Biped Bootcamp: Model and Control of a Three-Link Biped Walker (First Edition)

Oluwami Dosunmu-Ogunbi, Robotics PhD Pre-Candidate

June 10, 2021

Acknowledgements

This technical document is a written adaptation of the "biped bootcamp" offered by Professor Jessy Grizzle. Additional consultation came from Margret Eva Mungai and Grant Gibson.

Contents

1 Introduction

Biped Robotics is a field within robotics that studies two-legged locomotion of robots. Applications in this field include search and rescue, rehabilitation exoskeletons, delivery services, automation of manufacturing processes, and more. While much research has been done in this field to control a vast array of complex biped robots, the novice researcher must first understand the fundamentals of how to model and control a simple biped robot before they can hope to move on to the more advanced topics. The purpose of this document is to provide an introduction to the field of biped robotics by explaining how to model and control a simple three-link biped robot.

Disclaimer: The way that I write my Matlab code is not necessarily the best/most efficient way to do it. I am learning as I write it, so there is definitely room for improvement. However, you may find it worthwhile following my approach since I add to my code as I introduce new content. It will be easier to understand how new things are implemented if your code is the same as mine. But! If you are confident in your coding ability, it would be a good challenge to follow your own path. Your choice!

Note: I am using Matlab 2018a on a PC running Windows 10 for my coding.

2 Deriving Lagrangian Model for 3-Link Biped

In this section, I will derive the Lagrangian equations of motion for the simple biped robot depicted in Figure 1. Note that for this simplified model, the masses are located at the center of each leg (O_1 and O_2), the hip joint (O_H), and at the end of the torso link (O_T). The length of each leg is r meaning that the mass of each leg is located at $\frac{r}{2}$. The distance between O_H and O_T is l.

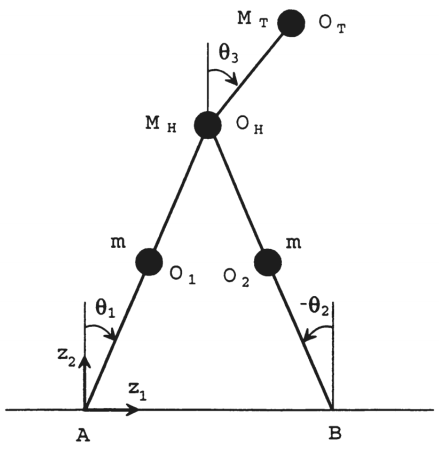

Figure 1: 3-Link Biped Robot

Lagrangian mechanics is one way to derive the equations of motion for a system.

Newtonian mechanics is another method to derive the equation of motion for a system and it uses Newton's Laws. Instead of Newton's laws, The Lagrangian uses energy to derive to equations of motion. "Overall, the Lagrangian has units of energy but no single expression for all systems. Any function which generates the correct equations of motion, in agreement with physical laws, can be taken as a Lagrangian. It is nevertheless possible to construct general expressions for large classes of applications. The non-relativistic Lagrangian for a system of particles can be defined by" [1]

$$L = KE - PE \tag{2.1}$$

where KE is kinetic energy and PE is potential energy. Kinetic energy can be calculated as follows:

$$KE = \frac{1}{2}m({v_x}^2 + {v_y}^2) \tag{2.2}$$

where m is mass, v_x is the velocity in the horizontal direction, and v_y is the velocity in the vertical direction. Potential energy can be calculated as follows:

$$PE = mgh \tag{2.3}$$

where m is mass, g is gravity, and h is height.

Eq. 2.1 does not help us much when it comes to determining a particle's relative motion. Thus, we take the partial derivative to get the Lagrange Equation in a more useable form:

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = \Gamma \tag{2.4}$$

where Γ is the external and internal forces, q is the generalized coordinates defined in this case by

$$q = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \tag{2.5}$$

and $\dot{q}$ is the generalized velocities defined in this case by

$$\dot{q} = \begin{bmatrix} \dot{\theta_1} \\ \dot{\theta_2} \\ \dot{\theta_3} \end{bmatrix} \tag{2.6}$$

This partial deferential form of the Lagrangian (Eq. 2.4) is known as the Euler-Lagrangian. We can rearrange this equation into a different form to make mathematical manipulation easier. This form is:

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \Gamma = Bu \tag{2.7}$$

where D is the mass inertial matrix, C is the centrifugal and coriolis forces matrix, G is the gravity vector, B is the input matrix, and u is the torques of the motors. Eq. 2.7 is known in some circles as the **robot equations**. More generally however, you will see these equations be referred to as the **equations of motion** (EoM). Careful. This equation may look simple and innocent, but when generating the equations of motion for, say, a five-link biped walker, writing out the equation in its entirety fills up over 10 pages using 12-point Times New Roman font. The equations of motion for the 40-variable Cassie biped (a bipedal robot developed by the company Agility Robotics) takes up *hundreds* of pages. In practice, you will generally have some sort of software develop the equations of motion for you, but since the purpose of this document is learning how to model and control a three-link biped, we are going to do most everything by hand to truly build up our understanding. Brace yourself!

How are the equations of motion from Eq. 2.7 generated? First, recognize that kinetic energy is a function of q and $\dot{q}$. That is, $KE(q,\dot{q})$. Potential energy is just a function of q. That is, $PE(q)$.

Let us define the variable p_i as the position vector for the mass at i where i =1,2,3,4, and

- $p_1 = p^{st}_{knee}$ refers to the position of the mass at the knee of the stance leg
- $p_2 = p_{hip}$ is the position of the mass at the hip
- $p_3 = p_{torso}$ is the position of the mass at the torso
- and $p_4 = p^{sw}_{knee}$ is the position of the mass at the knee of the swing leg.

When defining the position vectors for the masses of each link, they will be a function of the generalized coordinates q. Thus, the position of the mass p_i is defined as

$$p_i = \begin{bmatrix} p_x \\ p_y \end{bmatrix} \tag{2.8}$$

$$= f(q) \tag{2.9}$$

$$= f(\theta_1, \theta_2, \theta_3) \tag{2.10}$$

where p_x is the horizontal coordinates and p_y is the vertical coordinates. To find the velocity of the mass at this position, we use the chain rule to take the derivative:

$$v_i = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \tag{2.11}$$

$$= \frac{\partial f}{\partial q}\dot{q} \tag{2.12}$$

$$= \frac{\partial f}{\partial \theta_1}\dot{\theta}_1 + \frac{\partial f}{\partial \theta_2}\dot{\theta}_2 + \frac{\partial f}{\partial \theta_3}\dot{\theta}_3 \tag{2.13}$$

where v_x is the velocity in the horizontal direction and v_y is the velocity in the vertical direction.

NOTE: Here is a reminder on how to do transpose and matrix multiplication that will assist with the next step. For a matrix A where

$$A = \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.14}$$

and its transpose

$$A^T = \begin{bmatrix} x & y \end{bmatrix} \tag{2.15}$$

we know that

$$A^T A = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x^2 + y^2 \tag{2.16}$$

Another property that will be useful in the next step is the following:

$$(Ax)^T = x^T A^T \tag{2.17}$$

Thus, for example for a matrix A

$$(A^T A)^T = A^T (A^T)^T = A^T A \tag{2.18}$$

A website for computing matrix calculus can be found here (http://www.matrixcalculus.org/).

Using Eq. 2.2, we can write the kinetic energy for each mass to be

$$KE_i = \frac{1}{2}m(v_i)^T(v_i) \tag{2.19}$$

$$= \frac{1}{2}m\dot{q}^T\left(\frac{\partial f}{\partial q}\right)^T\left(\frac{\partial f}{\partial q}\right)\dot{q} \tag{2.20}$$

$$= \frac{1}{2}\dot{q}^T D(q)\dot{q} \tag{2.21}$$

where $D(q) = \left[m\left(\frac{\partial f}{\partial q}\right)^T\left(\frac{\partial f}{\partial q}\right)\right]$ and is a symmetric matrix since it is the matrix multiplication of a matrix and its transpose.

The total kinetic energy of the robot would simply be the sum of the kinetic energies of each of the masses,

$$KE = \sum_{i=1}^{4}\frac{1}{2}\dot{q}^T\underbrace{\left[m\left(\frac{\partial f}{\partial q}\right)^T\left(\frac{\partial f}{\partial q}\right)\right]}_{D_i(q)}\dot{q} \tag{2.22}$$

Plug in Eq. 2.1 into Eq. 2.4 and you get

$$\frac{d}{dt}\frac{\partial(KE-PE)}{\partial\dot{q}} - \frac{\partial(KE-PE)}{\partial q} = \Gamma \tag{2.23}$$

For the next step, you will need to know this property of taking a derivative of a vector. For a vector x and matrix M,

$$\frac{\partial}{\partial x}x^T Mx = 2Mx \tag{2.24}$$

Using our KE from Eq. 2.21 we can take the partial derivatives to get

$$\frac{d}{dt}\left[\frac{\partial}{\partial\dot{q}}\left(\frac{1}{2}\dot{q}^T D(q)\dot{q}\right) - \cancelto{0}{\frac{\partial[PE]}{\partial\dot{q}}}\right] - \frac{\partial}{\partial q}\left[\frac{1}{2}\dot{q}^T D(q)\dot{q} - PE\right] = \Gamma \tag{2.25}$$

$$\frac{d}{dt}\left[\frac{1}{\cancel{2}}(\cancel{2})D(q)\dot{q} - 0\right] - \frac{\partial}{\partial q}\left[\frac{1}{2}\dot{q}^T D(q)\dot{q} - PE\right] = \Gamma \tag{2.26}$$

$$\Bigg[D(q)\ddot{q} + \underbrace{\frac{\partial}{\partial q}[D(q)\dot{q}]\dot{q}\Bigg] - \Bigg[\frac{1}{2}\frac{\partial}{\partial q}[\dot{q}^T D(q)\dot{q}]}_{=C(q,\dot{q})\dot{q}} - \underbrace{\frac{\partial[PE]}{\partial q}}_{=G(q)}\Bigg] = \Gamma \tag{2.27}$$

NOTE: These calculations give you the product $C(q,\dot{q})\dot{q}$ not the matrix $C(q,\dot{q})$. The reason why I am not defining $C(q,\dot{q})$ here is because $C(q,\dot{q})$ is NOT unique whereas $C(q,\dot{q})\dot{q}$ is unique. For example, I can define the following 2 different C matrices, C_1 and C_2, and arrive at the same $C(q,\dot{q})\dot{q}$

$$\underbrace{\begin{bmatrix} \dot{q}_2 & 0 \\ \dot{q}_1\dot{q}_2 & 0 \end{bmatrix}}_{C_1(q,\dot{q})} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & \dot{q}_1 \\ 0 & (\dot{q}_1)^2 \end{bmatrix}}_{C_2(q,\dot{q})} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = \underbrace{\begin{bmatrix} \dot{q}_1\dot{q}_2 \\ (\dot{q}_1)^2\dot{q}_2 \end{bmatrix}}_{C(q,\dot{q})\dot{q}} \tag{2.28}$$

Therefore it is not necessary to define a $C(q,\dot{q})$, just leave it as $C(q,\dot{q})\dot{q}$.

NOTE: In Eq. 2.27 the centrifugal and Coriolis components of the $C(q,\dot{q})\dot{q}$ product is as follows:

$$C(q,\dot{q})\dot{q} = \underbrace{\frac{\partial}{\partial q}[D(q)\dot{q}]\dot{q}}_{\text{Coriolis}} - \underbrace{\frac{1}{2}\frac{\partial}{\partial q}[\dot{q}^T D(q)\dot{q}]}_{\text{centrifugal}} \tag{2.29}$$

2.1 Kinematics of the Robot (Position Information)

The first step is to determine the position information for the points of interest, namely A, B, O_1, O_2, O_H, and O_T. First we must establish an origin. You can locate the origin wherever you like (heck, you can even put it in like somewhere in Antarctica if you feel so inclined), but for simplicity, let's put the origin at the foot of the stance leg, A. Thus, the position of the foot on the stance leg is

$$p^{st}_{foot} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{2.30}$$

Using geometry, you can determine the position of each joint. The position of the knee on the stance leg is

$$p^{st}_{knee} = p^{st}_{foot} + \begin{bmatrix} \frac{r}{2}sin(\theta_1) \\ \frac{r}{2}cos(\theta_1) \end{bmatrix} \tag{2.31}$$

The position of the hip is

$$p_{hip} = p^{st}_{foot} + \begin{bmatrix} rsin(\theta_1) \\ rcos(\theta_1) \end{bmatrix} \tag{2.32}$$

The position of the torso is

$$p_{torso} = p_{hip} + \begin{bmatrix} lsin(\theta_3) \\ lcos(\theta_3) \end{bmatrix} \tag{2.33}$$

The position of the knee on the swing leg is

$$p^{sw}_{knee} = p_{hip} + \begin{bmatrix} -\frac{r}{2}sin(\theta_2) \\ -\frac{r}{2}cos(\theta_2) \end{bmatrix} \tag{2.34}$$

The position of the swing foot is

$$p^{sw}_{foot} = p_{hip} + \begin{bmatrix} -rsin(\theta_2) \\ -rcos(\theta_2) \end{bmatrix} \tag{2.35}$$

2.2 Calculate Potential Energy

Recall that the potential energy can be calculated using Eq. 2.3. To find the total potential energy of the robot, add the potential energies at each point of the robot. Since the masses of the robots are located at fixed points (as opposed to being distributed across each link), this is simple–you would find the potential energy at each mass and sum those. Thus, the potential energy of the robot is

$$PE = mg\left[\frac{r}{2}(cos(\theta_1)\right] + M_H g\left[rcos(\theta_1)\right] + M_T g\left[rcos(\theta_1) + lcos(\theta_3)\right] + mg\left[rcos(\theta_1) - \frac{r}{2}cos(\theta_2)\right] \tag{2.36}$$

2.3 Calculate Kinetic Energy

Recall that the kinetic energy can be calculated using Eq. 2.2. To find the total kinetic energy of the robot, we need to first find the velocities of each of the masses. We do this by taking the derivative of the position matrices that we found in Section 2.1.

