clear, close all

%Testing calculating the Jacobian for 4 sections

% Q = [1000 0

% 0 1];

%Weights for LQI

Q = [1 0 0

0 1 0

0 0 5000];

R = 1;

%Physical constants

m = .01; %lbfs^2/in

b = 1; %lbfs/in

k = 15; %lbf/in - modleing something like 60psi/4in

umax = 65; %lbf

%Inputs

n = 4;

r = 5\*2/pi/2; %in

%Find a desired pose

des\_pose = [10 -15 10 -1 .1 .5]';

%Constants for optimization

gamma = .01;

Kp = 2;

%Initial state

q0 = [4 4 4 4 4 4 4 4 4 4 4 4]'; %Test lengths

vel0 = zeros(3\*n,1);

integral\_in = zeros(3\*n,1);

q\_steady = 4\*ones(3\*n,1);

%Simulation parameters

num\_cycles = 200;

tspan = .025;

%Set up simulation

pose\_new = zeros(num\_cycles,6);

q\_set = zeros(num\_cycles,3\*n);

u\_tracker = zeros(num\_cycles,3\*n);

new\_q = q0;

new\_vel = vel0';

max\_delta = 2;

time = 0:tspan:(tspan\*(num\_cycles-1));

%Simulate

for i = 1:num\_cycles

q\_set(i,:) = new\_q;

[deltaL,pose\_new(i,:)] = find\_deltaL\_new(new\_q,des\_pose,gamma,Kp,r,n);

%Code ensuring that you don't go over limits

for k = 1:length(deltaL)

if abs(deltaL(k))>max\_delta

if deltaL(k) >0

deltaL(k) = max\_delta;

else

deltaL(k) = -max\_delta;

end

end

end

for j = 1:length(deltaL)

if (new\_q(j)+deltaL(j))>= 8

deltaL(j) = 0;

elseif (new\_q(j)+deltaL(j)) <= 4

deltaL(j) = 0;

end

end

new\_ref = new\_q+deltaL;

%Here I have to factor in the initial lengths

new\_q\_converted = new\_q-q\_steady;

new\_ref\_converted = new\_ref-q\_steady;

%Lets try out the new control function

[next\_state,u\_out,integral\_in] = LQI\_control(Q,R,m,b,k,umax,new\_q\_converted,new\_vel',new\_ref\_converted,tspan,false,integral\_in);

%next\_state = control(new\_q\_converted',new\_vel,new\_ref\_converted',tspan,1,false,false);

new\_q = next\_state(1,:)'+q\_steady;

new\_vel = next\_state(2,:);

u\_tracker(i,:) = u\_out';

end

%Plot results

plot(time,u\_tracker(:,2))

figure('Position',[10 10 1000 600])

tl = tiledlayout(3,2);

title(tl,"Robot Behavior n=4, Desired Pose = [ 10, -15, 10, -1, .1, .5]")

nexttile

plot(time,pose\_new(:,1))

ylabel('Robot x Position')

xlabel('Time (s)')

nexttile

plot(time,pose\_new(:,2))

ylabel('Robot y Position')

xlabel('Time (s)')

nexttile

plot(time,pose\_new(:,3))

ylabel('Robot z Position')

xlabel('Time (s)')

nexttile

plot(time,pose\_new(:,4))

ylabel('Robot Roll')

xlabel('Time (s)')

nexttile

plot(time,pose\_new(:,5))

ylabel('Robot Pitch')

xlabel('Time (s)')

nexttile

plot(time,pose\_new(:,6))

ylabel('Robot Yaw')

xlabel('Time (s)')

hold off

figure('Position',[10 10 1000 500])

for i = 1:3\*n

plot(time,u\_tracker(:,i))

hold on

end

title('Force Input for all Actuators')

xlabel('Time (s)')

ylabel('Force Input (lbf)')

animate\_from\_q\_matrix(q\_set,r ,.1)

%Lets try a better optimization function :)

function [deltaL,act\_pose] = find\_deltaL\_new(q,des\_pose,gamma,Kp,r,n)

%J = calc\_J(n,r,q);

J = numerical\_jacobian(q,r,n); %Numerical calculation, works somehow

tip\_T = find\_tip(q,r,n);

tip\_T = real(tip\_T);

%Convert to an actual pose

tip\_pos = tip\_T(1:3,4); %Tip position

%tip\_eul = rotm2eul(real(tip\_T(1:3,1:3)))'; % returns [yaw, pitch, roll]

