Integration: Conservative System

For the very initial introduction to the dynamics simulations I'll be going over the simulation of a rod with no external loads and one end fixed to a wall. So, it will start in some initially bent configuration and then bend back and forth forever. This will also be a very specialized scheme to specifically this situation which will be built on later both for adding loads and for abstracting the scheme to be useful in other contexts.

System

First, the system of PDEs is:

$$(M\dot{\eta} - ad_{\eta}^{T} M\eta) - (K\xi' - ad_{\xi}^{T} K\Delta\xi) = 0 \tag{1}$$

$$\dot{\xi} = \eta' + ad_{\xi}\eta \tag{2}$$

$$g' = g\hat{\xi} \tag{3}$$

$$\dot{g} = g\hat{\eta} \tag{4}$$

with $g(0) = I_{4\times 4}$ and $\xi(L) = \xi^*$ as the boundary conditions.

Now, there are no external loads so to get some dynamic behavior we have to include some initial bending which will be done by saying:

$$\xi(t=0) = \begin{bmatrix} 0 \\ \alpha \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

where α will be set at different amounts and we have $\xi^* = [0, 0, 0, 0, 0, 1]^T$.

For the geometry of the rod we will have L=10cm and D=1cm and for the material properties we will have $\rho = 1000 g/cm^3$, E=1MPa, and $G = E/3$. This should specify all the necessary constants.

Discretization

For the discretization process first we discretize the PDEs implicitly in time to get ODEs in space. Then we discretize the ODEs explicitly in space to do the integration. For this we will be using the implicit midpoint rule to discretize in time ($y_{i+1} = y_i + \Delta t \dot{y}_{i+1/2}$) and the explicit Euler in space ($y_{i+1} = y_i + \Delta s y'_i$). We will also use the RKMK versions for dealing with g.

We first approximate the time derivatives implicitly:

$$\dot{\xi}_{i+1/2} = \frac{1}{\Delta t}(\xi_{i+1} - \xi_i) \tag{6}$$

$$\dot{\eta}_{i+1/2} = \frac{1}{\Delta t}(\eta_{i+1} - \eta_i) \tag{7}$$

where the subscripts denote the time steps. It is important to notice the derivatives are at the half step, this will require a bit of extra work to make sure the stepping occurs on whole numbered time steps and may cause a little confusion at first.

So, now we can substitute the derivative approximations into the PDEs. When doing this we have to make sure the entire ODE is evaluated at the half step value to be consistent.

$$\xi'_{i+1/2} = K^{-1}(M(\frac{1}{\Delta t}(\eta_{i+1} - \eta_i)) - ad^T_{\eta_{i+1/2}} M\eta_{i+1/2} + ad^T_{\xi_{i+1/2}} K\Delta\xi_{i+1/2}) \tag{8}$$

$$\eta'_{i+1/2} = \frac{1}{\Delta t}(\xi_{i+1} - \xi_i) - ad_{\xi_{i+1/2}}\eta_{i+1/2} \tag{9}$$

Now, we have ODEs in space that we can integrate with any explicit method we'd like, for simplicity we use the explicit Euler method. However, there are a few details that need to be worked out first in terms of dealing with the half steps and the boundary conditions.

In order to integrate $\xi_{i+1/2}$ and $\eta_{i+1/2}$ we need to know their initial values. For η this is easy, we know the base is fixed and therefore $\eta(s = 0) = 0$. However, for ξ we don't know the initial condition only the final condition ($\xi(s = L) = \xi^*$). This means we have a boundary value problem and we choose to approach this with a shooting method. A shooting method works by first guessing the unknown initial condition, integrating the system, and then checking if the boundary condition is met. If the condition is met the system is solved and if it is not met the guess for the initial condition is updated and the process repeated. So, we have to guess the initial condition for $\xi_{i+1/2}$.

To deal with the half step values I choose to average the value between time steps:

$$\xi_{i+1/2} = \frac{1}{2}(\xi_{i+1} + \xi_i) \tag{10}$$

$$\eta_{i+1/2} = \frac{1}{2}(\eta_{i+1} + \eta_i) \tag{11}$$

So, what we will really do is guess ξ_{i+1} and use the integration and half step averaging to integrate the system.

For the stepping of g we have both a space and time stepping, although due to the system being left-invariant (global rigid body displacements do not effect the physics) we only have to do the time stepping, but both are necessary later.