The velocity of the knee on the stance leg is

$$v^{st}_{knee} = \frac{d}{dt}p^{st}_{knee} = \begin{bmatrix} \frac{r}{2}cos(\theta_1)\dot{\theta_1} \\ \frac{r}{2}(-sin(\theta_1))\dot{\theta_1} \end{bmatrix} \tag{2.37}$$

Now we can use this velocity to plug into Eq. 2.2 to find the kinetic energy of the stance knee.

$$KE^{st}_{knee} = \frac{1}{2}m\left[\left(\frac{r}{2}\right)^2(\dot{\theta_1})^2\left(cos^2(\theta_1) + (-sin^2(\theta_1))\right)\right] \tag{2.38}$$

Using the trig identity $cos^2(x) + sin^2(x) = 1$, we can simplify this to

$$KE^{st}_{knee} = \frac{1}{8}mr^2(\dot{\theta_1})^2 \tag{2.39}$$

The velocity of the hip is

$$v_{hip} = \frac{d}{dt}p_{hip} = \begin{bmatrix} rcos(\theta_1)\dot{\theta_1} \\ r(-sin(\theta_1))\dot{\theta_1} \end{bmatrix} \tag{2.40}$$

Thus the kinetic energy of the hip is

$$KE_{hip} = \frac{1}{2}M_H\left[\left(rcos(\theta_1)\dot{\theta_1}\right)^2 + \left(r(-sin(\theta_1))\dot{\theta_1}\right)^2\right] = \frac{1}{2}r^2M_H(\dot{\theta_1})^2 \tag{2.41}$$

The velocity of the torso is

$$v_{torso} = v_{hip} + \frac{d}{dt}\begin{bmatrix} lsin(\theta_3) \\ lcos(\theta_3) \end{bmatrix} = \begin{bmatrix} rcos(\theta_1)\dot{\theta_1} \\ -rsin(\theta_1)\dot{\theta_1} \end{bmatrix} + \begin{bmatrix} lcos(\theta_3)\dot{\theta_3} \\ -lsin(\theta_3)\dot{\theta_3} \end{bmatrix} \tag{2.42}$$

Thus the kinetic energy of the torso is

$$\begin{aligned} KE_{torso} &= \frac{1}{2}M_T\left[\left(rcos(\theta_1)\dot{\theta_1} + lcos(\theta_3)\dot{\theta_3}\right)^2 + \left(-rsin(\theta_1)\dot{\theta_1} - lsin(\theta_3)\dot{\theta_3}\right)^2\right] \\ &= \frac{1}{2}M_T\left[r^2\cancel{cos^2(\theta_1)}\dot{\theta_1}^2 + 2rlcos(\theta_1)cos(\theta_3)\dot{\theta_1}\dot{\theta_3} + l^2\cancel{cos^2(\theta_3)}\dot{\theta_3}^2 + \right. \\ &\qquad \left. r^2\cancel{sin^2(\theta_1)}\dot{\theta_1}^2 + 2rlsin(\theta_1)sin(\theta_3)\dot{\theta_1}\dot{\theta_3} + l^2\cancel{sin^2(\theta_3)}\dot{\theta_3}^2\right] \\ &= \frac{1}{2}M_T\left[r^2(\dot{\theta_1})^2 + l^2(\dot{\theta_3})^2 + 2rl\left[cos(\theta_1)cos(\theta_3) + sin(\theta_1)sin(\theta_3)\right]\dot{\theta_1}\dot{\theta_3}\right] \end{aligned} \tag{2.43}$$

The velocity of the knee on the swing leg is

$$v^{sw}_{knee} = v_{hip} + \frac{d}{dt}\begin{bmatrix} -\frac{r}{2}sin(\theta_2) \\ -\frac{r}{2}cos(\theta_2) \end{bmatrix} = \begin{bmatrix} rcos(\theta_1)\dot{\theta_1} \\ -rsin(\theta_1)\dot{\theta_1} \end{bmatrix} + \begin{bmatrix} -\frac{r}{2}cos(\theta_2)\dot{\theta_2} \\ \frac{r}{2}sin(\theta_2)\dot{\theta_2} \end{bmatrix} \tag{2.44}$$

Thus the kinetic energy of the knee on the swing leg is

$$\begin{aligned} KE^{sw}_{knee} &= \frac{1}{2}m\left[\left(rcos(\theta_1)\dot{\theta_1} - \frac{r}{2}cos(\theta_2)\dot{\theta_2}\right)^2 + \left(-rsin(\theta_1)\dot{\theta_1} + \frac{r}{2}sin(\theta_2)\dot{\theta_2}\right)^2\right] \\ &= \frac{1}{2}m\left[r^2\cancel{cos^2(\theta_1)}\dot{\theta_1}^2 - r^2cos(\theta_1)cos(\theta_2)\dot{\theta_1}\dot{\theta_2} + \frac{r^2}{4}\cancel{cos^2(\theta_2)}\dot{\theta_2}^2 \right. \\ &\qquad \left. + r^2\cancel{sin^2(\theta_1)}\dot{\theta_1}^2 - r^2sin(\theta_1)sin(\theta_2)\dot{\theta_1}\dot{\theta_2} + \frac{r^2}{4}\cancel{sin^2(\theta_2)}\dot{\theta_2}^2\right] \\ &= \frac{1}{2}m\left[r^2(\dot{\theta_1})^2 + \frac{r^2}{4}(\dot{\theta_2})^2 - r^2\left(cos(\theta_1)cos(\theta_2) + sin(\theta_1)sin(\theta_2)\right)\dot{\theta_1}\dot{\theta_2}\right] \end{aligned} \tag{2.45}$$

The total kinetic energy of the robot can be calculated by summing all of the kinetic energies at each location.

$$KE = KE_{knee}^{st} + KE_{hip} + KE_{torso} + KE_{knee}^{sw} \tag{2.46}$$

2.4 Define Matrices for the Equations of Motion using Matlab

Now that we have found the potential energy (Eq. 2.36) and kinetic energy (Eq. 2.46) of the robot, we can use these values to plug into the robot equations (Eq. 2.7) to define the matrices.

Because I do not hate myself, I will not be doing this by hand and you should not either. That is what Matlab is for! As a reminder, I am using Matlab 2018a for my coding.

Let's create a new script called `SymbolicModelForThreeLinkRobot.m`. We will use this script to find the symbolic notation of the matrices for the robot equations. Since we are deriving the matrices symbolically, our first step is to create the symbolic variables that we will be using to define the matrices. We use the `syms` command for this purpose. For q and $\dot{q}$, we need to make vector variables that will contain variables for $\theta_1, \theta_2, \theta_3, \dot{\theta_1}, \dot{\theta_2}$ and $\dot{\theta_3}$.

NOTE: In Matlab, I denote "dot" with a "d". For example, in Matlab I define $\dot{q}$ as `dq`.

Defining the symbolic variables needed in Matlab will look something like this

```
syms r l m M_H M_T g
syms th1 th2 th3
syms dth1 dth2 dth3

q = [th1; th2; th3];
dq = [dth1; dth2; dth3];
```

Next, we define the position vectors for each of the masses using the expressions that we derived in Section 2.1 Kinematics of the Robot (Position Information). The code will look something like this

```
p_st_foot = [0; 0];
p_st_knee = p_st_foot + [r/2*sin(th1); r/2*cos(th1)];
p_hip = p_st_foot + [r*sin(th1); r*cos(th1)];
p_torso = p_hip + [l*sin(th3); l*cos(th3)];
p_sw_knee = p_hip + [-r/2*sin(th2); -r/2*cos(th2)];
p_sw_foot = p_hip + [-r*sin(th2); -r*cos(th2)];
```

Next, we define the corresponding velocity vectors for each of the masses. Velocity is simply the derivative with respect to time of the position vectors. As I showed in Section 2.3 Calculate Kinetic Energy, the chain rule had to be used here. In Matlab, we can use the `jacobian` function to make this calculation. Thus for each position vector $f(q)$,

$$\frac{d}{dt}f(q) = \frac{\partial f(q)}{\partial q}\dot{q} = \texttt{jacobian(f,q)*dq} \tag{2.47}$$

Thus, the Matlab code for finding the velocity vectors will look something like this

```
v_st_knee = jacobian(p_st_knee,q)*dq;
v_hip = jacobian(p_hip,q)*dq;
v_torso = jacobian(p_torso,q)*dq;
v_sw_knee = jacobian(p_sw_knee,q)*dq;
v_sw_foot = jacobian(p_sw_foot,q)*dq;
```

Next, we define the kinetic energy at each mass by directly using Eq. 2.19. Use the Matlab function `simplify` to have Matlab simplify the result. Once the kinetic energy of each of the masses have been defined, sum them all together to get the total kinetic energy of the system. The code will look something like this

```
KE_st_knee = simplify((1/2)*m*v_st_knee.'*v_st_knee);
KE_hip = simplify((1/2)*M_H*v_hip.'*v_hip);
KE_torso = simplify((1/2)*M_T*v_torso.'*v_torso);
KE_sw_knee = simplify((1/2)*m*v_sw_knee.'*v_sw_knee);


KE = KE_st_knee + KE_hip + KE_torso + KE_sw_knee;
KE = simplify(KE);
```

Next, we define the potential energy of the system by directly using the equation found in Section 2.2 Calculate Potential Energy. The code will look something like this

```
PE = m*g*p_st_knee(2) + M_H*g*p_hip(2) + M_T*g*p_torso(2) + m*g*p_sw_knee(2);
PE = simplify(PE);
```

Now you have all the variables you need to find the matrices of the robot equations. Use the matrix definitions from Eq. 2.27, recalling that $D(q) = \left[m\left(\frac{\partial f}{\partial q}\right)^T\left(\frac{\partial f}{\partial q}\right)\right]$.

As discussed in the introduction to Section 2: Deriving Lagrangian Model for 3-Link Biped, the C matrix is not unique. In my code, I used the C matrix definition that was provided by Professor Jessy Grizzle. He also used a different method to derive the D matrix, but you will find that you will get the same result regardless of if you use his method or the D definition that I derived in this report.

The code to generate the G vector and the D and C matrices are as follows:

```
G=jacobian(PE,q).';
G=simplify(G)
D=simplify(jacobian(KE,dq).');
D=simplify(jacobian(D,dq))

syms C real
n=max(size(q));
for k=1:n
    for j=1:n
        C(k,j)=0*g;
        for i=1:n
            C(k,j)=C(k,j)+(1/2)*(diff(D(k,j),q(i))+ ...
            diff(D(k,i),q(j))-diff(D(i,j),q(k)))*dq(i);
        end
    end
end
C=simplify(C)
```

You may check your result for the symbolic representation of the matrices values in Appendix A of [3].

3 Controlling the 3-link Robot

In this section, we are going to control the 3-link walker that we defined in the previous section.

3.1 Defining the Input Matrix B

The first objective is to define the input matrix B from Eq. 2.7. The input matrix tells us where to apply our torques u (that is why they are multiplied together).

Our input matrix will be $3x2$ and is essentially a combination of two vectors, one for each motor, that define which link angles that respective motor influences. The first column is the angles that the motor attached to the stance leg influences, and the second column is the angles that the motor attached to the swing leg influences.

To determine the motor's effect on a leg, consider the **relative angle** of each leg. The relative angle of each leg is the angle between each respective leg and the torso (because each motor is placed between the leg and the torso). To help visualize this, consider Figure 2. Each leg is controlled by a motor (represented in blue in the figure) where the respective leg is attached to the motor and the torso link is attached to a connector that binds the two motors.

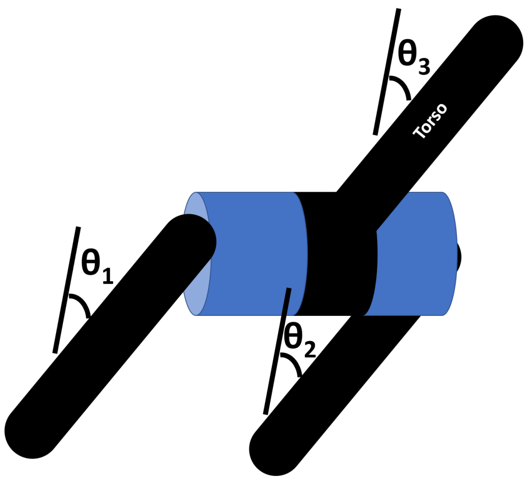

Figure 2: 3-Link Biped Robot Highlighting Motors on Each Link

Let us define α_{rel} to be the relative angle between the torso and the stance leg as shown in Figure 3.

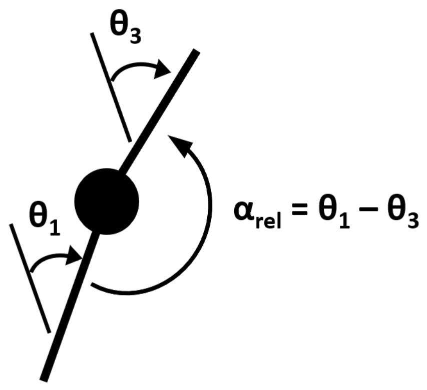

Figure 3: Relative angle between torso and stance leg, α_{rel}

Thus,

$$\alpha_{rel} = \theta_1 - \theta_3 \tag{3.1}$$

Similarly, we define β_{rel} to be the relative angle between the torso and the swing leg.

$$\beta_{rel} = \theta_2 - \theta_3 \tag{3.2}$$

Taking the partial derivative of α_{rel} and β_{rel} with respect to the generalized coordinates q (Eq. 2.5) (since we care about the torque influence at each angle, not the position influence at each angle), we get

$$\frac{\partial \alpha_{rel}}{\partial q} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \tag{3.3}$$

$$\frac{\partial \beta_{rel}}{\partial q} = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \tag{3.4}$$

We combine these two "input vectors" to form our input matrix

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \tag{3.5}$$

Since our system is under-actuated, meaning that there are less motors than there are movable joints, we see that each of our motors simultaneously controls two angles at once.

Recall from Eq. 2.7 that u is the torques of the motors

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{3.6}$$

where u_1 is the torque of the motor for the stance leg and u_2 is the torque of the motor for the swing leg.

3.2 Feedback Linearization

We will use **feedback linearization** control to control our 3-link walker system. For more information on this method, please refer to [7].