%act\_pose = [tip\_pos;tip\_eul];

R\_des = eul2rotm(des\_pose(4:6)', 'ZYX'); % Assuming ZYX Euler input

R\_act = tip\_T(1:3, 1:3);

% Rotation error matrix

R\_err = R\_des \* R\_act'; % NOTE: Not R\_act \* R\_des'

% Convert rotation error to rotation vector (axis \* angle)

% I don't really undersatand this part but its mroe stable than using

% Euler angles

axang = rotm2axang(real(R\_err)); % [axis(1:3), angle]

e\_rot = axang(1:3)' \* axang(4); % rotation vector (3x1)

% Full 6D error: [position\_error; rotation\_vector]

pos\_err = des\_pose(1:3) - tip\_pos;

%TEST

error = [pos\_err; e\_rot];

% Output actual pose for logging

act\_eul = rotm2eul(real(R\_act), 'ZYX')';

act\_pose = [tip\_pos; act\_eul];

act\_pose = real(act\_pose);

%deltaL = (J'\*J+gamma\*eye(3\*n))\(J'\*(Kp\*error));

H = J'\*J + gamma\*eye(3\*n);

f = -J' \* (Kp \* error);

lb = 4-q;

ub = 8-q;

% Use quadprog to solve the QP

options = optimoptions('quadprog','Display','off');

deltaL = quadprog(H, f, [], [], [], [], lb, ub, [], options);

end

%Lets tryout LQI control :)

function [next\_state, u\_out, updated\_integral] = LQI\_control(Q, R, m, b, k, u\_max, q\_act, vel\_act, q\_ref, tstep, fullsim, integral\_in)

n = length(q\_act);

tspan = 10;

if fullsim

t\_mat = 0:tstep:tspan;

else

t\_mat = 0:tstep/2:tstep;

end

% Preallocate

u\_out = zeros(n, 1);

next\_state = zeros(2, n);

updated\_integral = zeros(n, 1); % Output for integral term

% System matrices

A = [0 1;

-k/m -b/m];

B = [0;

1/m];

C = [1 0];

% Augmented system

sys = ss(A, B, C, 0);

[K, ~, ~] = lqi(sys, Q, R);

Kx = K(:, 1:2); % Feedback

Ki = K(:, 3); % Integral

for i = 1:n

x\_0 = [q\_act(i);

vel\_act(i)];

ref = q\_ref(i);

z\_0 = integral\_in(i);

if fullsim

x\_aug\_0 = [x\_0; z\_0];

[t\_LQI, x\_LQI] = ode45(@(t, x) SMD\_LQI(x, m, b, k, Kx, Ki, ref, u\_max), t\_mat, x\_aug\_0);

final = x\_LQI(end, :);

next\_state(:, i) = final(1:2);

updated\_integral(i) = final(3);

u\_out(i) = max(0, min(u\_max, -Kx \* final(1:2)' - Ki \* final(3)));

else

% Update integral manually for one step

err = ref - x\_0(1);

z\_new = z\_0 + err \* tstep;

u\_in = -Kx \* x\_0 - Ki \* z\_new;

u\_in = max(0, min(u\_max, u\_in));

[t\_LQI, x\_LQI] = ode45(@(t, x) SMD(x, m, b, k, u\_in), t\_mat, x\_0);

% Store final values

final = x\_LQI(end, :);

next\_state(1, i) = final(1);

next\_state(2, i) = final(2);

updated\_integral(i) = z\_new;

u\_out(i) = u\_in;

end

end

if fullsim

plot(t\_LQI, x\_LQI(:,1));

xlabel('Time (s)');

ylabel('Position');

title('LQI Response');

end

end

% LQI system with augmented state

function dx = SMD\_LQI(x\_aug, m, b, k, Kx, Ki, ref, u\_max)

x = x\_aug(1:2);

z = x\_aug(3);

u = -Kx \* x - Ki \* z;

u = max(0, min(u\_max, u));

dx1 = x(2);

dx2 = (1/m) \* (-k \* x(1) - b \* x(2) + u);

dz = ref - x(1);

dx = [dx1;

dx2;

dz];

end

%Function for animation

function animate\_from\_q\_matrix(q\_matrix, r\_real, frame\_delay)