In space we have:

$$g_{i+1/2}^{j+1} = g_{i+1/2}^{j} \exp(\Delta s \xi_{i+1/2}^{j}) \tag{12}$$

where the superscipts denote the node in space.

In time we have:

$$g_{i+1}^{j} = g_{i}^{j} \exp(\Delta t \eta_{i+1/2}^{j}) \tag{13}$$

Implementation

Linear Algebra Imports

To start we need to import a some useful functions and numpy for doing the linear algebra.

```
import numpy as np
from scipy.optimize import fsolve
from scipy.linalg import expm
import matplotlib.pyplot as plt
```

`fsolve` is for the equation solving and `expm` is the matrix exponential.

Useful Functions

Then for the implementation we need a few useful functions for dealing with the different transformations between matrix and algebra forms.

```
# map a vector to a skew symmetric matrix
def skew(x):
    return np.array([[0, -x[2], x[1]], [x[2], 0, -x[0]], [-x[1], x[0], 0]])

# map a twist to its adjoint form
def adjoint(x):
    return np.concatenate(
        [np.concatenate([skew(x[:3]), np.zeros((3, 3))], 1), np.concatenate([skew(x[3:]), sk

# flatten a homogeneous transformation matrix to a vector
def flatten(g):
    return np.concatenate([np.reshape(g[:3, :3], (9,)), g[:3, 3]])

# unflatten a homogeneous transformation
def unflatten(g):
```

```
    return np.row_stack((np.column_stack((np.reshape(g[:9], (3, 3)), g[9:])), np.array([0, 0
```

```
# the matrix representation of a twist vector
def se(x):
    return np.row_stack((np.column_stack((skew(x[:3]), x[3:])), np.array([0, 0, 0, 0])))
```

These will provide the necessary manipulations for the arithmetic and the storage of values (specifically for g with `flatten`).

Initial Conditions

In order to actually run the simulations we need to initialize the state of the rod to begin the process. We will need the initial values for g, ξ, and η. Since this case is just a rod bending back and forth we will start it in a bent position which is initially not moving. That means $\eta = 0$ and for ξ we can just select some constant value for each spatial point, here I picked $\xi = [0, pi/(4 * L), 0, 0, 0, 1]^T$. Determining g can then be done with the explicit Euler RKMK. The only thing we will need to specify is the number of discretizations, N, to use.

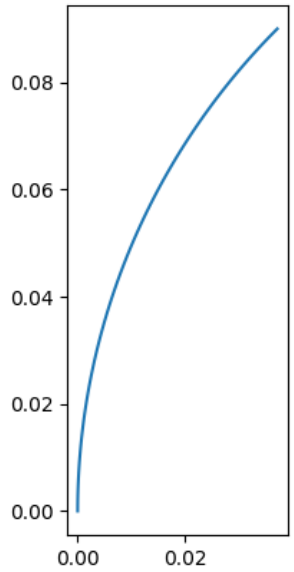
```
def initRod(N):
    L = 10e-2 # length of the rod

    g = np.zeros((N, 12))
    xi = np.repeat(np.array([[0, np.pi/4/L, 0, 0, 0, 1]]), N, 0)
    eta = np.zeros((N, 6))

    #explicit Euler RKMK
    G = np.eye(4)
    ds = L / (N - 1)
    g[0, :] = flatten(G)
    for i in range(1, N):
        G = G @ expm(se(ds * xi[i - 1, :]))
        g[i, :] = flatten(G)

    return g, xi, eta
```

This gives a pre-bent rod to start simulating. It should be noted that having to specify/hardcode L is a bit inconvenient, so it'll be something to address later. To check to see this works lets plot the rod to see if it makes sense. Note that technically the tip condition is not met given this initial configuration, but this shouldn't be a problem.

```
# start the figure
fig, ax = plt.subplots()
# get the initial conditions
g, xi, eta = initRod(100)
# make a 2D plot
ax.plot(g[:,9], g[:,11])
```

```
# make the axes equal
ax.set_aspect('equal')
# show it
plt.show()
```

The resulting image is:



Which seems like a sensible looking plot.