First, we must define our state variables in a vector called x. We will use our generalized coordinates q and its derivative $\dot{q}$ for this purpose.

$$x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{3.7}$$

Therefore, the derivative of the state vector is

$$\frac{d}{dt}x = \dot{x} = \begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} \tag{3.8}$$

From Eq. 3.7, we know that $\dot{q} = x_2$. But what is $\ddot{q}$? We can find $\ddot{q}$ by looking at the robot equations from Eq. 2.7. Subtracting $C(q,\dot{q})$ and $G(q)$ from both sides and left-multiplying everything by $D^{-1}(q)$, we find

$$\ddot{q} = -D^{-1}(q)C(q,\dot{q})\dot{q} - D^{-1}(q)G(q) + D^{-1}(q)Bu \tag{3.9}$$

Therefore

$$\dot{x} = \begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -D^{-1}(x_1)C(x_1,x_2)x_2 - D^{-1}(x_1)G(x_1) + D^{-1}(x_1)Bu \end{bmatrix} \tag{3.10}$$

Let's define $f(x)$ and $g(x)$ such that

$$f(x) = \begin{bmatrix} x_2 \\ -D^{-1}(x_1)C(x_1,x_2)x_2 - D^{-1}(x_1)G(x_1) \end{bmatrix} \tag{3.11}$$

and

$$g(x) = \begin{bmatrix} 0 \\ D^{-1}(x_1)B \end{bmatrix} \tag{3.12}$$

Therefore,

$$\dot{x} = f(x) + g(x)u \tag{3.13}$$

When an equation is in the form $y = mx + b$, this is called the **control afline form** meaning the equation is linear with respect to x.
So our equation $\dot{x} = f(x) + g(x)u$ is in control afline form because it is linear with respect to u.

We will define our output function to be

$$y = h(x_1) \tag{3.14}$$

so that the output only depends on x_1, the generalized positions.
Let's define our $h(x_1)$ to be

$$h(x_1) = h(q) = \begin{bmatrix} \theta_3 - \theta_3^d \\ \theta_1 + \theta_2 \end{bmatrix} \tag{3.15}$$

What does this $h(x)$ mean physically? We are imposing a walking objective that is $\theta_1 + \theta_2 = 0$ which means that the angle of the stance leg and the angle of the swing leg are always equal and opposite. To meet this walking objective, we need two constraints, $y_1 = \theta_3 - \theta_3^d$ and $y_2 = \theta_1 + \theta_2$. Using y_1 imposes the constraint of trying to keep the torso at a desired position (that is, keeping the angle of the torso link θ_3 at a desired angle θ_3^d). And y_2 helps us keep the angle of the stance and swing legs equal and opposite.

In order to build a controller, we need to keep taking the derivative of our output function until our inputs u pop up. The number of derivatives needed to get the inputs to pop up is known as the **relative degree** of the system.

So let's take the first derivative:

$$\dot{y} = \frac{\partial h}{\partial x_1}\dot{x}_1 \tag{3.16}$$

Simplifying this expression will show that the term with u cancels out. Thus, we take the second derivative and get

$$\ddot{y} = \frac{\partial h}{\partial x_1}\ddot{x}_1 + \left(\frac{\partial}{\partial x_1}\left(\frac{\partial h}{\partial x_1}\dot{x}_1\right)\right)\dot{x}_1 \tag{3.17}$$

Simplifying here shows that we retain our term with u. So the relative degree of this system is 2.

Wait, you don't believe me? Too much hand waving? Yeah...it turns out that taking the derivatives as is will get super messy super fast. So to avoid all of that mess, let me introduce you to Lie derivatives so that our notation does not look like a mathematician took a dump on a piece of paper.

Lie Derivatives

NOTE: Lie = "Lee," as in the mathmatician Sophus Lie.

Consider the generic differential equation

$$\dot{x} = X(x) \tag{3.18}$$

where

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix} \tag{3.19}$$

$X(x)$ can be $f(x)$ or $g(x)$ (from Eq. 3.13) or any right hand side of an ODE.

So for a generic function $y(x) = h(x)$, we can define its derivative $\dot{y}$ as

$$\dot{y} = \frac{\partial h}{\partial x}\dot{x} = \frac{\partial h}{\partial x}X(x) \tag{3.20}$$

Which is the Lie derivative. To state more explicitly, the Lie derivative (L, not to be confused with the Lagrangian which I also defined as L in a previous section) of $h(x)$ with respect to $X(x)$ is

$$\boxed{L_X h(x) := \frac{\partial h}{\partial x}X(x) \qquad (3.21)}$$

So for our control system

$$\dot{x} = f(x) + g(x)u$$
$$y = h(x)$$

we calculate the first derivative of our output function to be

$$\begin{aligned}
\dot{y} &= \frac{\partial h(x)}{\partial x}\dot{x} && (3.22)\\
&= \frac{\partial h(x)}{\partial x}\left[f(x) + g(x)u\right] && (3.23)\\
&= \frac{\partial h(x)}{\partial x}f(x) + \frac{\partial h(x)}{\partial x}g(x)u && (3.24)\\
&= L_f h(x) + \left[L_g h(x)\right]u && (3.25)
\end{aligned}$$

where L_f is the Lie derivative with respect to f and L_g is the Lie derivative with respect to g.

Given Eq. 3.15, we can know that

$$L_g h(x) = 0 \tag{3.26}$$

We can confirm this in Matlab.

$$L_g h(x) = \underbrace{\texttt{jacobian(h,x)*g(x)}}_{\left[\frac{\partial h(x)}{\partial x}\right]g(x)} = 0$$

where

$$x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}$$

Thus Eq. 3.25 becomes

$$\dot{y} = L_f h(x) + \cancelto{0}{\left[L_g h(x)\right]}u = \text{some function of x} \tag{3.27}$$

So our control inputs u do not show up in the first derivative. So let's take the second derivative using chain rule

$$\begin{aligned}
\ddot{y} &= \frac{\partial}{\partial x}[L_f h(x)]\dot{x} && (3.28)\\
&= \frac{\partial}{\partial x}[L_f h(x)]\big(f(x)+g(x)u\big) && (3.29)\\
&= L_f[L_f h(x)] + L_g[L_f h(x)]u && (3.30)\\
&= L_f^2 h(x) + \underbrace{L_g L_f h(x)}_{\text{decoupling matrix}} u && (3.31)
\end{aligned}$$

Aha! There's u!

Now we can build a controller. Let's design a PD controller with controller variables K_p and K_d. The objective is to build a Second Order system such that

$$\begin{aligned}
\ddot{y} + K_d\dot{y} + K_p y = 0 && (3.32)\\
\Rightarrow \ddot{y} = -K_d\dot{y} + K_p y && (3.33)
\end{aligned}$$

Therefore, combining Eq. 3.31 and Eq. 3.33 we get

$$L_f^2 h(x) + L_g L_f h(x)u = -K_d\dot{y} - K_p y \tag{3.34}$$

From Eq. 3.14 and Eq. 3.27, we know that $\dot{y} = L_f h(x)$ and $y = h(x)$. Thus Eq. 3.34 becomes,

$$L_f^2 h(x) + L_g L_f h(x)u = -K_d L_f h(x) - K_p h(x) \tag{3.35}$$

Let's solve for our controller, u, which is the torque values that needs to be delivered to each motor. First, we subtract $L_f^2 h(x)$ from both sides, then we take the inverse of $L_g L_f h(x)$ on both sides:

$$\begin{aligned}
\Rightarrow L_g L_f h(x)u &= -L_f^2 h(x) - K_d L_f h(x) - K_p h(x) && (3.36)\\
u &= \big(L_g L_f h(x)\big)^{-1}\big(-L_f^2 h(x) - K_d L_f h(x) - K_p h(x)\big) && (3.37)
\end{aligned}$$

Now, how do we determine values for K_p and K_d? One suggestion is to look at the closed loop transfer function $H(s)$ for a 2nd order system:

$$H(s) = \frac{\omega_n^2}{s^2 + \underbrace{2\zeta\omega_n}_{K_d} s + \underbrace{\omega_n^2}_{K_p}} \tag{3.38}$$

Thus to reiterate,

$$K_p = \omega_n^2 \tag{3.39}$$

$$K_d = 2\zeta\omega_n \tag{3.40}$$

You may choose to have a damping ratio ζ to make the system critically damped ($\zeta = 1$) or underdamped ($\zeta < 1$). But as far as the natural frequency ω_n goes, you will have to use trial and error. An idea to assist with this trial and error is trying to impose other constraints such as a 5% settling time t_s where we can choose to make the settling time approximately 0.3 seconds

$$t_s = \frac{3}{\zeta\omega_n} \approx 0.3 \text{ seconds} \tag{3.41}$$

or trying to match the step time of the Cassie robot. Cassie has a step time of approximately 0.6 seconds.

3.3 Deriving the Control Functions Symbolically in Matlab

Great! We know the expressions to use for the controller. Now let's get Matlab to derive those expressions symbolically.

Open up your `SymbolicModelForThreeLinkRobot.m` script that we used to find our G vector and D and C matrices. We will be appending the code to derive the control functions to the end.

First, we use `syms` to create the new variables that we will be working with. We also know what our matrix B is from Eq. 3.5. So the inital lines of code will look something like this

```
syms x1 x2 f g th3desired u1 u2

u = [u1; u2];

B = [1 0;0 1;-1 -1];
x1 = q;
x2 = dq;
x = [x1; x2];
```

Next, we define our functions for f, g and h which we know from Eq. 3.11, Eq. 3.12, and Eq. 3.15.

```
f = [x2; -inv(D)*C*x2 - inv(D)*G];
g = [zeros(3,2); inv(D)*B];
h = [th3-th3desired; th1+th2];
```

Note that for g we needed to use a matrix of 0's to ensure that our vector dimensions were consistent. Using the command `size(inv(D)*B)` – which is the size of the other matrix that we are using to define g – we learned that the size of this zero matrix needed to be $3x2$.

Finally, we can perform our Lie derivative calculations to ultimately come up with $\ddot{y}$.

```
Lgh = jacobian(h,x)*g
Lfh = jacobian(h,x)*f

L2fh = jacobian(Lfh,x)*f
LgLfh = jacobian(Lfh,x)*g

ddy = L2fh + LgLfh*u
```

4 Taking the First Step! (in Matlab Simulation)

So you made it this far, eh? Congratulations! You now have all you need to make a visual simulation in Matlab to get a 3-link biped robot to walk! Well...you know enough to make it take a step. Well...you know enough to make the motion of a step *just before* the swing leg touches the ground. But do not worry; we are getting there!

Generating the Matlab code to get our 3-link biped to take one step is going to require multiple functions and therefore multiple Matlab files. Here is the list of names of the Matlab files we are going to be using to get our biped to take a step:

- `SymbolicModelForThreeLinkRobot.m` – Used to generate the symbolic model for the biped.
- `ThreeLinkWalker_sim.m` – Top level code to actually run the simulation. Define initial conditions here.
- `ThreeLinkWalker_anim.m` – Animates the lines to show the biped moving.
- `ThreeLinkWalker_ODE45.m` – Solves the ODE to generate the outputs that define the position (θ_i) and velocity ($\dot{\theta}_i$) values over time for the links of the biped.
- `Points_ThreeLinkWalker.m` – Translates the generated θ_i and $\dot{\theta}_i$ values into point positions for the ends of each link.
- `model_params_stiff_legs.m` – Defines the model parameters for the biped (e.g. the actual the mass and length values, value of gravity, and initial position of the stance leg).

- `dyn_mod_ThreeLinkWalker.m` – Calculates the numerical values for each of the matrices for the robot equations.
- `ControlFunctions.m` – Calculates the numerical values for the functions used to define the controller u.
- `ThreeLinkEndStepEvents.m` – Creates the event function used to signify the end of a step.

4.1 Define Parameters and Variables

4.1.1 `SymbolicModelForThreeLinkRobot.m`

Before we can even start thinking about making our biped walk, we need to define the many parameters and variables that make up its system. Luckily, we already knocked out one of the major tasks for this step when we generated the code for `SymbolicModelForThreeLinkRobot.m` in Section 2.4 Define Matrices for the Equations of Motion using Matlab and in Section 3.3 Deriving the Control Functions Symbolically in Matlab. Nothing further needs to be added to this code.

4.1.2 `model_params_stiff_legs.m`

Next, let's look at `model_params_stiff_legs.m`. This file gives numerical values to the variables that we set up in `SymbolicModelForThreeLinkRobot.m`. We will make this Matlab script a function so that we can call on it in other Matlab functions and scripts that need it to get the numerical values of the variables. We have a separate function dedicated to giving values to the variables so that we can consolidate where we would need to change up variable values into a single location. In this code, I am also defining the initial position of the stance leg to be at the origin $(0,0)$. These variable values were provided to me by Professor Jessy Grizzle.

```
function [r,m,M_H,M_T,l,g,p_st_foot,Mtotal]=model_params_stiff_legs

r=1;   % length of a leg
m=5;   % mass of a leg
M_H=15; % mass of hips
M_T=10; % mass of torso
l=0.5; % distance between hips and torso
g=9.8; % acceleration due to gravity

Mtotal = m*2+M_H+M_T; % total mass of 3 link walker
```

```
% Define initial position of stance leg
p_stx_foot_o = 0; % x value
p_sty_foot_o = 0; % y value
p_st_foot = [p_stx_foot_o ; p_sty_foot_o]; % vector contain x and y values
```

4.1.3 `dyn_mod_ThreeLinkWalker.m`

The next Matlab file that I want to look at is `dyn_mod_ThreeLinkWalker.m`. This code will be another Matlab function. It will take the input arguments of `q,dq,parameters` where `q` is the position vector containing the numerical values for each of the θ values, `dq` is the velocity vector containing the numerical values for each of the $\dot{\theta}$ values, and `parameters` is the vector of all the parameters defined in `model_params_stiff_legs.m` (and also listed in the same order). The outputs of this function will be the numerical values for the D,C, and B matrices and the G vector.

```
function [D,C,G,B]= dyn_mod_ThreeLinkWalker(q,dq,parameters)
```

We start off the code by translating the input arguments into separate individual variables. For example, we break down the `q` vector into individual variables `th1`, `th2` and `th3`.

```
%% Define parameters
% q
th1=q(1);
th2=q(2);
th3=q(3);

% dq
dth1=dq(1);
dth2=dq(2);
dth3=dq(3);

r=parameters(1);   % length of a leg
m=parameters(2);   % mass of a leg
M_H=parameters(3); % mass of hips
M_T=parameters(4); % mass of torso
l=parameters(5); % distance between hips and torso
g=parameters(6); % acceleration due to gravity
```

Next, we define expressions for the D,C and B matrices and the G vector in terms of the variables defined above. Where do these expressions come from? You already have it! You simply plug in the symbolic notation for the D,C and B matrices and G vector that you generated in `SymbolicModelForThreeLinkRobot.m`. To reiterate, run your `SymbolicModelForThreeLinkRobot.m` code and copy and paste the symbolic values for each of the matrices into this `dyn_mod_ThreeLinkWalker.m` function. The way that I am doing it is that I first define each respective matrix in terms of a zeros matrix of the appropriate size using the `zeros()` function in Matlab. I then populate the non-zero values of the matrix using the symbolic notation that I found in `SymbolicModelForThreeLinkRobot.m`.

