MEC ENG 193B/292B: Feedback Control of Legged Robots

Homework 4

Professor Koushil Sreenath
UC Berkeley, Department of Mechanical Engineering
October 3, 2025

Larry Hui[1]
SID: 3037729658

[1]University of California at Berkeley, College of Engineering, Department of Mechanical Engineering. Author to whom any correspondence should be addressed. email: larryhui7@berkeley.edu

1 Problems

1.1 Two Link Downhill Walker

In this problem, we will setup a simulation for our first walking robot. We will use no control and perform the simulation of a passive dynamic walker, where the robot walks down a gentle slope - see Figure 1. To achieve this, do the following:

(a) Create `two_link_dynamics.m` file to be called by `ode45`. This file takes as input time and the state vector and computes the time-derivative of the state using the dynamical model given below. In particular, let the generalized coordinates, generalized velocities, and state vector be,

$$q = \begin{bmatrix} \theta \\ \phi \end{bmatrix}, \quad \dot{q} = \begin{bmatrix} \dot{\theta} \\ \dot{\phi} \end{bmatrix}, \quad x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}.$$

The continuous-time dynamics of the system is given by

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

where,

$$D(q) = \begin{bmatrix} 1 + 2\beta(1-\cos\phi) & -\beta(1-\cos\phi) \\ \beta(1-\cos\phi) & -\beta \end{bmatrix}, \quad C(q,\dot{q})\dot{q} = \begin{bmatrix} -\beta\sin\phi(\dot{\phi}^2 - 2\dot{\theta}\dot{\phi}) \\ \beta\dot{\theta}^2\sin\phi \end{bmatrix},$$

$$G(q) = \begin{bmatrix} \left(\frac{\beta g}{l}\right)\{\sin(\theta-\phi-\gamma) - \sin(\theta-\gamma)\} - \left(\frac{g}{l}\right)\sin(\theta-\gamma) \\ \left(\frac{\beta g}{l}\right)\sin(\theta-\phi-\gamma) \end{bmatrix}.$$

Assume $\beta = m/M = 0.01$, $\gamma = 0.01\,rad$, and $g/l = 1$.

Solution: Recall that in general we can write out continuous-time dynamics of the system as

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = B(q)u = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Using MATLAB, our function is written with the input t redacted since our dynamics are time-invariant

```
function dx = two_link_dynamics(t, x)

    beta = 0.01;
    gamma = 0.01;
    gl = 1;

    theta = x(1);  phi = x(2);  thdot = x(3);  phidot = x(4);

    D = [1 + 2*beta*(1-cos(phi)), -beta*(1-cos(phi));
         beta*(1-cos(phi)), -beta];

    Cqd = [-beta*sin(phi)*(phidot^2 - 2*thdot*phidot);
            beta*(thdot^2)*sin(phi)];

    Gq = [beta*gl*(sin(theta - phi - gamma) - sin(theta - gamma)) - gl*sin(
       theta - gamma);
          beta*gl*sin(theta - phi - gamma)];

    qdd = - D\(Cqd + Gq);

    dx = [thdot; phidot; qdd(1); qdd(2)];
end
```

(b) Create `two_link_impactdynamics.m` file to be called at an impact event. This file takes the pre-impact state $\mathbf{x}^-$ as input and outputs the post-impact state $\mathbf{x}^+$. We will do the re-labelling as part of this file so that the output state vector has the re-labelled generalized

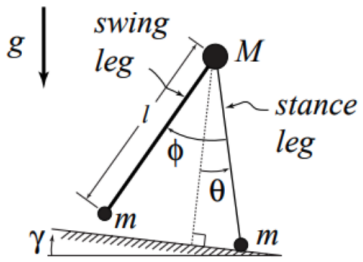


Figure 1: The Two-Link downhill Walker. The legs are symmetric with length l with a hip mass M and foot mass m. The ground slope is γ. The stance leg makes an angle θ with respect to the ground vertical and the swing leg makes an angle ϕ with respect to the stance leg

positions and velocities. We will assume the impact map with the re-labelling is given by the following transformation:

$$\begin{bmatrix} \theta \\ \phi \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix}^+ = \begin{bmatrix} -1 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 \\ 0 & 0 & \cos(2\theta) & 0 \\ 0 & 0 & \cos(2\theta)(1-\cos(2\theta)) & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \phi \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix}^-$$

Note that the above equation also reflects a change of names for the stance and swing legs (i.e., re-labeling is included.)