Integrator

For the integration there are several components. First, we are taking one time step into the future which constitutes solving the tip boundary condition and determining the initial condition for $\xi(0)$. Then, once the system is solved it is integrated to determine the next state's values. The functoin for this will look like:

```
def step(g, xi, eta):
    # determine xi0 by solving tip condition
    xi0 = fsolve(lambda x: condition(g, xi, eta, x), xi[0, :])
    # integrate the system with the solved xi0
    return integrate(g, xi, eta, xi0)
```

We use the previous value for $\xi(0)$ as an initial guess because it should be be pretty close. Now we need to implement `condition` and `integrate`. For the

condition we just need to integrate the system given the guess for $\xi(0)$ and then see if $\xi(L) = \xi^*$.

```
def condition(g, xi, eta, xi0):
    g_next, xi_next, eta_next = integrate(g, xi, eta, xi0)
    return xi_next[-1, :] - np.array([0, 0, 0, 0, 0, 1])
```

Now we need to implement the integration scheme that uses implicit midpoint in time and the explicit Euler in space. This is a pretty mechanical process from the derivations and the result is:

```
def integrate(g, xi, eta, xi0):
    # initialize empty matrices for storage
    g_next = np.zeros_like(g)
    xi_next = np.zeros_like(xi)
    eta_next = np.zeros_like(eta)

    # determine number of spatial points, just believe everything is the right size
    (N, _) = xi.shape

    # set the guessed value
    xi_next[0, :] = xi0

    # material and geometric properties
    xi_ref = np.array([0, 0, 0, 0, 0, 1])
    L = 10e-2
    D = 1e-2
    E = 1e6
    rho = 1e3
    ds = L / (N - 1)
    dt = 0.01
    A = np.pi / 4 * D ** 2
    I = np.pi / 64 * D ** 4
    J = 2 * I
    G = E / 3
    K = np.diag([E * I, E * I, G * J, G * A, G * A, E * A])
    M = rho * np.diag([I, I, J, A, A, A])

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

        # implicit midpoint approximation
        xi_dot = (xi_next[i, :] - xi[i, :]) / dt
        eta_dot = (eta_next[i, :] - eta[i, :]) / dt
```

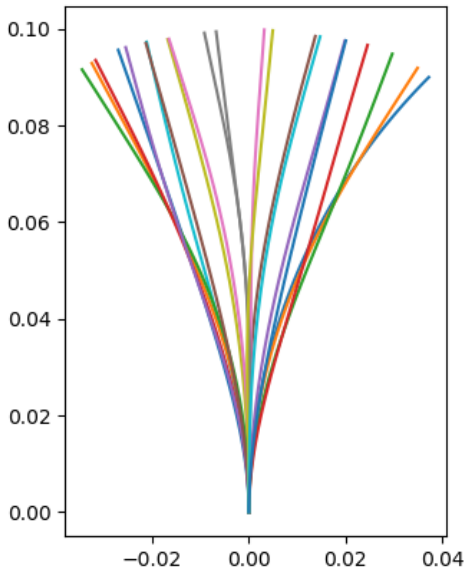
```
        # spatial derivatives
        xi_der = np.linalg.inv(K) @ (
                (M @ eta_dot) - (adjoint(eta_half).T @ M @ eta_half) + (adjoint(xi_half).T @
        eta_der = xi_dot - (adjoint(xi_half) @ eta_half)

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

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

    # midpoint RKMK to step the g values
    for i in range(N):
        g_next[i, :] = flatten(unflatten(g[i,:]) @ expm(se(dt * (eta_next[i,:] + eta[i,:])/2

    return g_next, xi_next, eta_next
```

This fully defines the integrator. We can see that specifying the material properties and such in the integrator is a bit wasteful, every iteration the data gets allocated and freed even though it never changes and it is inconvenient to update. Both of these will be addressed later.

To get an initial glance at how well this performs we run several steps of the dynamics to see how it looks.

```
def plotDynamics(N, steps):
    # start figure
    fig, ax = plt.subplots()
    g, xi, eta = initRod(N)
    ax.plot(g[:,9], g[:,11])
    ax.set_aspect('equal')
    plt.pause(0.01) # make the plots show up as they're updated

    for i in range(steps):
        g, xi, eta = step(g, xi, eta)
        ax.plot(g[:,9], g[:,11])
        plt.pause(0.01) # make the plots show up as they're updated

    #make sure it stays open for looking at and saving
    plt.show()
```

Running `plotDynamics(100, 20)` we get:



Which wobbles back and forth as expected.