Note: this is one of the many areas in my code where things could have been simplified/cleaned up a bit more if I was not doing this for the first time and learning as I go. For example, rather than creating an entire function that gives numerical values for the D,C and B matrices and the G vector, you can have Matlab create individual functions for each matrix/vector and call these functions with the necessary parameters needed to generate a numerical value for them. Check out MathWork's page on "Generate MATLAB Functions from Symbolic Expressions."

```
% D Matrix
D=zeros(3,3);
D(1,1)=(r^2*(4*M_H + 4*M_T + 5*m))/4;
D(1,2)=-(m*r^2*cos(th1 - th2))/2;
D(1,3)=M_T*l*r*cos(th1 - th3);
D(2,1)=-(m*r^2*cos(th1 - th2))/2;
D(2,2)=(m*r^2)/4;
D(3,1)=M_T*l*r*cos(th1 - th3);
D(3,3)=M_T*l^2;

% C Matrix
C=zeros(3,3);
C(1,2)=-(dth2*m*r^2*sin(th1 - th2))/2;
C(1,3)=M_T*dth3*l*r*sin(th1 - th3);
C(2,1)=(dth1*m*r^2*sin(th1 - th2))/2;
C(3,1)=-M_T*dth1*l*r*sin(th1 - th3);

% G Matrix
G=zeros(3,1);
G(1)=-(g*r*sin(th1)*(2*M_H + 2*M_T + 3*m))/2;
G(2)=(g*m*r*sin(th2))/2;
G(3)=-M_T*g*l*sin(th3);
```

```
% B Matrix
B = [1 0;0 1;-1 -1];

return
```

4.1.4 ControlFunctions.m

The next Matlab function that we are going to look at is `ControlFunctions.m`. This is yet another function and it actually follows the same principle as `dyn_mod_ThreeLinkWalker.m`. We take numerical input variables of the system but this time output matrices essential for the control of our system. So we have the same input arguments `q,dq,parameters` in addition to a fourth input argument `th3desired`. This last input argument is the desired position for θ_3. The output arguments are various Lie derivative matrices and $h(x)$.

```
function [h,Lfh,L2fh,LgLfh] = ControlFunctions(q,dq,parameters,th3desired)
```

The first part of this function is identical to the first part of `dyn_mod_ThreeLinkWalker.m` where we translate the input arguments into individual variables. Simply copy and paste the code from that function here. Note that you do not need to redefine `th3desired`. We will leave that input argument as is.

In the next part of this function, we define expressions for `h,Lfh,L2fh,LgLfh` using the calculated symbolic notation that we derived in `SymbolicModelForThreeLinkRobot.m`. You can follow the technique that I used for `dyn_mod_ThreeLinkWalker.m` to do this. First define the matrix in terms of an appropriate sized zero matrix using the `zeros()` function. Then you populate the non-zero entries in the matrix. Since some of these expressions are ridiculously long, I will not be re-writing them in their entirety here. Please generate the appropriate expressions using `SymbolicModelForThreeLinkRobot.m`. Also note that Matlab typically does not like lines of code to be super long. When copying and pasting your long expressions in Matlab, use "`...`" to break up the lines between pluses ("+") or minuses ("-").

```
% define h
h=zeros(2,1);
h(1)=th3-th3desired;
h(2)=th1+th2;

% define Lfh
Lfh=zeros(2,1);
Lfh(1)=dth3;
Lfh(2)=dth1 + dth2;
```

```
% define L2fh
L2fh=zeros(2,1);
L2fh(1)=...
L2fh(2)=...

% define LgLfh
LgLfh=zeros(2,2);
LgLfh(1,1)=...
LgLfh(1,2)=...
LgLfh(2,1)=...
LgLfh(2,2)=...

return
```

4.2 Generating the Outputs

4.2.1 `ThreeLinkWalker_ODE45.m`

Now that we have functions to call for the numerical values of our dynamical system, we can create another function that will use those functions to solve the ODE that will apply our controller to our system resulting in outputs in terms of our position vector q, velocity vector $\dot{q}$ and corresponding time t. So this function will take the input arguments of `t_end` and `x0` where `t_end` is the length of the simulation (in seconds) and `x0` is the initial conditions of q and $\dot{q}$. More explicitly,

$$\mathtt{x0} = \begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \dot{\theta_1} \\ \dot{\theta_2} \\ \dot{\theta_3} \end{bmatrix} \tag{4.1}$$

The outputs of this function are the vectors `tout` and `xout` where `tout` is the time vector corresponding to the solution vector `xout` that contains values for the position q and velocity $\dot{q}$.

```
function [tout, xout]=ThreeLinkWalker_ODE45(t_end,x0)
```

The crux of this function is the Matlab `ode45` function. It is one of Matlab's many ODE solvers. This particular ODE solver is hailed as being quite versatile and thus a good starting place for most problems [5]. We will first set up the "options" that we want using the odeset function. We will also define our K_p and K_d values (using the method discussed in Section 3.2) that will be used to build up our controller in another function that we call in this function. We are also defining a variable called stepSizeAngle which we are using to say how wide we want our steps to be. This variable becomes more relevant when I talk about ThreeLinkEndStepEvents.m in Section 4.4.1.

```
refine=4;
RelTol = 10^-5;
AbsTol = 10^-6;

stepSizeAngle = pi/8;
wn = 10;
zeta = 0.8;
Kp = wn^2;
Kd = 2*zeta*wn;

% options withOUT event
options = odeset('Refine',refine, 'RelTol',RelTol,'AbsTol',AbsTol);

% options for with impact function
options = odeset('Events', @ThreeLinkEndStepEvents,'MaxStep',0.01);
```

Note that in comments that I have in the above code, I note that there are two sets of "options" that I am defining here. One set of options is for a case "without an event" and the other is for with an "impact function." I will discuss more on this later when I introduce the ThreeLinkEndStepEvents.m Matlab file in Section 4.4.1. For now, realize that if you used just the first set of options alone, our biped walker will be able to make a step but it will not be able to "end" the step and so will fall through the ground after making the motion of a step. The second set of options becomes relevant when we implement what is called the Events function later on.

Now with those parameters identified, we need to define one last parameter for the start time t_start and then call the ode45 function.

```
t_start=0;
[tout, xout] = ode45(@f, [t_start t_end], x0, options, stepSizeAngle, Kp, Kd);
```

So the function that the ode45 is solving is f. We need to set up this function f such that it represents the ODE that we are trying to solve for a controller applied to our dynamical system. We could make a separate Matlab file for f, but instead we are going to append it to the end of our ThreeLinkWalker_ODE45.m file.

The first part of the function is defining the parameters by calling on the `model_params_stiff_legs.m` file. We put those variables into a vector called "parameters" that we will use as an input argument for calls to other Matlab functions. We then set up our nonlinear dynamical system by calling the `dyn_mod_ThreeLinkWalker.m` file. Then we set up our control system by first calling on the `ControlFunctions.m` file. At this step we also define a `th3desired`. Professor Jessy Grizzle suggested a `th3desired` value of $\frac{\pi}{6}$. We use these control values to define our controller `u`. We are then able to generate the $\dot{x}$ vector using Eq. 3.10.

```
function dx=f(t,x,stepSizeAngle,Kp,Kd)

% Set Parameters
[r,m,M_H,M_T,l,g,p_st_foot,Mtotal]=model_params_stiff_legs();
parameters = [r,m,M_H,M_T,l,g,p_st_foot,Mtotal];

% Simulate NL model
dx=zeros(6,1);
q=x(1:3);
dq=x(4:6);
[D,C,G,B]=dyn_mod_ThreeLinkWalker(q,dq,parameters);

%NL controller
th3desired = pi/6;
[h,Lfh,L2fh,LgLfh]=ControlFunctions(q,dq,parameters,th3desired);

u=inv(LgLfh)*(-L2fh-Kd*Lfh-Kp*h); %controller

dx(1:3)=x(4:6);
dx(4:6)=inv(D)*(-C*dq - G + B*u);

return
```

4.2.2 `Points_ThreeLinkWalker.m`

Great, so you have the q and $\dot{q}$ values but if we want to "animate" the walker, we need to translate these angles into position coordinates for the ends of the links. That way our `ThreeLinkWalker_anim.m` Matlab file (which we will discuss in the next subsection) can draw lines between the points at every `tout` value, thus simulating motion.

The good news is that you already know how to do this conversion! You generated the position vectors in terms of angle position in Section 2.1 Kinematics of the Robot

(Position Information). You also typed up the code defining the position vectors in the SymbolicModelForThreeLinkRobot.m file. We are simply going to copy and paste the lines defining the positions for p_hip, p_torso and p_sw_foot. Before we do that, of course, we need to make a call to the model_params_stiff_legs.m file to get the numerical values of our variables.

```
function [p_st_foot,p_hip,p_torso,p_sw_foot] = Points_ThreeLinkWalker(q,dq)

% Set parameters
[r,m,M_H,M_T,l,g,p_st_foot]=model_params_stiff_legs();
th1 = q(1);
th2 = q(2);
th3 = q(3);

% Calculate positions based on geometry
p_hip = p_st_foot + [r*sin(th1) ; r*cos(th1)]; % position of hip
p_torso = p_hip + [l*sin(th1) ; l*cos(th3)]; % position of torso
p_sw_foot = p_hip + [r*(-sin(th2)) ; r*(-cos(th2))]; % position of swing foot

end
```

4.3 Visualizing the Data

4.3.1 ThreeLinkWalker_anim.m

So close! We have all that we need to animate stuff! The first step is to define our input arguments. All we need is the time vector t, solution vector xout that contains the solutions q and $\dot{q}$, and time step Ts. Actually, honestly we do not even use t and Ts so I guess you could leave those out. But I'm keeping them because why not.

```
function ThreeLinkWalker_anim(t,xout,Ts)
```

Next, we are going to use those input arguments (specifically xout) to define q and $\dot{q}$. We will also define the variables xRow and xCol as the dimensions of xout which we will use later to know how many times to run through a for loop.

```
% Define inputs
q = xout(:,[1 2 3]);
dq = xout(:,[4 5 6]);

[xRow xCol] = size(xout); % number of data points we are dealing with
```

Next, we will set up the figure that we will be using to do the animation. We will define the dimensions of the plot as well as draw a horizontal line that will represent the "ground" that the biped will walk on.

```
%% set up plot
figure();

% define plot window boundaries
xmax = 20;
xmin = -1;
ymax = 2;
ymin = -2;
axis([xmin xmax ymin ymax]);
axis off

% plot ground
ground = line([0 xmax],[0 0]);
set(ground,'LineWidth',3,'Color','k');
```

Next, we are going to find and plot the initial conditions of each of the three links by first calling on the `Points_ThreeLinkWalker.m` file and then using the Matlab `line` function to draw a line between the points for each link using those position values generated in `Points_ThreeLinkWalker.m`.

```
%% Find and plot initial positions of links
% position calculations
[p_st_foot,p_hip,p_torso,p_sw_foot] = Points_ThreeLinkWalker(q,dq);

% position of stance link
link_st = line([p_st_foot(1) p_hip(1)],[p_st_foot(2) p_hip(2)]);
set(link_st,'LineWidth',2,'Color','b');

% position of swing link
link_sw = line([p_hip(1) p_sw_foot(1)],[p_hip(2) p_sw_foot(2)]);
set(link_sw,'LineWidth',2,'Color','r');

% position of torso
link_torso = line([p_hip(1) p_torso(1)], [p_hip(2) p_torso(2)]);
set(link_torso,'LineWidth',2,'Color','g');
```

Finally, we will iterate through the link positions using a `for` loop. The amount of values that we are sifting through corresponds to the number of rows in `xout`. We use the Matlab function `drawnow` to update the figure as we redefine our link positions. We also use `pause()` to ensure that each redraw of the links stays on screen long enough to produce the image of fluid motion.

```
%% Iterate between link positions

for j = 1:xRow
    % Calculate points for each iteration
    q_current = xout(j,1:3);
    dq_current = xout(j,4:6);
    [p_st_foot_current,p_hip_current,p_torso_current,p_sw_foot_current] = ...
        Points_ThreeLinkWalker(q(j,:),dq(j,:));

    set(link_st,'XData',[p_st_foot_current(1) p_hip_current(1)], ...
        'YData',[p_st_foot_current(2) p_hip_current(2)]);
    set(link_sw,'XData',[p_hip_current(1) p_sw_foot_current(1)], ...
        'YData',[p_hip_current(2) p_sw_foot_current(2)]);
    set(link_torso,'XData',[p_hip_current(1) p_torso_current(1)], ...
        'YData',[p_hip_current(2) p_torso_current(2)]);

    drawnow;
    pause(0.001);
end
return;
```

4.3.2 `ThreeLinkWalker_sim.m`

We have produced Matlab code to define variables and parameters, perform calculations, and animate. Now we just need code to call on all of that stuff! `ThreeLinkWalker_sim.m` serves as the top level script that we will be using to "Run" our code. It is not a function; it is a script. It is also here where we define our initial conditions. I am also using this script to plot some potentially helpful figures of q and $\dot{q}$ over time.

```
clear all
close all
clc

% Set Initial Conditon
```

```
x0=[-pi/8;pi/8;pi/6;1.5;-1.5;0] ; %[th1, th2, th3, dth1, dth2, dth3]
t_end=1; % length of simulation
[tout, xout]=ThreeLinkWalker_ODE45(t_end,x0);

q = xout(:,[1 2 3]);
dq = xout(:,[4 5 6]);

figure(1);
plot(tout,q,'linewidth',2.5);
xlabel('Time (Sec)','fontsize',16);
ylabel('q (rad)','fontsize',16);
title('Link Angles','fontsize',18);
legend('\theta_1','\theta_2','\theta_3')
grid on

figure(2);
plot(tout,dq,'linewidth',2.5);
xlabel('Time (Sec)','fontsize',16);
ylabel('dq/dt (rad/s)','fontsize',16);
title('Change in Link Angles','fontsize',18);
legend('d\theta_1','d\theta_2','d\theta_3')
grid on


% Animation to see what is happening
pause(1)
ThreeLinkWalker_anim(tout,xout,0.01);
```

4.4 Ending the Step

4.4.1 `ThreeLinkEndStepEvents.m`

You would think that we were done, but we are not! Recall in Section 4.2.1 where I was setting up the "options" for the `ode45` function. This set of options is rather generic and results in our biped not actually "finishing" a step. To make our biped know when the step is over (besides messing with the `t_end` variable, which by the way had to be 1 for our particular set of initial conditions), we need to create an "event" using Matlab's `Events` function (see [6] for more information). Basically, you use an "event" to tell your ODE solver when to stop the solution. You do this instead of, say, specifying a particular end time (like we were doing with the `t_end` variable) when you do not know the specific time an "event" might occur. A visual example is an apple falling from a tree. In this example, you would want your ODE solver to stop once the apple hits the ground, but you may not know the exact time this may happen beforehand. You may also want to use an event even when you do not want to terminate a solution. For example, you may use an event to keep track of how many times the moon orbits the Earth. So in this case, an "event" would be one full orbit about the Earth.

