First Abstractions

In the first implementation of the dynamics we can see some repetition and inconvenience in the code, primarily in the definition of the rod properties and simulation parameters. Here we will work on making this a slightly more friendly interface to work with while still focusing on the same system as before.

First we can abstract the definition of the rod properties. In this case we still only look at a uniform cylindrical body for ease. We can define the `Rod` class to store this information.

```
class Rod():
    """
    Rod stores the material properties and geometric properties of the cylindrical rod
    Need to specify:
        D: diameter
        L: length
        E: Young's Modulus
        rho: density
    """
    def __init__(self,D,L,E,rho):
        #setup properties
        A = np.pi/4*D**2
        I = np.pi/64*D**4
        J = 2*I
        G = E/3 # assuming incompressible material

        # store values important to simulation
        self.K = np.diag([E*I,E*I,G*J,G*A,G*A,E*A])
        self.M = rho*np.diag([I,I,J,A,A,A])
        self.L= L
        self.xi_ref = np.array([0,0,0,0,0,1])
```

Now this specifies the basics of what we need to know about the rod, but during the simulation process it'd be nice to be storing more information. We can initialize the rod here are store the state for every step.

```
class Rod():
    """
    Rod stores the material properties and geometric properties of the cylindrical rod
```

```
    Need to specify:
        D: diameter
        L: length
        E: Young's Modulus
        rho: density
        N: number of discretizations
        xi_init: function of s that specifies the initial value of xi (defaults to straight)
        eta_init: function os s that specifies the initial value of eta (defaults to station
    """

    def __init__(self, D, L, E, rho, N, xi_init=lambda s: np.array([0, 0, 0, 0, 0, 1]),
                 eta_init=lambda s: np.array([0, 0, 0, 0, 0, 0])):
        # setup properties
        A = np.pi / 4 * D ** 2
        I = np.pi / 64 * D ** 4
        J = 2 * I
        G = E / 3  # assuming incompressible material

        # store values important to simulation
        self.K = np.diag([E * I, E * I, G * J, G * A, G * A, E * A])
        self.M = rho * np.diag([I, I, J, A, A, A])
        self.L = L
        self.xi_ref = np.array([0, 0, 0, 0, 0, 1])
        self.ds = L / (N - 1)
        self.N = N
        self.xi_init = xi_init
        self.eta_init = eta_init

        # initialize state
        self.g = None
        self.xi = None
        self.eta = None
        self.initRod()

    def initRod(self):
        # setup g, xi, and eta for the initial configuration
        g = np.zeros((self.N, 12))
        xi = np.zeros((self.N, 6))
        eta = np.zeros((self.N, 6))

        # set xi and eta
        for i in range(self.N):
            s = self.ds * i
            xi[i, :] = self.xi_init(s)
            eta[i, :] = self.eta_init(s)
```