% Animate from a n\_samples x 3\*n matrix

[num\_samples, q\_length] = size(q\_matrix);

if mod(q\_length, 3) ~= 0

error('Each row of q\_matrix must be a multiple of 3 (i.e., 3\*n).');

end

% array of q vectors

q\_sequences = cell(1, num\_samples);

for i = 1:num\_samples

q\_sequences{i} = q\_matrix(i, :);

end

% run animation

animate\_vine\_robot(q\_sequences, r\_real, frame\_delay);

end

function animate\_vine\_robot(q\_sequences, r\_real, frame\_delay)

% Animate multiple vine robot configurations given a vector of q values

% defaults for radius and frame delay

if nargin < 2 || isempty(r\_real)

r\_real = 5\*2/pi/2;

end

if nargin < 3 || isempty(frame\_delay)

frame\_delay = 0.5;

end

% input debugging

for i = 1:length(q\_sequences)

if mod(length(q\_sequences{i}), 3) ~= 0

error('Each q vector must have a length that is a multiple of 3');

end

end

% create figure

fig = figure;

axis equal;

grid on;

xlim([-20 20]);

ylim([-20 20]);

zlim([0 40]);

hold on;

view(3);

xlabel('X'); ylabel('Y'); zlabel('Z');

title('Vine Robot Animation');

% initial visualization with first config

h = visualize\_configuration(q\_sequences{1}, r\_real);

% loop the animation

for seq\_idx = 1:length(q\_sequences)

q = q\_sequences{seq\_idx};

% clear previous visualization

delete(get(gca, 'Children'));

% update the visualization

h = visualize\_configuration(q, r\_real);

% pause between the frames

drawnow;

pause(frame\_delay);

% break if figure gets closed

if ~isvalid(fig)

break;

end

end

end

function h = visualize\_configuration(q, r\_real)

% Create a visualization for a single configuration

% Actuator start positions (column vectors)

act\_11\_start = [r\_real; 0; 0; 1];

act\_12\_start = [r\_real\*cos(2\*pi/3); r\_real\*sin(2\*pi/3); 0; 1];

act\_13\_start = [r\_real\*cos(4\*pi/3); r\_real\*sin(4\*pi/3); 0; 1];

% Base coordinate system

Base\_coord = eye(4);

% Generate transformation matrices and collect all points

prev\_T = Base\_coord;

num\_segments = length(q)/3;

all\_centers = [Base\_coord(1:3,4)];

all\_actuator\_points = cell(3, num\_segments + 1);

% Store initial points

all\_actuator\_points{1,1} = act\_11\_start(1:3);

all\_actuator\_points{2,1} = act\_12\_start(1:3);

all\_actuator\_points{3,1} = act\_13\_start(1:3);

for seg = 1:num\_segments

q\_start = (seg-1)\*3 + 1;

[T\_k, act\_1\_end, act\_2\_end, act\_3\_end] = ...

gen\_transform\_2(q(q\_start), q(q\_start+1), q(q\_start+2), r\_real, ...

act\_11\_start, act\_12\_start, act\_13\_start, prev\_T);

prev\_T = prev\_T \* T\_k;

all\_centers(:,seg+1) = prev\_T(1:3,4);

% Store actuator endpoints

all\_actuator\_points{1,seg+1} = act\_1\_end(1:3);

all\_actuator\_points{2,seg+1} = act\_2\_end(1:3);

all\_actuator\_points{3,seg+1} = act\_3\_end(1:3);

end

% Initialize handles structure

h = struct();

% Plot center points

h.centers = plot3(all\_centers(1,:), all\_centers(2,:), all\_centers(3,:), ...

'.', 'MarkerSize', 10, 'Color', 'black');

% Draw smooth continuous curves for each actuator

colors = ['r', 'g', 'b']; % Actuator colors

h.actuators = gobjects(1,3);

for act = 1:3

% Collect all points for this actuator

actuator\_pts = zeros(3, size(all\_actuator\_points,2));

for seg = 1:size(all\_actuator\_points,2)

actuator\_pts(:,seg) = all\_actuator\_points{act,seg};

end

% Fit a smooth spline through all points

t = cumsum([0, sqrt(sum(diff(actuator\_pts,1,2).^2,1))]);

tt = linspace(0,t(end),100);

xx = spline(t, actuator\_pts(1,:), tt);

yy = spline(t, actuator\_pts(2,:), tt);

zz = spline(t, actuator\_pts(3,:), tt);