Now we want to see if it is actually energy preserving as claimed. For this we want to approximate the energy in the system which is just summing the kinetic and strain energy over the whole body. We do this as:

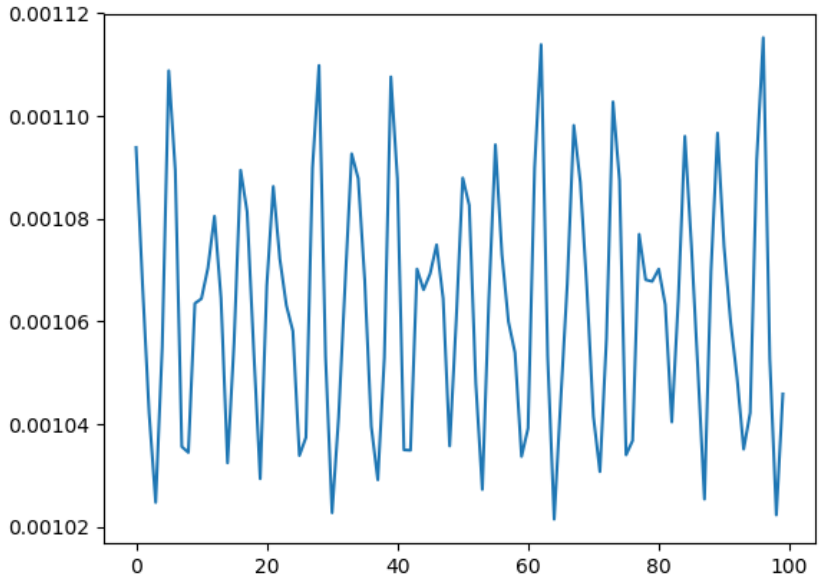
```
def energy(xi,eta):
    # similar to the setup for the integrator
    (N, _) = xi.shape
    xi_ref = np.array([0, 0, 0, 0, 0, 1])
    L = 10e-2
    D = 1e-2
    E = 1e6
    rho = 1e3
    ds = L / (N - 1)
    dt = 0.01
    A = np.pi / 4 * D ** 2
    I = np.pi / 64 * D ** 4
    J = 2 * I
    G = E / 3
    K = np.diag([E * I, E * I, G * J, G * A, G * A, E * A])
    M = rho * np.diag([I, I, J, A, A, A])

    H = 0 # total energy
```

```
    # integrate over the rod
    for i in range(N):
        T = eta[i,:].T @ M @ eta[i,:]
        U = (xi[i,:]-xi_ref).T @ K @ (xi[i,:]-xi_ref)
        H += 1/2*(T + U)
    return ds*H #multiply by discrete step size to scale
```

Then running several steps of the dynamics and recording the energy:

```
def plotEnergy(N, steps):
    fig, ax = plt.subplots()
    g, xi, eta = initRod(N)
    E = []

    for i in range(steps):
        g, xi, eta = step(g, xi, eta)
        E.append(energy(xi,eta))

    ax.plot(E)
    plt.show()
```

Running `plotEnergy(100, 100)` we get:



Here we see that there is oscillation of the energy; however, it is rather small

and due to numerical precision. The energy continues to oscillate around the same point over a long time indicating the energy is preserved.

It is worth trying the simulations with different numbers of spatial points and lenths of time steps. You should see instability isues when the time step and the number of spatial points are small.