We are going to use the `Events` function to end a step. Specifically, we are going to use it to say that once the stance leg reaches a certain angle, we want to terminate the solution. The outputs of this function are `value, isterminal,` and `direction`.

The `value` variable is what we want to eventually equal to 0. Once `value` equals 0, the the "event" would have occurred. In our application, the `value` variable is describing how close we are for getting the current angle of the stance leg to some desired angle value. The desired angle we want is stored in the `stepSizeAngle` variable that we defined in `ThreeLinkWalker_ODE45.m`. However, we are going to create an `if statement` saying that if a `stepSizeAngle` variable is not given, we are going to set the default value to $\frac{\pi}{8}$. A fun little fact here is that if we used a value as high as $\frac{\pi}{4}$, we would be taking 90 degree step sizes and thus our biped might not have the momentum to progress to another step afterwards.

The `isterminal` output variable says whether we want to terminate the ODE solver once the event occurs. A value of 1 says that we do want to terminate. A value of 0 says that we do **not** want to terminate. In our application, we do want to terminate because we are ending a step (and later going to switch to the impact function which I will talk about in Section 5) so our value will be 1. An example of a case we would use a value of 0 would be the example that I mentioned previously where we were using the `Events` function to track the number of orbits of the moon around the Earth. In this example, we are using `Events` to recognize when a full orbit has been made, but keep orbiting afterwards – ie do not terminate.

The `direction` output variable says which direction we are approaching the `value` equal to 0 from. If we are starting with positive numbers and decreasing to 0, then `direction` would equal to -1. If we are starting with negative numbers and increasing to 0, the `direction` would equal to 1. In our application, we are approaching a `value` of 0 from the positive values, thus our `direction` would equal to -1.

The input parameters that we are using are `t, x,` and `stepSizeAngle`. Actually, we are not really using the time `t` variable, so I guess you could leave that off. We do use the solution vector `x` to ultimately extract the current position value for the angle of the stance leg. Finally, we use the `stepSizeAngle` input to know what step size we are taking.

That code looks like this

```
function [value,isterminal,direction] = ThreeLinkEndStepEvents(t,x,stepSizeAngle)

if nargin < 3
    stepSizeAngle=pi/8;
end

q = x(1:3);
dq = x(4:6);

value = stepSizeAngle-q(1);

isterminal = 1;
direction = -1;

return
```

NOTE: `nargin` is a special Matlab variable that stores the number of input arguments that was used when the function was called.

4.5 You did it!

Hooray, you did it! You have written all of the code that you need to get your biped to make its first step. Now go to your `ThreeLinkWalker_sim.m` file and run it to see all of your hard work pay off!

5 The Impact Function

So you took your first step. But how do we take multiple steps in a row? In the last section we "ended" the step just before the swing leg hit the ground to keep things simple. In this section, we are going to model what happens when that swing leg makes contact with the ground using what is called the "impact" function. After impact, the swing leg becomes the stance leg and the stance leg becomes the new swing leg and the whole process of the new swing leg moving to take the next step starts all over again.

So let's explore how to model the impact that the swing foot of our biped makes with the floor. How long does it take for the swing leg to decelerate to 0 when it hits the ground? The actual answer is somewhere between 25-30 ms – practically instantaneous. If we treated the impact event as **not** instantaneous, then we would have to model the forces that act on the swing leg to decelerate it quickly to a stop. This could get especially tricky if our biped is moving across different surfaces. We would have to constantly redefine various parameters to fit the exact conditions that we were in. That sounds like a nightmare!

To avoid this modeling problem, we assume that the impact is indeed instantaneous. As the time of impact Δt_I approaches zero, the impact forces go to impulses. To reiterate in math-speak

$$\begin{aligned} \Delta t_I &= \text{impact duration} \to 0 \\ \text{impact forces} &\to \text{impulses} \end{aligned}$$

Impulses in a second order ODE results in a jump in the velocity component of the state whereas the position remains unchanged. This means that over an instantaneous span of time, there will be a *velocity* before impact (which we will call $\dot{q}^-$) and a *velocity* after impact (which we will call $\dot{q}^+$). Additionally, the *position* before impact (q^-) and the *position* after impact (q^+) is equivalent ($q^- = q^+$).

5.1 The Floating Base Model

In Section 2, we derived the equations of motion for our biped (Eq. 2.7). What I did not tell you was that this derivation of the equations of motion was based on the **pinned base** model, meaning I assumed that the stance foot was pinned to a point on the ground. This model is fine if you plan on just taking one step. But if you want to take multiple steps, you cannot have a foot pinned to the ground! At some point, that stance foot needs to move in order to become the new swing foot. Thus, we shift to the **floating base** model.

So to be explicitly clear, let me rewrite the equations of motion derived from the pinned based model here:

$$D\ddot{q} + C\dot{q} + G = \Gamma = Bu$$

The equations of motion derived from the floating base model are defined as follows:

$$D_e\ddot{q}_e + C_e\dot{q}_e + G_e = \Gamma_e = B_eu + J_{st}^T F_{st} + J_{sw}^T \delta F_{sw} \tag{5.1}$$

Not much has really changed. We added an "e" subscript to our vectors and matrices and we have these new terms $J_{st}^T F_{st}$ and $J_{sw}^T \delta F_{sw}$. So what does this new equation mean? First, the "e" subscript stands for "extended," and we could have just as easily used an "fb" subscript to stand for "floating base" instead. This "extended" version of the matrices and vectors indicate the presence of two additional position coordinates in our generalized coordinate vector q. We will call these two new coordinates z_1 and z_2. These represent the x (z_1) and y (z_2) position of the stance leg as shown in Figure 4. As a reference, in the pinned base model $z_1 = z_2 = 0$ to indicate that the stance foot never moved. Now since we are "un-pinning" the stance foot, we need position coordinates to indicate its location.

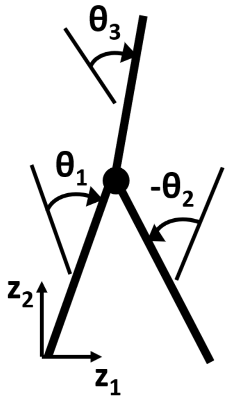

Figure 4: Extended coordinates for 3-link biped

Thus, our generalized coordinates q become the extended generalized coordinates q_e defined as

$$q_e = \begin{bmatrix} q \\ z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ z_1 \\ z_2 \end{bmatrix} \tag{5.2}$$

Similarly, our extended velocity vector becomes

$$\dot{q}_e = \begin{bmatrix} \dot{q} \\ \dot{z}_1 \\ \dot{z}_2 \end{bmatrix} = \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{z}_1 \\ \dot{z}_2 \end{bmatrix} \tag{5.3}$$

The D,C and B matrices and G vector will change as a result of the new generalized coordinate definitions. For example the D matrix which was $3x3$ will become the D_e matrix which is $5x5$. Because I still do not hate myself, I will not be re-deriving these matrices by hand, and neither should you. That is what Matlab is for! We will discuss how to do this in Section 6.

Another new feature in the floating base model equation of motion in Eq. 5.1 is the $J_{st}^T F_{st}$ terms. J_{st}^T indicates the transpose of the jacobian for the stance leg and F_{st} represents the ground reaction forces on the stance leg. These two terms are factored in the pinned based model; however, they show up as part the D and C matrices and the G vector. The floating base model extracts these terms from those matrices so that they may be more easily factored in future calculations.

Similarly, the $J_{sw}^T \delta F_{sw}$ terms are also technically factored in the pinned based model, but the floating base model extracts them so that they may be factored into our calculations. J_{sw}^T indicates the transpose of the jacobian for the swing leg and F_{sw} represents the ground reaction forces on the swing leg as the swing foot hits the ground. The δ term indicates that the overall $J_{sw}^T \delta F_{sw}$ will have an impulsive force over as specific instantaneous amount of time (when the swing foot hits the ground). It simply serves as a notation tool to remind us of this fact. We could have just as easily written the entire term to be just $J_{sw}^T F_{sw}$.

Demystifying the Equations of Motion

So now we have seen the equations of motion written in two different ways, one for the pinned based model and one for the floating base model. You may be wondering, are there other ways to write the equations of motion? What is its most general form?

The most general form of the equations of motion is written as the following

$$D_e\ddot{q}_e + C_e\dot{q}_e + G_e = \Gamma_e = B_e u + J^T F + J_{ext}^T F_{ext} \tag{5.4}$$

where $D_e, C_e, G_e, B_e, \ddot{q}_e, \dot{q}_e, \Gamma_e$, and u are defined exactly as we have defined them before in the floating base model. Remember that the Γ_e indicates the total internal and external forces. So while the $B_e u$ terms represent the internal forces (in our case, the torque inputs from the motors), the $J^T F$ and $J_{ext}^T F_{ext}$ terms represent the total external forces. More specifically, $J^T F$ represents the ground reaction forces (in our three-link application, we have two sets of ground reaction forces during impact – for both the swing and stance legs – and just one set of ground reaction forces during any other time of the walk – acting on just the stance foot) and $J_{ext}^T F_{ext}$ represents all of the other external forces that are not the ground reaction forces (we do not have any in our application for the three-link biped). The jacobians here serve as transformation matrices to convert to the generalized coordinates q_e because technically the forces F and F_{ext} are defined in the world coordinates.

Another popular form of the equations of motion is

$$D\ddot{q} + H(q,\dot{q}) = Bu + J^T F \tag{5.5}$$

where $H(q,\dot{q}) = C\dot{q} + G$.

Similarly, another form you might see is

$$D\ddot{q} = F_v(q,\dot{q}) + Bu + J^T F \tag{5.6}$$

where $F_v(q,\dot{q}) = -C\dot{q} - G$.

All of these forms of the equation of motion are equally valid. Depending on what you are trying to do, you may favor one form over the other.

NOTE: In other literature, the ground reactions forces may be defined as λ instead of F.

5.2 Deriving the Impact Map

Using Eq. 5.1 we can derive what is called the **impact map** which is a tool we will use to determine the velocities of the swing leg after impact in addition to the ground reaction impulsive forces.

Since finding velocity is the end goal, we need to integrate Eq. 5.1. As we established at the beginning of this section, the impact of the swing foot on the ground is instantaneous and therefore occurs over an instantaneous length of time. The time before impact is t^- and the time after impact is t^+. Integrating over this yields

$$\int_{t^-}^{t^+} \left(D_e \ddot{q}_e + C_e \dot{q}_e + G_e = B_e u + J_{st}^T F_{st} + J_{sw}^T \delta F_{sw}\right) \tag{5.7}$$

Here is where you are going to do a little more research in the literature to understand what exactly is going on here. But the $B_e u$ term goes to 0 since it is physically impossible for the motors controlling the links to produce instantaneous torques. The $C_e \dot{q}_e$ and G_e terms also go to 0 since the changes in these terms are going to be so small that they can be considered negligible. The ground reaction forces on the stance foot F_{st} go to 0 because there is no change in these forces over an instantaneous length of time. Thus, the only ground reaction forces we are left with are on the swing foot F_{sw}. Since we are integrating over an instantaneous amount of time, these forces become impulses F_I. We ignore the δ term because it just served as a notation tool to indicate that the ground reaction forces on the swing foot were impulsive. Therefore we get

$$D_e(\dot{q}_e^+ - \dot{q}_e^-) = J_{sw}^T F_I \tag{5.8}$$

where D_e is the extended D matrix (5x5 matrix), $\dot{q}_e^+$ are the velocities after impact (5x1 vector), $\dot{q}_e^-$ are the velocities before impact (5x1 vector), J_{sw}^T is the transpose of the jacobian of the swing leg $\left(\text{where } J_{sw} = \frac{\partial p_{foot,e}^{sw}}{\partial q_e}\right)$, and F_I are the impulsive forces (2x1 vector) such that

$$F_I = \begin{bmatrix} F_{I,1} \\ F_{I,2} \end{bmatrix} = \begin{bmatrix} \text{horizontal component} \\ \text{vertical component} \end{bmatrix} \tag{5.9}$$

NOTE: In some literature, F_I could be referred to as Λ.

Our known values include $\dot{q}_e^-$. We know that the non-extended component of this vector $\dot{q}^-$ is simply the angular velocity values of the links right before impact which you can get from the `Event` function that we created in Section 4. Meanwhile, we know that the $\dot{z}_1$ and $\dot{z}_2$ components of this vector are both equal to 0 because the stance leg is not moving before impact (and as we defined, z_1 and z_2 are the positions of the stance foot so therefore $\dot{z}_1$ and $\dot{z}_2$ are the velocitices of the stance foot). Thus,

$$\dot{q}_e^- = \begin{bmatrix} \dot{q}^- \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{bmatrix} \tag{5.10}$$

All together, that makes 5 known values from $\dot{q}_e^-$. But what we do not know are the velocity values after impact $\dot{q}_e^+$ (5 values) or the impulsive forces F_I (2 values). So that means we would get 5 equations but have 7 unknowns. Bummer...