Solution: Just simply implementing this in code, we have

```
function x_plus = two_link_impactdynamics(x_minus)
    th = x_minus(1);
    A = [ -1,  0,  0, 0;
          -2,  0,  0, 0;
           0,  0,  cos(2*th), 0;
           0,  0,  cos(2*th)*(1 - cos(2*th)), 0 ];
    x_plus = A * x_minus;
end
```

(c) Create `two_link_event.m` file to be called by `ode45` to detect when an impact with the ground happens. We will ignore foot scuffing as well as ground penetration of the swing foot and define the collision to occur when the following geometric condition is met:

$$\phi(t) - 2\theta(t) = 0.$$

Write your function so that you only detect $-ve$ to $+ve$ crossings. Moreover, you will need to stop integration when the event is detected.

Solution: We just fill in the event definition from lecture (is terminal ensures integration is stopped when event is detected, and direction is +1 to allow us to detect -ve and +ve crossings)

```
function [value, isterminal, direction] = two_link_event(~, x)
    th = x(1); phi = x(2);
    value = phi - 2*th;
    isterminal = 1;
    direction = +1;
end
```

(d) Create `simulate_two_link_walker.m` that setups up a single step of walking simulation using all the functions you created above. You can use the following initial condition to start your simulation:

$$\mathbf{x}_0 = \begin{bmatrix} 0.2065 \\ 0.4130 \\ -0.2052 \\ -0.0172 \end{bmatrix}.$$

Once this works, modify this function to simulate for $N = 10$ steps. (You can use the provided `animate_two_link_walker.m` to animate your simulated data.)

Solution:

```
function [t_sol, x_sol, t_I] = simulate_two_link_walker(N)

    x0 = [0.2065; 0.4130; -0.2052; -0.0172];

    options = odeset('Events', @(t, x) two_link_event(t, x));
    x_sol= []; t_sol = []; t_I = [];
    t0 = 0;

    for i = 1:N
        [t, x] = ode45(@(t, x) two_link_dynamics(t, x), [t0, t0+10], x0,
           options);
        x_sol = [x_sol; x];
        t_sol = [t_sol; t];
        t_I(i) = find(t_sol == t_sol(end), 1);
        t0 = t_sol(end);
        x0 = two_link_impactdynamics(x(end, :));
    end
end
```