The full code is posted below and in conservative.py

```
import numpy as np
from scipy.optimize import fsolve
from scipy.linalg import expm
import matplotlib.pyplot as plt

# Some utilities
# map a vector to a skew symmetric matrix
def skew(x):
    return np.array([[0, -x[2], x[1]], [x[2], 0, -x[0]], [-x[1], x[0], 0]])

# map a twist to its adjoint form
def adjoint(x):
    return np.concatenate(
        [np.concatenate([skew(x[:3]), np.zeros((3, 3))], 1), np.concatenate([skew(x[3:]), sk

# flatten a homogeneous transformation matrix to a vector
def flatten(g):
    return np.concatenate([np.reshape(g[:3, :3], (9,)), g[:3, 3]])

# unflatten a homogeneous transformation
def unflatten(g):
    return np.row_stack((np.column_stack((np.reshape(g[:9], (3, 3)), g[9:])), np.array([0, 0

# the matrix representation of a twist vector
def se(x):
    return np.row_stack((np.column_stack((skew(x[:3]), x[3:])), np.array([0, 0, 0, 0])))


# Initialization
def initRod(N):
    L = 10e-2 # length of the rod

    g = np.zeros((N, 12))
    xi = np.repeat(np.array([[0, np.pi/4/L, 0, 0, 0, 1]]), N, 0)
    eta = np.zeros((N, 6))

    #explicit Euler RKMK
    G = np.eye(4)
```

```
    ds = L / (N - 1)
    g[0, :] = flatten(G)
    for i in range(1, N):
        G = G @ expm(se(ds * xi[i - 1, :]))
        g[i, :] = flatten(G)

    return g, xi, eta

#Integration
def step(g, xi, eta):
    # determine xi0 by solving tip condition
    xi0 = fsolve(lambda x: condition(g, xi, eta, x), xi[0, :])
    # integrate the system with the solved xi0
    return integrate(g, xi, eta, xi0)

def condition(g, xi, eta, xi0):
    g_next, xi_next, eta_next = integrate(g, xi, eta, xi0)
    return xi_next[-1, :] - np.array([0, 0, 0, 0, 0, 1])

def integrate(g, xi, eta, xi0):
    # initialize empty matrices for storage
    g_next = np.zeros_like(g)
    xi_next = np.zeros_like(xi)
    eta_next = np.zeros_like(eta)

    # determine number of spatial points, just believe everything is the right size
    (N, _) = xi.shape

    # set the guessed value
    xi_next[0, :] = xi0

    # material and geometric properties
    xi_ref = np.array([0, 0, 0, 0, 0, 1])
    L = 10e-2
    D = 1e-2
    E = 1e6
    rho = 1e3
    ds = L / (N - 1)
    dt = 0.01
    A = np.pi / 4 * D ** 2
    I = np.pi / 64 * D ** 4
    J = 2 * I
    G = E / 3
    K = np.diag([E * I, E * I, G * J, G * A, G * A, E * A])
    M = rho * np.diag([I, I, J, A, A, A])
```

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

        # implicit midpoint approximation
        xi_dot = (xi_next[i, :] - xi[i, :]) / dt
        eta_dot = (eta_next[i, :] - eta[i, :]) / dt

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

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

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

    # midpoint RKMK to step the g values
    for i in range(N):
        g_next[i, :] = flatten(unflatten(g[i,:]) @ expm(se(dt * (eta_next[i,:] + eta[i,:])/2

    return g_next, xi_next, eta_next


# Testing functions
def plotDynamics(N, steps):
    # start figure
    fig, ax = plt.subplots()
    g, xi, eta = initRod(N)
    ax.plot(g[:,9], g[:,11])
    ax.set_aspect('equal')
    plt.pause(0.01) # make the plots show up as they're updated

    for i in range(steps):
        g, xi, eta = step(g, xi, eta)
        ax.plot(g[:,9], g[:,11])
        plt.pause(0.01) # make the plots show up as they're updated

    #make sure it stays open for looking at and saving
    plt.show()
```

```
def energy(xi,eta):
    # similar to the setup for the integrator
    (N, _) = xi.shape
    xi_ref = np.array([0, 0, 0, 0, 0, 1])
    L = 10e-2
    D = 1e-2
    E = 1e6
    rho = 1e3
    ds = L / (N - 1)
    dt = 0.01
    A = np.pi / 4 * D ** 2
    I = np.pi / 64 * D ** 4
    J = 2 * I
    G = E / 3
    K = np.diag([E * I, E * I, G * J, G * A, G * A, E * A])
    M = rho * np.diag([I, I, J, A, A, A])

    H = 0 # total energy

    # integrate over the rod
    for i in range(N):
        T = eta[i,:].T @ M @ eta[i,:]
        U = (xi[i,:]-xi_ref).T @ K @ (xi[i,:]-xi_ref)
        H += 1/2*(T + U)
    return ds*H #multiply by discrete step size to scale

def plotEnergy(N, steps):
    fig, ax = plt.subplots()
    g, xi, eta = initRod(N)
    E = []

    for i in range(steps):
        g, xi, eta = step(g, xi, eta)
        E.append(energy(xi,eta))

    ax.plot(E)
    plt.show()

# Call the script as python conservative.py
if __name__ == "__main__":
    # plotDynamics(100, 20)
    plotEnergy(100,100)
```