Generalizing the Loads

For working with the rods the main determinant of the behavior are the present external loads, but having to define and redefine them over and over could get tedious especially when they get mixed and matched. This leads to wanting to generalize the loads a bit so they can be easily swapped in and out for the simulations. The loads we have considered so far are gravity, viscosity, and cable actuators and they influence the tip load for the boundary condition and generate $\bar{A}$ and $\bar{B}$ for the distributed load. So, a load should implement how to compute their distributed load and their tip load and any internal information they need. I'm going to make an execpetion for viscosity as it is a material property so I'm leaving it in the rod definition. The resulting abstract class for loads is:

```
from abc import ABCMeta, abstractmethod

class Load(metaclass=ABCMeta):
    """
    The general class for dealing with loads
    need to implement distributed load that gives both A_bar and B_bar
    need to implement tip load
    takes current g, xi, eta, rod properties, and inputs
    """

    @abstractmethod
    def dist_load(self, g, xi, eta, xi_dot, eta_dot, rod, q):
        return 0, 0 # if no distributed load

    @abstractmethod
    def tip_load(self, g, xi, eta, xi_dot, eta_dot, rod, q):
        return 0 # no tip load
```

Where the methods default to returning 0 to indicate no influence. So, to implement the defintions for gravity and cable loads we have:

```
class Gravity(Load):
    """
    Gravity load
    """
```

```
    def __init__(self, grav):
        self.grav = grav  # acceleration vector

    def dist_load(self, g, xi, eta, xi_dot, eta_dot, rod, q):
        R = g[:3, :3]
        A_bar = 0
        B_bar = rod.rho * rod.A * np.concatenate([np.array([0, 0, 0]), R.T @ self.grav])

        return (A_bar, B_bar)

    def tip_load(self, g, xi, eta, xi_dot, eta_dot, rod, q):
        return 0


class Cables(Load):
    """
    Cables load
    """

    def __init__(self, r, N):
        self.r = r  # displacements
        self.N = N  # number of actuators

    def dist_load(self, g, xi, eta, xi_dot, eta_dot, rod, q):
        omega = xi[:3]
        nu = xi[3:]
        R = g[:3, :3]
        A_bar = 0
        B_bar = 0
        for i in range(self.N):
            r_i = self.r(i)
            pa_der = R @ (nu - skew(r_i) @ omega)
            P = R.T @ -skew(pa_der) * skew(pa_der) / np.linalg.norm(pa_der) ** 3 @ R
            b = P @ skew(omega) @ (nu - skew(r_i) @ omega)
            B_bar += q[j] * np.concatenate([skew(r_i) @ b, b])
            A_bar += q[j] * np.concatenate([np.concatenate([-skew(r_i) @ P @ skew(r_i), skew
                                            np.concatenate([-P @ skew(r_i), P], 1)])

        return A_bar, B_bar

    def tip_load(self, g, xi, eta, xi_dot, eta_dot, rod, q):
        W = 0
        z = np.array([0, 0, 1])
        for i in range(self.N):
            W += q[i] * np.concatenate([-skew(self.r(i)) @ z, z])
```

```
        return W
```

Now to implement the changes to the rod we just include a list of the loads we want to use in the simulation and iterate over them in the appropriate spots.

```
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
        mu: shear viscosity
        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
        loads: list of Loads to use in the simulation
    """

    def __init__(self, D, L, E, rho, mu, 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]), loads=[]):
        # 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.V = mu * np.diag([3 * I, 3 * I, J, A, A, 3 * A])
        self.L = L
        self.D = D
        self.rho = rho
        self.A = A
        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()
```

```
        # the different kinds of loads, viscosity always assumed to occur
        self.loads = loads

    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

    def step(self, dt, q):
        # 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, q), self.xi[0, :])

    def _condition(self, prev, dt, xi0, q):
        # integrate and see if the tip condition is satisfied
        self._integrate(prev, dt, xi0, q)

        # all tip loads
        W = 0
        # data
        g = unflatten(self.g[-1,:])
        xi = self.xi[-1,:]
        eta = self.xi[-1,:]
        xi_dot = (self.xi[-1,:] - prev.xi[-1,:])/dt
        eta_dot = (self.eta[-1,:] - prev.eta[-1,:])/dt
        for load in self.loads:
            W += load.tip_load(g, xi, eta, xi_dot, eta_dot, self, q)

        return self.K @ (self.xi[-1, :] - self.xi_ref) - W

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

        # integration over the body (don't need the initial point as the initial values are
        g_half = np.eye(4)  # known initial condition
        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

            # external loads
            A_bar = 0
            B_bar = 0
            # viscosity
            B_bar += self.V @ xi_dot

            # other loads
            for load in self.loads:
                A, B = load.dist_load(g_half, xi_half, eta_half, xi_dot, eta_dot, self, q)
                A_bar += A
                B_bar += B
```

```
            # spatial derivatives
            xi_der = np.linalg.inv(self.K - A_bar) @ (
                    (self.M @ eta_dot) - (adjoint(eta_half).T @ self.M @ eta_half) + (
                    adjoint(xi_half).T @ self.K @ (xi_half - self.xi_ref)) + B_bar)
            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
            g_half = g_half @ expm(se(self.ds * xi_half))

            # 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, :] +
```

Since the code is just reorganized and not changed all the previous results should hold.