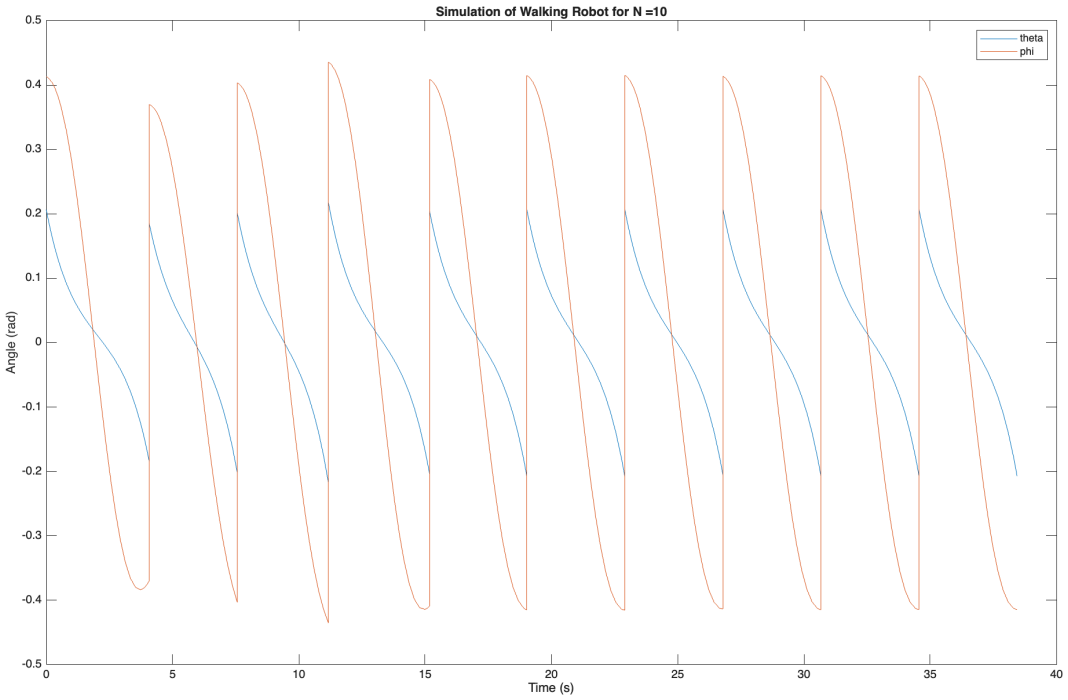
We used the following code to plot just ϕ and θ

```
[t_sol, x_sol, t_I] = simulate_two_link_walker(10);
th = x_sol(:,1); phi = x_sol(:,2);

figure;
plot(t_sol, th);
hold on;
plot(t_sol, phi);
xlabel('Time (s)'); ylabel('Angle (rad)');
title('Simulation of Walking Robot for N =10');
legend('theta', 'phi');

animate_two_link_walker(t_sol, x_sol, 0.01, t_I)
```

The resulting figure is shown on the next page with a couple pictures from the simulation