We need 2 more equations. Luckily, I still have a few tricks up my sleeve! Something else that we know is that after impact, the swing leg velocity will equal to 0 (since – I would hope – our foot would not slide as it hits the floor). Thus,

$$\frac{d}{dt} p_{foot}^{sw} \bigg|_{t^+} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.11}$$

Using chain rule, we get

$$\left(\frac{\partial}{\partial q_e} p_{foot}^{sw} \right) \dot{q}_e^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.12}$$

NOTE: Remember that the positions before and after impact are equal to each other (ie $q_e^+ = q_e^-$) and so I simply refer to both as just q_e.

Recognizing $\frac{\partial}{\partial q_e} p_{foot}^{sw}$ as the jacobian of the swing foot J_{sw} we get

$$J_{sw} \dot{q}_e^+ = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{5.13}$$

So let us put all of that together, our 7 knowns and 7 unknowns into a single matrix equation

$$\begin{bmatrix} D_e & -J_{sw}^T \\ J_{sw} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix} \begin{bmatrix} \dot{q}_e^+ \\ F_I \end{bmatrix} = \begin{bmatrix} D_e \dot{q}_e^- \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{bmatrix} \tag{5.14}$$

Eq. 5.14 is known as the **impact map**. Looking at this matrix equation, you will be able to see that the top row is equivalent to Eq. 5.8 and the bottom row is equivalent to Eq. 5.13.

Just for reference, here are the sizes of each matrix within this matrix equation

$$\begin{bmatrix} \overbrace{D_e}^{5x5} & \overbrace{-J_{sw}^T}^{5x2} \\ \underbrace{J_{sw}}_{2x5} & \underbrace{\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}}_{2x2} \end{bmatrix} \begin{bmatrix} \overbrace{\dot{q}_e^+}^{5x1} \\ \underbrace{F_I}_{2x1} \end{bmatrix} = \begin{bmatrix} \overbrace{D_e \dot{q}_e^-}^{5x1} \\ \underbrace{\begin{bmatrix} 0 \\ 0 \end{bmatrix}}_{2x1} \end{bmatrix}$$

Using matrix inversion, we can solve for $\dot{q}_e^+$ and F_I.

$$\begin{bmatrix} \dot{q}_e^+ \\ F_I \end{bmatrix} = \begin{bmatrix} D_e & -J_{sw}^T \\ J_{sw} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}^{-1} \begin{bmatrix} D_e \dot{q}_e^- \\ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{bmatrix} \tag{5.15}$$

Hybrid Systems

You have made it this far, so let me introduce a new term for you. A **hybrid system** is a system that cannot be described by a single set of dynamics – like our system! In our three-link biped walker, we have an impact when our swing foot touches the ground which changes our dynamic model. Thus, we have two dynamic models for our one system: one for during a step and one for impact.

One way to help visualize a hybrid system is using a diagram called a *hybrid automata*. The hybrid automata for our system is depicted in Figure 5.

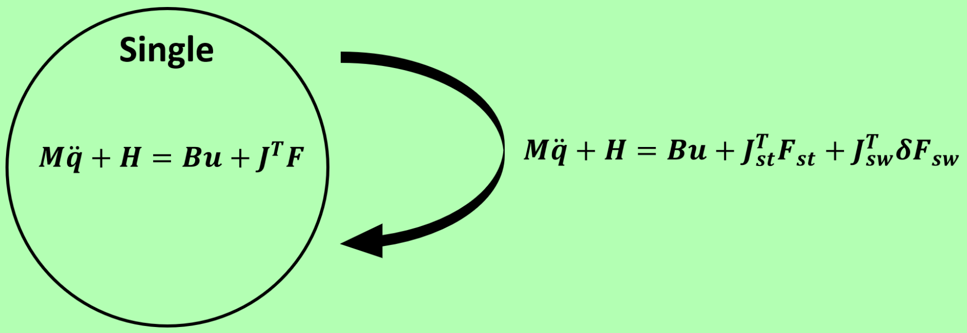
Single
$M\ddot{q} + H = Bu + J^T F$
$M\ddot{q} + H = Bu + J_{st}^T F_{st} + J_{sw}^T \delta F_{sw}$

Figure 5: Hybrid Automata for a Symmetric, Planar Biped

Our two dynamic models are represented by the two equations of motion in the model. The circle depicts the continuous domain (during a step) and outside that is our impact domain. The arrow indicates that we cycle from the continuous to impact domain with each step. Since our system is both planar and symmetric, we just have one continuous domain which I just call "Single."

A hybrid automata for a more complicated system is depicted in Figure 6.

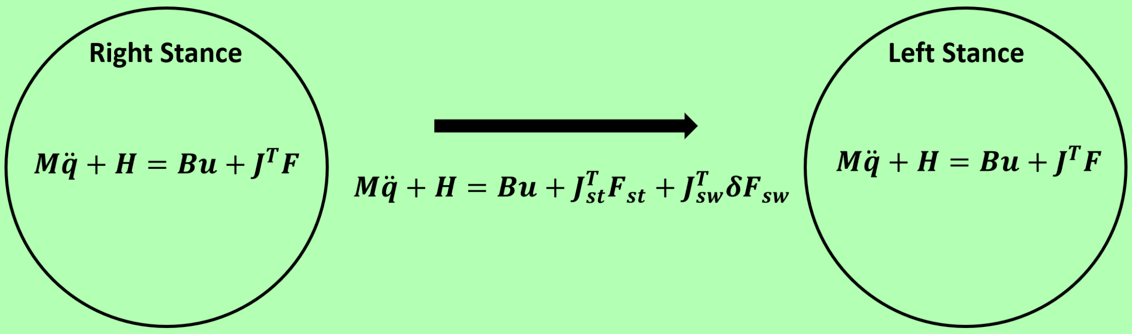
Right Stance
$M\ddot{q} + H = Bu + J^T F$
$M\ddot{q} + H = Bu + J_{st}^T F_{st} + J_{sw}^T \delta F_{sw}$
Left Stance
$M\ddot{q} + H = Bu + J^T F$

Figure 6: Hybrid Automata for a Non-Planar Biped

Here, we have a non-planar biped which means that there is a distinction between when the right foot is the stance leg versus the left foot. Thus, our entire system is represented by three dynamic models: one for impact, one for when the right leg is the stance leg, and one for when the left leg is the stance leg.

6 Taking the Second Step (and Beyond) in Matlab Simulation

So now with all of the mathematical derivations explained in the previous section, we can apply our impact function in Matlab so that we can make the biped take a second consecutive step. We will be implementing three new Matlab functions:

- `ExtendedSymbolicModelForThreeLinkRobot.m` – Used to generate the symbolic model for the extended (floating base) biped.
- `ext_dyn_mod_ThreeLinkWalker.m` – Used to calculate the numerical values for certain matrices and vectors from our symbolic model.
- `Impact.m` – Uses the impact map to calculate $\dot{q}_e^+$ and F_I.

Additionally, we will be modifying the following current existing Matlab files

- `Points_ThreeLinkWalker.m` – Add a new input argument for `p_foot` and use that to update the position of the stance foot based off of what step you are on.
- `ThreeLinkWalker_anim.m` – Add input parameter `p_foot` and use when animating the link positions.
- `ThreeLinkWalker_sim.m` – Add code to incorporate defining `p_foot` for first and second step. Add code that uses the `Impact.m` function to make calculations for the next step.

6.1 Some new code

6.1.1 `ExtendedSymbolicModelForThreeLinkRobot.m`

Let's begin by looking at the new Matlab files. In this first script, you are going to first simply copy and paste everything that you had in your `SymbolicModelForThreeLinkRobot.m` script into this new script. Now make the following modifications.

First, update your symbolic variable definitions to incorporate the new z_1 and z_2 coordinates into the generalized coordinates. You will also need to rename `q` and `dq` to `qe` and `dqe` respectively to indicate that we are using the extended coordinates. Be sure to replace every instance of `q` and `dq` in the script to `qe` and `dqe`

```
%% Define Symbolic Terms

syms r l m M_H M_T g
syms th1 th2 th3
syms dth1 dth2 dth3
syms z1 z2 dz1 dz2

qe = [th1; th2; th3; z1; z2];
dqe = [dth1; dth2; dth3; dz1; dz2];
```

In the next section where you define your position vectors, you will only be changing one thing. Instead of defining the position of the stance foot as (0,0), you will be defining it as (z_1, z_2).

```
p_st_foot = [z1; z2];
```

Now, I want you to create a new section between the defining the position vectors and defining the velocity vectors sections called the `Jac` section. In this part of the script, you will be defining the jacobian of the swing foot (which will be used in the impact map calculation).

```
%% Jac
Jac = jacobian(p_sw_foot,qe);
```

The defining the velocity vectors, kinetic energy, and potential energy sections stay the same. In the section where you define your matrices for the equations of motion, you will add the "e" subscript to each of the D and C matrices and the G vector to indicate that these are the matrix/vector definitions for the floating base model.

```
Ge=jacobian(PE,qe).';
Ge=simplify(Ge);
De=simplify(jacobian(KE,dqe).');
De=simplify(jacobian(De,dqe));

syms Ce real
n=max(size(qe));
for k=1:n
    for j=1:n
        Ce(k,j)=0*g;
        for i=1:n
            Ce(k,j)=Ce(k,j)+(1/2)*(diff(De(k,j),qe(i))...
            +diff(De(k,i),qe(j))-diff(De(i,j),qe(k)))*dqe(i);
        end
    end
end
Ce=simplify(Ce);
```

In the last section of the script where you define and calculate your variables for the control system, you are first going to update your B matrix to append a $2x2$ matrix of zeros at the end to indicate that the control inputs at z_1 and z_2 are 0 (since our motors are not influencing these points). Also rename B to Be.

```
Be = [1 0;0 1;-1 -1; 0 0; 0 0];
```

Lastly, where you define your g vector, you are going to need to modify the size of your zero matrix to account for the new sizes of the equations of motion matrices.

```
g = [zeros(5,2); inv(De)*Be];
```

6.1.2 ext_dyn_mod_ThreeLinkWalker.m

This Matlab function is similar to the dyn_mod_ThreeLinkWalker.m Matlab function that we created earlier when we were just taking one step, except this time we only care about outputing the numerical values for the De matrix and the jacobian of the stance foot, Jac. At the beginning of the code, you will need to define your parameters in the exact same way you did in dyn_mod_ThreeLinkWalker.m, only this time be sure to also incorporate definitions for z1, z2, dz1, and dz2 since our input arguments are qe and dqe instead of q and dq this time. The symbolic definitions of De and Jac can be directly generated from ExtendedSymbolicModelForThreeLinkRobot.m.

```
function [De,Jac]= ext_dyn_mod_ThreeLinkWalker(qe,dqe,parameters)

%% Define parameters
% qe
th1=qe(1);
th2=qe(2);
th3=qe(3);
z1=qe(4);
z2=qe(5);

% dqe
dth1=dqe(1);
dth2=dqe(2);
dth3=dqe(3);
dz1=dqe(4);
dz2=dqe(5);

r=parameters(1);   % length of a leg
```

```
m=parameters(2);   % mass of a leg
M_H=parameters(3); % mass of hips
M_T=parameters(4); % mass of torso
l=parameters(5); % distance between hips and torso
g=parameters(6); % acceleration due to gravity


%% Define Matrices
% (using symbolic (model generated from ExtendedSymbolicModelForThreeLinkRobot.m)

% De Matrix
De=zeros(5,5);
De(1,1)=(r^2*(4*M_H + 4*M_T + 5*m))/4;
De(1,2)=-(m*r^2*cos(th1 - th2))/2;
De(1,3)=M_T*l*r*cos(th1 - th3);
De(1,4)=(r*cos(th1)*(2*M_H + 2*M_T + 3*m))/2;
De(1,5)=-(r*sin(th1)*(2*M_H + 2*M_T + 3*m))/2;
De(2,1)=-(m*r^2*cos(th1 - th2))/2;
De(2,2)=(m*r^2)/4;
De(2,4)=-(m*r*cos(th2))/2;
De(2,5)=(m*r*sin(th2))/2;
De(3,1)=M_T*l*r*cos(th1 - th3);
De(3,3)=M_T*l^2;
De(3,4)=M_T*l*cos(th3);
De(3,5)=-M_T*l*sin(th3);
De(4,1)=(r*cos(th1)*(2*M_H + 2*M_T + 3*m))/2;
De(4,2)=-(m*r*cos(th2))/2;
De(4,3)=M_T*l*cos(th3);
De(4,4)=M_H + M_T + 2*m;
De(5,1)=-(r*sin(th1)*(2*M_H + 2*M_T + 3*m))/2;
De(5,2)=(m*r*sin(th2))/2;
De(5,3)=-M_T*l*sin(th3);
De(5,5)=M_H + M_T + 2*m;


% Jac = jacobian(p_sw_foot,qe) where p_sw_foot is the extended form
Jac = [r*cos(th1) -r*cos(th2) 0 1 0 ; -r*sin(th1)  r*sin(th2) 0 0 1];

return
```

6.1.3 `Impact.m`

The final **new** Matlab file that you will create is `Impact.m`. In this Matlab function, you are going to want to output values for $\dot{q}_e^+$ (`dqe_plus`) and F_I (`F_I`) given the input arguments q^- (`q_minus`) and $\dot{q}^-$ (`dq_minus`).

First, you are going to want to create your extended position and velocity vectors `qe_minus` and `dqe_minus` by appending `[0 ; 0]` to both input argument vectors `q_minus` and `dq_minus`. Next, you are going to want to call on your `model_params_stiff_legs.m` file to get your numerical parameter values. Then you will use those values and your `qe_minus` and `dqe_minus` values to call on your `ext_dyn_mod_ThreeLinkWalker.m` file to get `De` and `Jac`. Using these values, you will set up your impact map as was defined in Eq. 5.14. You will define an `A` matrix and `B` matrix such that you can solve for the `x` vector using matrix inversion and multiplication. The first five values in `x` will be your `dqe_plus` values while the last two values will be your `F_I` values.

```
function [dqe_plus,F_I] = Impact(q_minus,dq_minus)

qe_minus = [q_minus; 0; 0];
dqe_minus = [dq_minus; 0; 0];

[r,m,M_H,M_T,l,g]=model_params_stiff_legs();
parameters = [r,m,M_H,M_T,l,g];

[De,Jac] = ext_dyn_mod_ThreeLinkWalker(qe_minus,dqe_minus,parameters);

% Solve system of equations
A = [De -Jac'; Jac zeros(2,2)];
B = [De*dqe_minus; 0; 0];

x = inv(A)*B;

dqe_plus = x(1:5);
F_I = x(6:7);

return
```

6.2 Updating older code

6.2.1 `Points_ThreeLinkWalker.m`

Now that we have our new Matlab files, we need to make some changes to our older files to make everything mesh together.