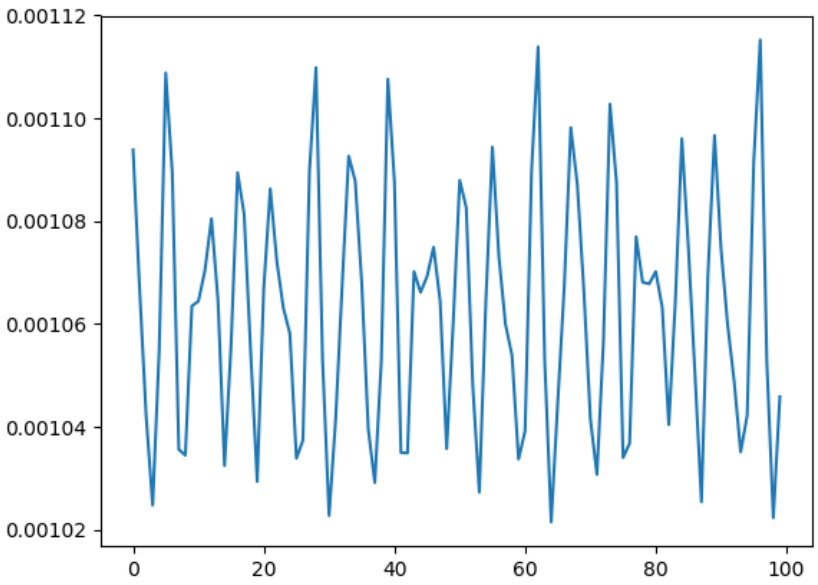
```
        # integrate G
        G = np.eye(4)
        g[0, :] = flatten(G)
        for i in range(1, self.N):
            G = G @ expm(se(self.ds * xi[i - 1, :]))
            g[i, :] = flatten(G)

        # set state
        self.g = g
        self.xi = xi
        self.eta = eta

    def plot(self, ax=None):
        # not sure if this is the best way, but if an axis isn't specified generate it, if
        if ax is None:
            fig, ax = plt.subplot(111, projection='3d')
        ax.plot(self.g[9,:], self.g[10,:], self.g[11,:])
        return ax

    def energy(self):
        H = 0 #total energy (aka Hamiltonian)
        for i in range(self.N):
            T = self.eta[i,:].T @ self.M @ self.eta[i,:]
            U = (self.xi[i,:]-self.xi_ref).T @ self.K @ (self.xi[i,:]-self.xi_ref)
            H += 1/2*(T + U)
        return self.ds*H
```

This should provide all the information for setting up and starting the simulation as well as convenience functions for plotting and computing the energy. Now we need to include the stepping. Stepping should work exactly the same except now we are storing the state in the object rather than passing it around. This also means we can potentially eliminate an extra integration step because the integration after solving for `xi0` is the same as the last step in the solving process, so it is redundant.

```
class Rod():
    """
    Removed for clarity
    """
    def step(self, dt):
        # since we are modifying the state want to keep track of the previous state for the
        prev = copy.deepcopy(self)
        # just need to solve for xi0 and the state should be updated
        xi0 = fsolve(lambda x: self.condition(prev, dt, x), self.xi[0, :])

    def condition(self, prev, dt, xi0):
        # integrate and see if the tip condition is satisfied
```

```
        self.integrate(prev, dt, xi0)
        return self.xi[-1, :] - self.xi_ref

    def integrate(self, prev, dt, xi0):
        self.xi[0, :] = xi0

        # integration over the body (don't need the initial point as the initial values are
        for i in range(self.N - 1):
            # averaging over steps to get half step values
            xi_half = (self.xi[i, :] + prev.xi[i, :]) / 2
            eta_half = (self.eta[i, :] + prev.eta[i, :]) / 2

            # implicit midpoint approximation
            xi_dot = (self.xi[i, :] - prev.xi[i, :]) / dt
            eta_dot = (self.eta[i, :] - prev.eta[i, :]) / dt

            # spatial derivatives
            xi_der = np.linalg.inv(self.K) @ (
                    (self.M @ eta_dot) - (adjoint(eta_half).T @ self.M @ eta_half) + (
                        adjoint(xi_half).T @ self.K @ (xi_half - self.xi_ref)))
            eta_der = xi_dot - (adjoint(xi_half) @ eta_half)

            # explicit Euler step
            xi_half_next = xi_half + self.ds * xi_der
            eta_half_next = eta_half + self.ds * eta_der

            # determine next step from half step value
            self.xi[i + 1, :] = 2 * xi_half_next - prev.xi[i + 1, :]
            self.eta[i + 1, :] = 2 * eta_half_next - prev.eta[i + 1, :]

        # midpoint RKMK to step the g values
        for i in range(self.N):
            self.g[i, :] = flatten(unflatten(prev.g[i, :]) @ expm(se(dt * (self.eta[i, :] +
```

This should be the final state of the simulation code. Now it is a bit more convenient to use and there is less duplication and hardcoding of properties. To make sure it is all implemented properly lets try the energy again.



Which looks the same as before so it should be the same implementation.

So now the code is more organized and useful to use. Adding in a couple modifications: moving utility functions to their own file and marking some functions as private (python doesn't actually have private functions, but '_' before the name implies that). The final code is in `conservative` and `utils` and to test it run `conservative`.