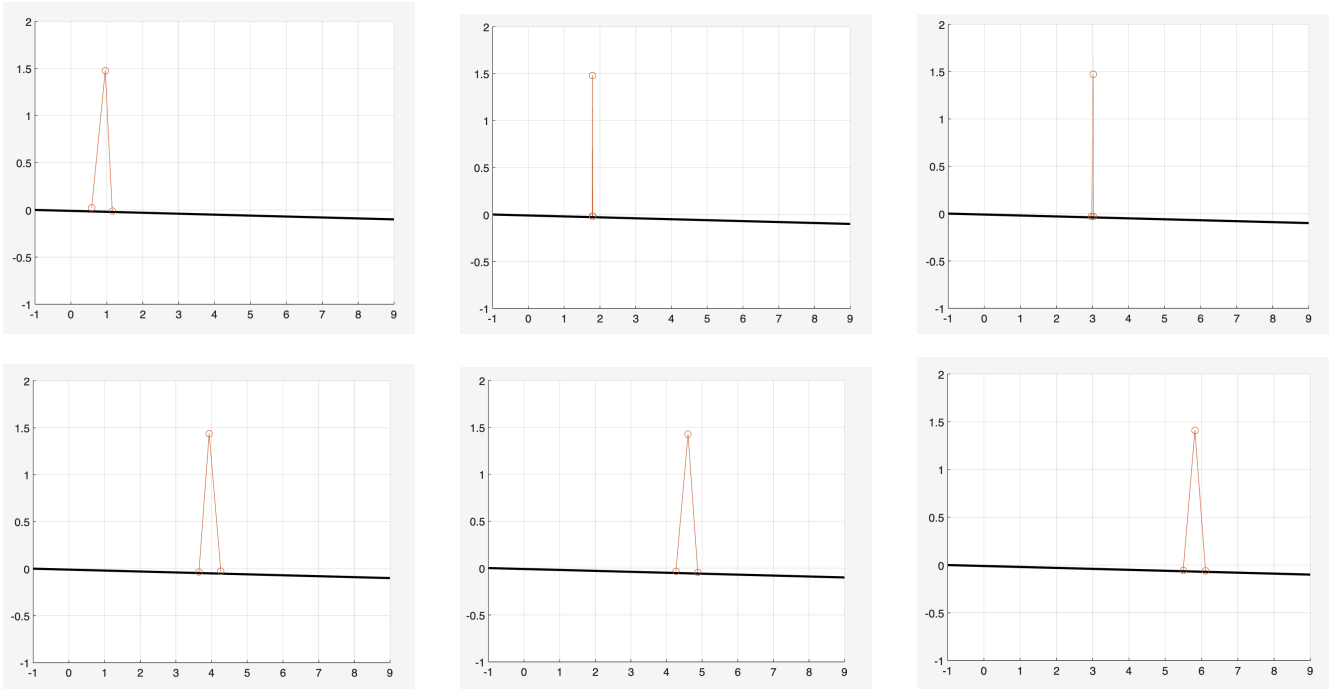


Figure 2: Simulation during 6 time steps

1.2 Two Link Downhill Walker Poincaré Map

We will numerically compute the Poincare map for the two link downhill walker system. Consider the Poincaré section defined by

$$\mathcal{S} := \{\mathbf{x} | \phi - 2\theta = 0\} \tag{1}$$

Write a MATLAB function `x1 = TwoLinkPoincare(x0)` that takes in a point `x0` on the Poincaré section and returns the point on the Poincaré section after one complete cycle (i.e. at the next intersection of the solution with the Poincaré section). Inside the function, you should change θ to ensure $\mathbf{x}_0 \in \mathcal{S}$. Compute the linearized Poincaré map about the fixed point given by $\mathbf{x}_0$. Comment on the stability of this periodic downhill walking gait.

Solution: Firstly, we need to enforce the condition by changing θ to ensure that $\mathbf{x}_0 \in \mathcal{S}$ so rearranging our constraint, we have $\theta = \phi/2$. The linearization for the Poincaré map, A, must hold in the following relationship $\delta x_{k+1} = A\delta x_k$ where $A = \frac{\partial P}{\partial x}\Big|_{x=x^*=x_0}$. Commenting on the stability of the system, we get the Jacobian or the linearized Poincaré map using a $\delta =$ `eps` to be

$$A = \begin{bmatrix} 0 & 0.5 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Since this matrix is upper triangular, the eigenvalues are just the diagonal entries (excluding 0, since that is from the constraint). Thus, the remaining eigenvalues are just $\lambda_i = 1$ this means that the periodic downhill walking gait can be considered marginally stable in the sense of Lyapunov, but not asymptotic nor exponentially stable. Again, intuitively, if we disturb the walker as it is transitioning to x_0, there are no actuators that can force the walker to x_0 again.

```
x0 = [0.2065; 0.4130; -0.2052; -0.0172];

function x1 = TwoLinkPoincare(x0)
    % change one of the coords of x0 to ensure x0 is in S
    x0(1) = x0(2)/2; % theta = phi/2
    x0(1) = x0(1) + eps;

    options = odeset('Events', @(t, x) two_link_event(t, x));

    % can get rid of t cuz time invariant
    [~, x_final] = ode45(@(t,x) two_link_dynamics(t,x), [0, 10], x0, options);
    x1 = x_final(end, :);
end

delta = eps;
e1 = [1; 0; 0; 0];
e2 = [0; 1; 0; 0];
e3 = [0; 0; 1; 0];
e4 = [0; 0; 0; 1];

J = zeros(4,4);

pos_peturb1 = TwoLinkPoincare(x0 + delta*e1);
pos_peturb2 = TwoLinkPoincare(x0 + delta*e2);
pos_peturb3 = TwoLinkPoincare(x0 + delta*e3);
pos_peturb4 = TwoLinkPoincare(x0 + delta*e4);

neg_peturb1 = TwoLinkPoincare(x0 - delta*e1);
neg_peturb2 = TwoLinkPoincare(x0 - delta*e2);
```

```
neg_peturb3 = TwoLinkPoincare(x0 - delta*e3);
neg_peturb4 = TwoLinkPoincare(x0 - delta*e4);

J(:, 1) = (pos_peturb1 - neg_peturb1)/(2*delta);
J(:, 2) = (pos_peturb2 - neg_peturb2)/(2*delta);
J(:, 3) = (pos_peturb3 - neg_peturb3)/(2*delta);
J(:, 4) = (pos_peturb4 - neg_peturb4)/(2*delta);

J
eig(J)
```