One of the major changes that is happening overall is the incorporation of a new variable `p_foot`. We use this variable to keep track of the position of the stance foot. The idea is that the location of the stance foot starts off at $(0,0)$ throughout the first step. Then the position of the stance foot for the next step becomes the value of the final position of the swing foot from the last step. In this way, the old stance foot becomes the swing foot and the old swing foot becomes the new stance foot.

The changes to `Points_ThreeLinkWalker.m` is quite simple. First, you add a new input argument `p_foot`.

```
function [p_st_foot,p_hip,p_torso,p_sw_foot] = Points_ThreeLinkWalker(q,dq,p_foot)
```

Then, you add the following `if statement` **before** you do the position calculations for the hip, torso, and swing foot.

```
if nargin > 2
    p_st_foot = p_foot;
end
```

What this is doing is ensuring that our biped is progressing forward for each new step that it takes. We may not have a `p_foot` value every time we call this function which is why we are using this `if statement`.

6.2.2 `ThreeLinkWalker_anim.m`

The changes in this Matlab function will also include the incorporation of the `p_foot` variable. First, add `p_foot` to the input arguments.

```
function ThreeLinkWalker_anim(t,xout,Ts,p_foot)
```

Next, in the last section of this code where you are iterating between the link positions, you need to add one more small change. Where you call the `Points_ThreeLinkWalker.m` function to assign the "current" link positions, you are going to want to pass the `p_foot` value at the `j`'th position. You will have to transpose this vector because of the way that we will be defining `p_foot` in `ThreeLinkWalker_sim.m`.

```
[p_st_foot_current,p_hip_current,p_torso_current,p_sw_foot_current] = ...
        Points_ThreeLinkWalker(q(j,:),dq(j,:),p_foot(j,:)');
```

6.2.3 `ThreeLinkWalker_sim.m`

For the sake of clarity, I am just going to re-write the whole code for this Matlab script. So let's start from a blank Matlab script. The overview for this script is that we are going to first make all of the calculations for the first step, then make all of the calculations for the second step, and then we are going to append all those calculated vectors together and send them to our animation function to animate.

Since this is the top level script, we are going to start it off by clearing up the workspace and closing everything.

```
clear all
close all
clc
```

Now we make the calculations for the first step. This process is the exact same as what we did before (with one minor addition). We set our initial conditions and the length of the simulation and send those values to our ODE solver to generate `tout` and `xout`. As before, we can extract `q` and `dq` from `xout`. The small addition that I want to make here is to define a vector for `p_foot`. The first position of the stance foot will be $(0,0)$, so what we need is to define this position as a row vector and then replicate this row "n" times where "n" is the number of rows in `tout` (or `xout`, since they are the same length). This way we will have a corresponding `p_foot` for every data point. To do this replication, we use Matlab's `repmat` function.

```
%% First Step
% Set Initial Condition
x0=[-pi/8;pi/8;pi/6;1.5;-1.5;0]; %[th1, th2, th3, dth1, dth2, dth3]
t_end=1; % length of simulation

% Calculate
[tout, xout]=ThreeLinkWalker_ODE45(t_end,x0);

% Define q and dq
q = xout(:,[1 2 3]);
dq = xout(:,[4 5 6]);

% Location of the stance foot
p_foot0 = [0 0];
p_foot=repmat(p_foot0, size(tout));
```

Now for the second step. We need an updated set of "initial conditions" to use to send to our ODE solver. These "initial conditions" are simply the q and $\dot{q}$ values **immediately after** impact **AND** after we have swapped our swing leg for the stance leg and vice versa. We will combine these vectors into a single vector called x^+ (x_plus), where $x^+ = [q;\dot{q}]$.

So first, we need to calculate the q and $\dot{q}$ values after impact (q^+ and $\dot{q}^+$). This is where the Impact.m function comes into play. To call Impact.m, we need q_minus and dq_minus (which are the set of generalized position and velocity vector values before impact). These values are equal to the last row of values in q and dq from the first step (since these are the final q and $\dot{q}$ values before impact). From the Impact.m function, we get dq_plus and F_I. But what is q_plus? We already know it! Remember, position does not change after impact so q_plus = q_minus.

```
%% Second Step
q_minus = q(end,:)';
dq_minus = dq(end,:)';

[dq_plus, F_I] = Impact(q_minus,dq_minus);

q_plus = q_minus;
```

Now we can define a vector x^- (x_minus) that comprises of q_plus and dq_plus which are the position and velocity vectors before the swapping of feet. To swap the feet, we will use a matrix which I wall call swap that will be used to swap the θ_1 values with the θ_2 values as well as swap the $\dot{\theta}_1$ values with the $\dot{\theta}_2$ values. The x_plus vector is simply equal to the multiplication of the swap matrix and x_minus vector. Use this x_plus to call the ODE solver (using the same t_end from the first step). Be sure to call the output variables tout_next and xout_next this time so that we can preserve the values from the first step.

```
x_minus = [q_plus; dq_plus(1:3)];

swap = [0 1 0 0 0 0; ...
        1 0 0 0 0 0;...
        0 0 1 0 0 0;...
        0 0 0 0 1 0;...
        0 0 0 1 0 0;...
        0 0 0 0 0 1];

x_plus = swap*x_minus;
[tout_next,xout_next] = ThreeLinkWalker_ODE45(t_end,x_plus);
```

Finally, we want to update what p_foot should be during this second step but call it p_foot_next so that we preserve our first step's values. The location of the new stance foot p_foot_next in the second step is equal to the location of the final position of the swing foot from the first step. Like we did for the first step, we want to use the repmat function to create a row vector of the p_foot so that we have a stance position value for each data point.

```
[p_st_foot,p_hip,p_torso,p_sw_foot] = Points_ThreeLinkWalker(q_minus,dq_minus);
p_foot_next0 = p_sw_foot;
p_foot_next = repmat(p_foot_next0', size(tout_next));
```

Now, we want to combine our values from the first and second step into single variables. Don't forget that for the time vector, we need to add the value from time at the end of the first step to all of the values in this next step so that the overall time continues (rather than just reset back to 0 at the beginning of the next step).

```
%% Append values
tout = [tout; tout_next+tout(end)];
xout = [xout; xout_next];
p_foot = [p_foot; p_foot_next];
```

Finally, we can use the combined variable definitions to call our animation function.

```
%% Animate
pause(1)
ThreeLinkWalker_anim(tout,xout,0.01,p_foot);
```

6.3 You did it! You can take two steps!

That's it! Your code is ready to run! Go to ThreeLinkWalker_sim.m, hit "Run" and watch your biped take two steps!

6.4 Taking steps 3, 4, 5, 6, ...

Alrighty, so you got your biped to take two steps. But what about three steps? Four? Five? The good news is that you do not need to change much at all! In fact, you will only be making three minor changes to your ThreeLinkWalker_sim.m code.

The first change is to create a new variable numSteps and set that equal to the number of steps you want to take minus one (since the first step will not be included). You should define this variable right at the beginning of the "Second Step" section in your code (and by the by, you can rename this section "Next Steps").

```
numSteps = 10;
```

Next, you are going to create a `for` loop and have it iterate from 1 to `numSteps`. It will encompass all of your code for the calculation of "Next Step." So, the `for` loop will start right after you define `numSteps` and end right before the "Animate" section.

```
for j = 1:numSteps
    ...
end
```

Finally, within that `for` loop where you are defining `p_foot_next0`, you are going to change that to equal to the swing foot position **plus** the position of the stance foot from the last step taken – don't forget to transpose the `p_foot(end,:)` vector.

```
p_foot_next0 = p_sw_foot + p_foot(end,:)' ;
```

Boom! That's it! You're done! Watch that beauty run (err...walk)!

Something that you may notice as you make your biped take multiple steps (especially upwards of like 6 or so) is that the legs appear to sink further into the ground with each subsequent step. A possible explanation of this is that the biped is ending each step with state variable values that result in the biped being in a lower position than it was when it first started the step. To resolve this issue, you need to implement a constraint on your walker such that it begins and ends each step with the same state variable values. This requires setting up an optimization problem which we will discuss in a later section.

7 Extracting useful Information from our Model

Now that you got your biped to take multiple steps, it is time that we look at how we might make its movements more efficient. For example, how can we optimize the torque input that we use for our motors? We will look into optimization in the next section. Before we can do optimization, we need to know what we are optimizing. We also need to be able to better visualize our results. The simulation is nice, but plots of useful variable data would be better.

In this section, we will be extracting our torque input u (Eq. 3.6), virtual constraints h (Eq. 3.15), and ground reaction forces F. Extracting u and h from our simulation data is fairly straight forward since our dynamic function `f` (which you will recall is our Matlab function that is appended to the end of `ThreeLinkWalker_ODE45.m`) already calculates these values. So getting those values is a matter of coding manipulation. Finding F however requires a little bit more of calculation on our part.

7.1 Computing Ground Reaction Forces

There are actually a few ways we can compute the ground reaction forces. I will propose two ways in this subsection.

7.1.1 Method 1: Using the Robot Equations from the Floating Base Model

This first method of computing ground reaction forces uses Eq. 5.1. Everything looks the same except instead of looking at the ground reaction forces of the swing leg as it impacts the ground, we will like to look at the ground reaction forces acting on the stance leg throughout a step. So the equation becomes

$$D_e\ddot{q}_e + C_e\dot{q}_e + G_e = \Gamma_e = B_e u + J_{st}^T F_{st} \tag{7.1}$$

where you will notice we have just a $J_{st}^T F_{st}$ term rather than also including a $J_{sw}^T \delta F_{sw}$ term. We are looking at just the stance leg instead of also the swing leg because the ground reaction forces are acting only on the stance leg **during** a step.

We also want to look at the acceleration of the stance foot. First, recall that we established the position of the stance foot to be

$$p_{foot}^{st} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

So we can define the jacobian of the stance foot J_{st} to be

$$J_{st} = \frac{\partial p_{foot}^{st}}{\partial q_e} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.2}$$

where we use q_e since we are in the extended coordinates.

Since the stance foot is not moving during the step, that means that its velocity is zero. Thus,

$$\begin{aligned} \dot{p}_{foot}^{st} &= \frac{d}{dt} p_{foot}^{st} && (7.3) \\ &= \frac{\partial p_{foot}^{st}}{\partial q_e}\dot{q}_e && (7.4) \\ &= J_{st}\dot{q}_e = \begin{bmatrix} 0 \\ 0 \end{bmatrix} && (7.5) \end{aligned}$$

Consequently, the acceleration of the stance foot would also equal to zero. Thus,

$$\ddot{p}^{st}_{foot} = \cancelto{0}{\frac{d}{dt}(J_{st})}\dot{q}_e + J_{st}\ddot{q}_e = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{7.6}$$

We see that the time derivative of J_{st} term cancels to 0 since J_{st} is a matrix of 0's and 1's (as shown in Eq. 7.2) and the derivative of these constants are all simply 0. Thus we are left with

$$J_{st}\ddot{q}_e = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{7.7}$$

We can rearrange Eq. 7.1 to solve for $\ddot{q}_e$

$$\ddot{q}_e = D^{-1}\left(-C_e\dot{q}_e - G_e + B_e u + J^T_{st}F_{st}\right) \tag{7.8}$$

Plugging in Eq. 7.8 into Eq. 7.7 we get

$$J_{st}D^{-1}\left[-C_e\dot{q}_e - G_e + B_e u + J^T_{st}F_{st}\right] = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{7.9}$$

Starting to rearrange things to ultimately solve for F_{st} (factor the $J_{st}D^{-1}$ on the F_{st} term and then move the other terms to the right hand side), we see

$$\left(J_{st}D^{-1}J^T_{st}\right)F_{st} = -J_{st}D^{-1}\left(-C_e\dot{q}_e - G_e + B_e u\right) \tag{7.10}$$

and finally we multiply by the inverse of the terms in front of F_{st} on both sides to get

$$F_{st} = \left(J_{st}D^{-1}J^T_{st}\right)^{-1}\left[-J_{st}D^{-1}\left(-C_e\dot{q}_e - G_e + B_e u\right)\right] \tag{7.11}$$

Thus, you have solved for the ground reaction forces on the stance foot.

7.1.2 Method 2: Using Center of Mass

Alternatively, you can solve for the ground reaction forces on the stance foot during a step by using the center of mass of the robot. Using your knowledge on mechanics (yes I know that I am doing some hand waving here because I do not quite understand the physics behind this, so you are going to have to trust me here), the ground reaction forces can be solved by the following way

$$F_{st} = M_{total}\ddot{p}_{cm} + \underbrace{M_{total}\begin{bmatrix}0\\g\end{bmatrix}}_{\text{effects due to gravity}} \tag{7.12}$$

where M_{total} is the total mass of the robot, $\ddot{p}_{cm}$ is the acceleration of the center of mass of the robot, and that final term is the "effects due to gravity."

For our simple 3-link robot,

$$M_{total} = 2m + M_H + M_T \tag{7.13}$$

The position of the center of mass p_{cm} of a robot can be calculated as follows

$$p_{cm} = \left(\sum_{i=1}^{N} p_i m_i\right)\frac{1}{M_{total}} \tag{7.14}$$

where p_i is the position coordinates of the corresponding mass at m_i. So for our robot,

$$p_{cm} = \left(p^{st}_{knee}m + p^{sw}_{knee}m + p_{hip}M_H + p_{torso}M_T\right)\frac{1}{M_{total}} \tag{7.15}$$

The velocity $\dot{p}_{cm}$ can be calculated as follows

$$\dot{p}_{cm} = \frac{\partial p_{cm}}{\partial q}\dot{q} = J_{cm}\dot{q} \tag{7.16}$$

The acceleration $\ddot{p}_{cm}$ can be calculated as follows

$$\ddot{p}_{cm} = J_{cm}\ddot{q} + \left[\frac{\partial}{\partial q}\left(J_{cm}\dot{q}\right)\right]\dot{q} \tag{7.17}$$

Thus, using these variable definitions you can compute the ground reaction forces using Eq. 7.12.