% Plot the smooth curve

h.actuators(act) = plot3(xx, yy, zz, 'Color', colors(act), 'LineWidth', 2);

end

% Draw cross-sections

h.cross\_sections = gobjects(1, size(all\_actuator\_points,2));

h.actuator\_markers = gobjects(3, size(all\_actuator\_points,2));

for seg = 1:size(all\_actuator\_points,2)

% Get the three actuator points for this cross-section

P1 = all\_actuator\_points{1,seg};

P2 = all\_actuator\_points{2,seg};

P3 = all\_actuator\_points{3,seg};

% Calculate the normal vector to the cross-section plane

normal = cross(P2 - P1, P3 - P1);

if norm(normal) < 1e-6

normal = [0; 0; 1]; % Default if points are colinear

else

normal = normal / norm(normal);

end

% Calculate the center point

center = mean([P1, P2, P3], 2);

% Calculate radius

radius = mean([norm(P1-center), norm(P2-center), norm(P3-center)]);

% Create a circle in the cross-section plane

v1 = P1 - center;

v1 = v1 / norm(v1);

v2 = cross(normal, v1);

v2 = v2 / norm(v2);

theta = linspace(0, 2\*pi, 100);

circle\_points = center + radius\*(v1\*cos(theta) + v2\*sin(theta));

% Plot the circle

h.cross\_sections(seg) = plot3(circle\_points(1,:), circle\_points(2,:), circle\_points(3,:), ...

'k-', 'LineWidth', 1);

% Mark the actuator points on the circle

h.actuator\_markers(1,seg) = plot3(P1(1), P1(2), P1(3), 'ro', 'MarkerFaceColor', 'r', 'MarkerSize', 3);

h.actuator\_markers(2,seg) = plot3(P2(1), P2(2), P2(3), 'go', 'MarkerFaceColor', 'g', 'MarkerSize', 3);

h.actuator\_markers(3,seg) = plot3(P3(1), P3(2), P3(3), 'bo', 'MarkerFaceColor', 'b', 'MarkerSize', 3);

end

end

function [T\_k, act\_1\_end, act\_2\_end, act\_3\_end, rho\_k] = ...

gen\_transform\_2(L\_k1, L\_k2, L\_k3, r, act\_1\_start, act\_2\_start, act\_3\_start, T\_prev)

phi\_kj = 0; % fixed angle

% calculate length

L\_ck = (L\_k1 + L\_k2 + L\_k3)/3;

beta\_k = 2\*sqrt(L\_k1^2 + L\_k2^2 + L\_k3^2 - L\_k1\*L\_k2 - L\_k1\*L\_k3 - L\_k2\*L\_k3)/(3\*r);

theta\_k = atan2(3\*(L\_k2 - L\_k3), sqrt(3)\*(L\_k2 + L\_k3 - 2\*L\_k1));

rho\_k = beta\_k/L\_ck; % Curvature calc

% Rotation matrix

ct = cos(theta\_k);

st = sin(theta\_k);

cb = cos(beta\_k);

sb = sin(beta\_k);

R\_k = [

cb\*ct^2 + st^2, (cb-1)\*ct\*st, ct\*sb;

(cb-1)\*ct\*st, ct^2 + cb\*st^2, st\*sb;

-ct\*sb, -st\*sb, cb

];

% Position vector

P\_k = (1/rho\_k) \* [(1-cb)\*ct; (1-cb)\*st; sb];

% Transformation matrix

T\_k = [R\_k P\_k; 0 0 0 1];

if abs(rho\_k) < 1e-6

T\_k = [1 0 0 0

0 1 0 0

0 0 1 L\_ck

0 0 0 1];

end

% actuator end point calculation

act\_1\_end = T\_prev\*T\_k\*act\_1\_start;

act\_2\_end = T\_prev\*T\_k\*act\_2\_start;

act\_3\_end = T\_prev\*T\_k\*act\_3\_start;

end

%This works better than the anayltical

function J\_numeric = numerical\_jacobian(q, r, n, epsilon)

%NUMERICAL\_JACOBIAN Numerically compute the 6x(3n) Jacobian matrix

% using finite differences on actuator lengths q.

