an N body simulation")

    parser.add_argument("file",nargs="+",
                        help="files containing the mass,"
                        " initial position and velocity of the bodies")
    args = parser.parse_args()

    day = 86400 # day in seconds
    year = 365 # year in days
    au = 149.6e9 # astronomical unit of distance

    dt = 10*day # 10 day in seconds
    t = 10*year # 100 years in units of dt

    bufsize=1000

    m,x,v,names,styles = read_file(args.file)
    # change the position and velocity of the Sun so that
    # the centre of mass is stationary and at the origin
    x[0] = -numpy.sum(x[1:]*m[1:,numpy.newaxis],axis=0)/m[0]
    v[0] = -numpy.sum(v[1:]*m[1:,numpy.newaxis],axis=0)/m[0]

    # instantiate the simulation
    solar_system = N_bodies(m,x,v,dt,softening=0)
    v = solar_system.v_correction() # move velocities out of phase

    # setup orbit plot
    plot = Plotter(dict(zip(names,styles)),axis=[-9e11,9e11,-9e11,9e11],
            bufsize=bufsize,rtol=0,atol=1e7)

    # instantiate an energy follower
    energy = Energy()
    # setup energy plot
    ax = plot.add_subplot({'Energy':'r','KE':'b','PE':'g'},bufsize=bufsize,
        position=[0.3,0.75,0.6,0.2],axis=[0,t*dt/day/year,-7.5e35,4e35])
    ax.set_xticklabels([])
    ax.text(0.4*t*dt/day/year,-2e35,'Total Energy',
        verticalalignment='center')
    ax.text(0.1*t*dt/day/year,1.9e35,'Kinectic Energy',
        verticalalignment='center')
    ax.text(0.6*t*dt/day/year,-6e35,'Potential Energy')
    ax.set_ylabel('Energy / J',rotation='horizontal')

    # setup Earth orbital radius plot
    ax = plot.add_subplot({'R':'r'},bufsize=bufsize,
        position=[0.3,0.7,0.6,0.05],axis=[0,t*dt/day/year,0.9945,0.9985])
    ax.set_yticks([0.995,0.998])
    ax.set_yticklabels([0.995,0.998])
    ax.set_xlabel('Time / yr',labelpad=-10)
    ax.set_ylabel('R / AU',rotation='horizontal')

    # draw a rectangle around subplots
    import matplotlib.patches as mpatches
    rect = mpatches.Rectangle((-0.1,0.7),1.3,0.41,clip_on=False,zorder=1000,
        facecolor='white',edgecolor='black',transform=plot.ax.transAxes)
    plot.ax.add_patch(rect)

    # main event loop
    for n in xrange(t):
        x,v = solar_system.leapfrog() # do one simulation step
        energy(solar_system)

        time_in_yr = n*float(dt)/float(day)/float(year)

        plot.send(dict(zip(names,x[:,0:2].tolist()))) # plot positions
        plot.send({
            'Energy':(time_in_yr,energy.total),
            'KE':(time_in_yr,energy.ke),
            'PE':(time_in_yr,energy.pe),
            'R':(time_in_yr,solar_system.r_norm[1,0]/au),
        }) # plot energy and radius

        # print solar_system.r_norm[1,0]/au

print 'Total Energy %% Change: %+.3e' % energy.percent_total
pyplot.show()
```