Cable Actuation

With the external loads added it is time to look at more complicated loads from actuators. The simplest kind of actuator is the cable actuator. It exerts both a distributed load and a load at the tip of the rod. For this we assume that the cable is embedded within the rod and that there is no friction. There will be several changes necessary to the code to accommodate these actuators. We now need to handle time varying inputs, specifying actuator positions, and there are some complications with the distributed load that makes the integration a bit more involved. At first we will hardcode everything and then generalize as we have been doing.

The actuators will be embedded some displacement from the centerline, r_i, where i indicates the actuator. For now lets assume there are 3 cables half the radius from the center of the rod and placed symmetrically around the center. The definition of r_i is:

$$r_i = \frac{D}{4}\begin{bmatrix}\cos(\frac{2\pi}{3}i)\\ \sin(\frac{2\pi}{3}i)\\ 0\end{bmatrix}$$

Then, looking at the tip condition we see that the applied wrench must balance with the internal wrench, so the load due to the wrench balances with the stiffness:

$$K\Delta\xi(L) = W = \sum q_i \begin{bmatrix}-\hat{r}_i\vec{z}\\ \vec{z}\end{bmatrix}, \vec{z} = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

where q_i are the inputs or cable tensions.

For the distributed load we have something a bit more complicated and have to deal with the distributed load having a linear dependence of ξ'. The ξ' issue is due to having both ξ' on the left and right sides of the differential equation, so we need to rearrange to get it only on one side.

To start lets look at the dynamics equation.

$$M\dot{\eta} - ad_\eta^T M\eta - K\xi' + ad_\xi^T K\Delta\xi + \bar{W} = 0$$

Our goal for the integrator is to determine $\xi' = f(\dots)$ and since $\bar{W}$ can depend on ξ' we need to do some manipulation. First we can split $\bar{W}$ into components that do and do not depend on ξ', here it is necessary to assume linear dependence on ξ' to actually get what we want.

$$M\dot{\eta} - ad_\eta^T M\eta - K\xi' + ad_\xi^T K\Delta\xi + \bar{A}\xi' + \bar{B} = 0$$

where $\bar{A}\xi'$ is the ξ' dependence and $\bar{B}$ is independent of ξ'. Now we can rearrange to get:

$$\xi' = (K - \bar{A})^{-1}(M\dot{\eta} - ad_\eta^T M\eta + ad_\xi^T K\Delta\xi + \bar{B})$$

This is useful for the integration and for dealing with distributed loads we will have to split terms into $\bar{A}$ and $\bar{B}$ components. This means in the integration all loads independent of ξ' will be the same and any that do will need some extra work to separate into $\bar{A}$ and $\bar{B}$ components.

For cable actuators we can do this splitting; however, I won't be going over the details and just give the implementation.

```
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 statio
    """

    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])):
        # 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])
```

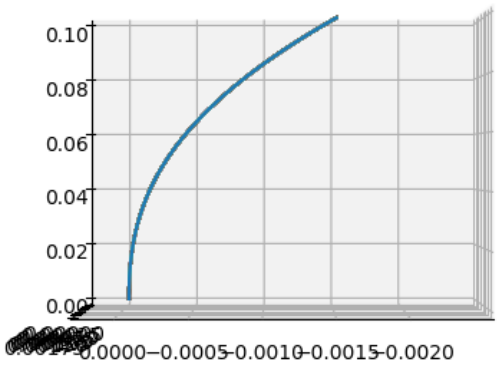
```
        self.L = L
        self.D = D
        self.rho = rho
        self.A = A
        self.xi_ref = np.array([0, 0, 0, 0, 0, 1])
        self.ds = L / (N - 1)
        self.N = N
        self.xi_init = xi_init
        self.eta_init = eta_init

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

        # just temporary for now
        self.r = lambda i: self.D/4*np.array([np.cos(2*np.pi/3*i), np.sin(2*np.pi/3*i), 0])

    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)
    W = 0
    z = np.array([0,0,1])
    for i in range(3):
        W += q[i] * np.concatenate([-skew(self.r(i)) @ z, z])
    return self.K@ (self.xi[-1, :] - self.xi_ref) - W

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

    # cable displacements
    r = self.r

    # 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
            # gravity
            R = g_half[:3,:3]
            B_bar += self.rho *self.A * np.concatenate([np.array([0,0,0]), R.T @ grav])
            # cables, have to iterate over each actuator
            omega = xi_half[:3]
            nu = xi_half[3:]
            for j in range(3):
                pa_der = R @ (nu - skew(r(j)) @ omega)
                P = R.T @ -skew(pa_der)*skew(pa_der)/np.linalg.norm(pa_der)**3 @ R
                b = P @ skew(omega) @ (nu - skew(r(j)) @ omega)
                B_bar += q[j] * np.concatenate([skew(r(j)) @ b, b])
                A_bar += q[j] * np.concatenate([np.concatenate([-skew(r(j)) @ P @ skew(r(j))


            # 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, :] +
```

Where r has been added to the rod definition and q needs to be supplied to `step`. The forces are now amended to be split into $\bar{A}$ and $\bar{B}$ and the tip condition includes the cable loads.

To check whether the implementation makes sense again we can check the previous experiments and they work fine. For checking the cables we have a less concrete test as the system won't be conservative when the cable is in tension because work is added to the system, but we can see if the general behavior looks alright. If we apply tension to one cable we expect it to bend in the direction of the cable. With a tension of 0.1 N we get:



Note that if you play around with this we start to see potential issues with the actuation. If a load is applied too suddenly then the solution can become ill-condition and the simulation won't function. Dealing with this includes changing resolutions of the simulations or requiring gradual changes to the inputs rather than instant ones.