%

% Inputs:

% q - actuator lengths vector (3n x 1)

% r - robot radius or relevant parameter

% n - number of segments

% epsilon - small perturbation value (optional, default 1e-6)

%

% Output:

% J\_numeric - numerical Jacobian matrix (6 x 3n)

%

% Pose is [position; rotation\_vector] where rotation\_vector is axis-angle

if nargin < 4

epsilon = 1e-6;

end

m = length(q); % number of actuator variables

J\_numeric = zeros(6, m);

% Helper function to convert rotation matrix to rotation vector safely

function rotvec = rotm2rotvec\_safe(R)

axang = rotm2axang(R); % [axis(1:3), angle]

angle = axang(4);

if abs(angle) < 1e-5

% Angle near zero → no rotation, set rotation vector to zero

rotvec = zeros(3,1);

else

rotvec = axang(1:3)' \* angle;

end

end

% Compute nominal pose

T0 = find\_tip(q, r, n);

p0 = T0(1:3,4);

R0 = T0(1:3,1:3);

rotvec0 = rotm2rotvec\_safe(R0);

pose0 = [p0; rotvec0];

% Compute finite difference for each actuator variable

for i = 1:m

dq = zeros(m,1);

dq(i) = epsilon;

q\_perturbed = q + dq;

T1 = find\_tip(q\_perturbed, r, n);

p1 = T1(1:3,4);

R1 = T1(1:3,1:3);

rotvec1 = rotm2rotvec\_safe(R1);

pose1 = [p1; rotvec1];

% Numerical partial derivative

J\_numeric(:, i) = (pose1 - pose0) / epsilon;

end

% === Inject artificial rotation near singularities ===

lambda = 1000; % small artificial gain

tol = 1e-5; % tolerance for detecting singularity

for k = 1:n

idx = (k-1)\*3 + (1:3); % Indices for segment k

qk = q(idx); % Lengths for segment k

% Check if lengths are nearly equal (singular configuration)

if max(qk) - min(qk) < tol

% Inject structured rotation sensitivity

J\_numeric(4:6, idx) = J\_numeric(4:6, idx) + lambda \* ...

[1 0 -1;

0 1 -1;

-1 1 0];

end

end

end

%Finction that finds orientation of the end effector

function T\_tot = find\_tip(q,r,n)

T\_tot = eye(4);

for i = 1:n %For each segment

L\_1 = q(1+3\*(i-1));

L\_2 = q(2+3\*(i-1));

L\_3 = q(3+3\*(i-1));

T\_k = gen\_transform(L\_1,L\_2,L\_3,r);

T\_tot = T\_tot\*T\_k;

end

end

%Function that finds T\_k from set of lengths and r

function T\_k = gen\_transform(L\_k1,L\_k2,L\_k3,r)

%Credit to https://doi.org/10.1038/s41467-024-54327-6 for the modeling

%walkthrough

%Might need to define phi\_kj

%Length matrix

%Length of center

L\_ck = (L\_k1+L\_k2+L\_k3)/3;

%One angle

beta\_k = 2\*sqrt(L\_k1^2+L\_k2^2+L\_k3^2-L\_k1\*L\_k2-L\_k1\*L\_k3-L\_k2\*L\_k3)/(3\*r);

%Another angle - see paper

theta\_k = atan2(3\*(L\_k2-L\_k3),sqrt(3)\*(L\_k2+L\_k3-2\*L\_k1));

%How to calculate the length of each actuator

rho\_k = beta\_k/L\_ck;

%Logic for if straight

if abs(rho\_k) < 1e-6

T\_k = [1 0 0 0

0 1 0 0

0 0 1 L\_ck

0 0 0 1];

return

end

R\_k = [cos(beta\_k)\*cos(theta\_k)^2 + sin(theta\_k)^2 (-1+cos(beta\_k))\*cos(theta\_k)\*sin(theta\_k) cos(theta\_k)\*sin(beta\_k)

(-1+cos(beta\_k))\*cos(theta\_k)\*sin(theta\_k) cos(theta\_k)^2+cos(beta\_k)\*(sin(theta\_k))^2 sin(theta\_k)\*sin(beta\_k)

-cos(theta\_k)\*sin(beta\_k) -sin(beta\_k)\*sin(theta\_k) cos(beta\_k)];

%This does not work when all lengths are equal

P\_k = 1/rho\_k\*[(1-cos(beta\_k))\*cos(theta\_k) (1-cos(beta\_k))\*sin(theta\_k) sin(beta\_k)]';

T\_k = [R\_k P\_k

0 0 0 1];

end

function dx = SMD(x,m,b,k,u)

dx(1,1) = x(2);

dx(2,1) = 1/m\*(-k\*x(1)-b\*x(2)+u);

end