7.2 Using Matlab to solve for useful variables and plotting them

With our calculations for our ground reaction forces on the stance foot during a step F_{st} done, we can use these formulas to plug into Matlab so we can solve for them. We will also be extracting values for our torque inputs u and virtual constraints h.

The Matlab files that we will be modifying to accomplish this are the following:

- `SymbolicModelForThreeLinkRobot.m` – Add symbolic calculations for position, velocity, and acceleration at the center of mass.
- `dyn_mod_ThreeLinkWalker.m` – Add the output arguments for the numerical values of position, velocity, and acceleration at the center of mass.

- ThreeLinkWalker_ODE45.m – Include ground reaction force calculations in the f function. Add output arguments for ground reaction forces, torque inputs, and virtual constraints.

- ThreeLinkWalker_sim.m – Generate plots for ground reaction forces, torque inputs, and virtual constraints.

7.2.1 SymbolicModelForThreeLinkRobot.m

The first modifications to our code is to include our new calculations for ground reaction forces F_{st} that we defined in the previous subsection. We will be using Method 2 in our Matlab implementation, therefore, our first objective is to add symbolic calculations for the center of mass.

In your SymbolicModelForThreeLinkRobot.m file, after your position vector definitions section, you will add a new section called "Center of Mass Calculations." Using Eq. 7.15, Eq. 7.16, and Eq. 7.17 you will create equations for position (pcm), velocity (vcm), and acceleration (acm) at the center of mass respectively. Do not forget to create new symbolic variables ddth1, ddth2, and ddth3 so you can define the acceleration of the generalized coordinate vector $\ddot{q}$ (ddq).

```
%% Center of Mass Calculations
% position
pcm = (p_st_knee*m + p_sw_knee*m + p_hip*M_H + p_torso*M_T)/(2*m+M_H+M_T);

% velocity
Jcm = jacobian(pcm,q);
vcm = Jcm*dq;

% acceleration
syms ddth1 ddth2 ddth3
ddq = [ddth1; ddth2; ddth3];
acm = Jcm*ddq + (jacobian(Jcm*dq,q))*dq;
```

7.2.2 dyn_mod_ThreeLinkWalker.m

So now that we have the symbolic definitions of our center of mass calculations, we want to add them to our dynamics model code so that we can use them to generate numerical values.

The first thing that you are going to want to do is add the output arguments for pcm, vcm, and acm. You will also add an input argument for $\ddot{q}$ (ddq).

```
function [D,C,G,B,pcm,vcm,acm]= dyn_mod_ThreeLinkWalker(q,dq,parameters,ddq)
```

Now, go to the end of your code, all the way after where you define the B matrix. Here you will first define zero vectors for `pcm`, `vcm`, and `acm`. All the vectors should be $2x1$.

```
% Center of Mass
pcm = zeros(2,1);
vcm = zeros(2,1);
acm = zeros(2,1);
```

Next, we are going to use an `if statement`. If this function is called with more than 3 input parameters (meaning that we are given a `ddq`), then we will redefine `pcm`, `vcm`, and `acm` to be those equations generated from the `SymbolicModelForThreeLinkRobot.m` file. Make sure to also define the variables `ddth1`, `ddth2`, and `ddth3` using the values from the input vector `ddq`.

```
if nargin > 3

    ddth1 = ddq(1);
    ddth2 = ddq(2);
    ddth3 = ddq(3);

    pcm(1,1) = ...
    pcm(2,1) = ...

    vcm(1,1) = ...
    vcm(2,1) = ...

    acm(1,1) = ...
    acm(2,1) = ...

end
```

7.2.3 `ThreeLinkWalker_ODE45.m`

Now to actually implement the calculation for F_{st} as well as extract values for the ground reaction forces, torque inputs, and virtual constraints from our dynamics equation.

First go to where you define your `f` Matlab function at the end of the code. After where you define `dx` at the very end, you will do your calculations for ground reaction forces F_{st} (which we will call `GRF` in Matlab) using Eq 7.12. You will need to call `dyn_mod_ThreeLinkWalker.m`, but this time make sure to include the `ddq` input argument which is simply the last three values in your `dx` vector.

```
% Ground Reaction Forces
[~,~,~,~,pcm,vcm,acm]= dyn_mod_ThreeLinkWalker(x(1:3),x(4:6),parameters,dx(4:6));
GRF = Mtotal*acm + Mtotal*[0;g];
```

Now for the extracting part. So Matlab's ode45 solver does compute values for input torques u, virtual constraints h, and ground reaction forces F_{st} corresponding to each solution value in the xout matrix, but it does not store these values anywhere which is a shame. The only way to retrieve these values is to take the solutions that were found using the ode45 solver and throw them back into our f function. So let's do that!

First, modify your output arguments for f to include u, h, and GRF.

```
[dx,u,h,GRF]=f(t,x,stepSizeAngle,Kp,Kd)
```

Note that Matlab's ode45 solver will ignore these additional output variables. So we need to retrieve them ourselves.

After where you call the ode45 function in ThreeLinkWalker_ODE45.m, you will use a for loop to iterate through each time value tout (or solution value xout) **minus** 1. We subtract 1 because the last solution of the first step becomes the first solution value of the next step (and so on for subsequent steps). Since we do not want to have repeated points, we simply trim this last point.

Inside the for loop, you will define the time t and solution vector x at the current point and use those as input parameters when you call your f function. Note that your input values for stepSizeAngle, Kp, and Kd will remain constant. You will store the torque inputs, virtual constraints, and ground reaction forces for each solution value in vectors uout, hout, and GRFout respectively.

```
uout = [];
hout = [];
GRFout = [];
for i = 1:length(tout)-1
    t = tout(i);
    x = xout(i,:)';
    [~,uout(:,i),hout(:,i),GRFout(:,i)] = f(t,x,stepSizeAngle,Kp,Kd);
end
```

Finally, update the output arguments for ThreeLinkWalker_ODE45.m to include uout, hout, and GRFout.

```
function [tout, xout, uout, hout, GRFout]=ThreeLinkWalker_ODE45(t_end,x0)
```

7.2.4 ThreeLinkWalker_sim.m

Now to plot these useful parameters. First, you need to update your code to make sure that you are getting values for uout, hout, and GRFout. In the "First Step" section, update your call to ThreeLinkWalker_ODE45.m to include these new variables.

```
[tout, xout, uout, hout, GRFout]=ThreeLinkWalker_ODE45(t_end,x0);
```

Do the same when you make the call to ThreeLinkWalker_ODE45 in the "Next Steps" section.

```
[tout_next,xout_next, uout_next, hout_next, GRFout_next] = ...
    ThreeLinkWalker_ODE45(t_end,x_plus);
```

Finally, in the part of the code where you are appending the values for your "next steps" to the values for your "first step," make sure to do this appending process for your uout, hout, and GRFout vectors.

```
uout = [uout uout_next];
hout = [hout hout_next];
GRFout = [GRFout GRFout_next];
```

Now you can add a section for plotting after your "Next Steps" section. Feel free to plot whatever you like here. The following code is just a suggestion. Something to note when plotting, however, is that if you intend to plot uout, hout, or GRFout against time tout, be sure to **exclude** the last 1+numSteps from the tout vector or you will get an error. This is because when we defined uout, hout, and GRFout in ThreeLinkWalker_ODE45.m, we excluded the last point in the solution vector to avoid repeated points.

```
%% Plots
figure
subplot(2,1,1);
plot(tout(1:end-numSteps-1),hout(1,:),'LineWidth',2);
title("\theta_3 - \theta_3(des) Virtual Constraint")
xlabel('time (s)')
ylabel('error (rad)')
subplot(2,1,2);
plot(tout(1:end-numSteps-1),hout(2,:),'LineWidth',2);
title("\theta_1 + \theta_2 Virtual Constraint")
xlabel('time (s)')
ylabel('error (rad)')
```

```
figure
plot(tout,xout(:,[1 2 3]),'LineWidth',2)
title('Time versus Generalized Positions')
xlabel('time (s)')
ylabel('Angle (rad)')
legend('\theta_1','\theta_2','\theta_3')

figure
plot(tout,xout(:,[4 5 6]),'LineWidth',2)
title('Time versus Generalized Velocities')
xlabel('time (s)')
ylabel('Angular Velocity (rad/s)')
legend('d\theta_1','d\theta_2','d\theta_3')

figure
plot(tout(1:end-numSteps-1),uout,'LineWidth',2)
title('Time versus Torque Input')
xlabel('time (s)')
ylabel('torque input (Nm)')
legend('u_1','u_2')

figure
subplot(2,1,1);
plot(tout(1:end-numSteps-1),GRFout(1,:),'LineWidth',2);
xlabel('time (s)')
ylabel('Force (N)')
title("Ground Reaction Forces in X Direction")
subplot(2,1,2);
plot(tout(1:end-numSteps-1),GRFout(2,:),'LineWidth',2);
ylabel('Force (N)')
title("Ground Reaction Forces in Y Direction")
```

8 Optimization

Ooo, this is exciting! You got your three-link biped taking a bunch of steps and you know how to monitor some key output variables. What more could you ask for?! It turns out...quite a bit more, actually. Our biped walker is disappointingly inefficient. The motors are working overtime to produce motion on a rigid trajectory.

Take a look at the torso link as your biped takes multiple steps. You will notice that it snaps back to the desired position (θ_3^d) instantly as it enters another step. Your legs are also following a rigid path along which your motors are providing continuous torque to keep in line. What if we let our torso link bob up and down with each step instead of having it snap back in place? What if we also took advantage of gravity to help alleviate some of the work that our motors have to put into the system?

What if we were able to somehow set an ... **optimal** path along which our biped could follow to achieve these goals?

Ladies and gentlemen, that is exactly what we are going to be doing in this section!

8.1 Introducing the Cubic Function to the Virtual Constraints

So to loosen up our biped a bit so that it is not stuck doing rigid motions, we need to take a look at what's making it perform so rigidly in the first place. Remember those virtual constraints that we defined when we set up our controller (see Eq. 3.15)? Yup, that is the culprit.

The problem is that we are forcing our "desired" positions to be held at a constant point. For example, θ_3 is being controlled to be held at a constant position θ_3^d. The same goes for our legs. We set up the constraints such that the angles of each leg are to be equal and opposite. That's quite the rigid setup.

So let's loosen things up a bit. Let's make it such that the desired positions can be defined by cubic functions, rather than be set to a rigid point. And let's let those cubic functions be a function of the stance leg angle θ_1. Since we have two virtual constraints (one for the torso position and one for the stance legs), we will have two different cubic functions, one for each constraint.

Let's start with the torso constraint. The function will be of the form

$$y_1 = \theta_3 - \theta_3^d(\theta_1) \tag{8.1}$$

where the function for the first virtual constraint y_1 is defined as setting our torso link angle θ_3 to equal to a desired angle $\theta_3^d(\theta_1)$ (which is a function of the stance leg angle θ_1). The θ_3 will stay as itself, but how do we define the desired angle $\theta_3^d(\theta_1)$? Well, how is any cubic function define? It is simply a series of exponentials of the dependent variable (which in our case is θ_1) being raised to the 0^{th} power all the way to the 3^{rd} power which are all being added to each other. For each power, there is also a corresponding coefficient. Thus,

$$\theta_3^d(\theta_1) = a_{1,0} + a_{1,1}(\theta_1) + a_{1,2}(\theta_1)^2 + a_{1,3}(\theta_1)^3 \tag{8.2}$$

where $a_{1,0}, a_{1,1}, a_{1,2}$, and $a_{1,3}$ are coefficients.

A little note on my notation here. I am defining my coefficients $a_{1,0}, a_{1,1}, a_{1,2}$, and $a_{1,3}$ where the first number in the subscript designates the virtual constraint to which the coefficient belongs to. So in this case, the first number is 1 since these coefficients are all for the first virtual constraint y_1.
The second number in the subscript denotes where that coefficient falls within the cubic function. For example, $a_{1,0}$ is the coefficient that is multiplied by the dependent variable (θ_1) raised to the 0^{th} power (ie $a_{1,0}(\theta_1)^0 = a_{1,0}$). Similarly, $a_{1,1}$ is the coefficient that is multiplied by the dependent variable (θ_1) raised to the 1^{st} power (ie $a_{1,1}(\theta_1)^1 = a_{1,1}(\theta_1)$). And so on.

Now for the second constraint. The form of this virtual constraint is similar to the first. We define the second virtual constraint y_2 such that

$$y_2 = \theta_2 - \theta_2^d(\theta_1) \tag{8.3}$$

where we are constraining the angle of the swing leg θ_2 to be defined by a desired angle $\theta_2^d(\theta_1)$ (which is a function of θ_1).

We still want to maintain the mirror law at the beginning and end of each step, meaning that we want each step to begin and end such that the angles for the stance and swing legs are equal and opposite. Everywhere else should be represented by a cubic function. To do this, we must define θ_2^d to be

$$\theta_2^d(\theta_1) = -\theta_1 + \left[a_{2,0} + a_{2,1}(\theta_1) + a_{2,2}(\theta_1)^2 + a_{2,3}(\theta_1)^3\right]\left(\theta_1 + \theta_1^d\right)\left(\theta_1 - \theta_1^d\right) \tag{8.4}$$

where $a_{2,0}, a_{2,1}, a_{2,2}$, and $a_{2,3}$ are all coefficients (with the same notation format as with the first virtual constraint) and θ_1^d is the desired angle for the stance leg.

The last two terms that is multiplied by the cubic function part of this equation $\left(\theta_1 + \theta_1^d\right)\left(\theta_1 - \theta_1^d\right)$ is how we ensure the mirror law is achieved at the start and end of each step. At the beginning and end of each step, the stance leg angle θ_1 will be equal to some desired value θ_1^d. Once that happens, one of those multiplied terms will equal to 0, thus reducing the entire equation down to $\theta_2^d(\theta_1) = -\theta_1$. Plug that into the virtual constraint y_2 and we get $y_2 = \theta_2 - \theta_1$, which is the mirror law.

Boom! There's our new virtual constraints! To be explicitly clear, I am redefining my virtual constraint matrix $h(x)$ to be

$$h(x) = \begin{bmatrix} \theta_3 - \theta_3^d(\theta_1) \\ \theta_2 - \theta_2^d(\theta_1) \end{bmatrix} = \begin{bmatrix} \theta_3 - \left[a_{1,0} + a_{1,1}(\theta_1) + a_{1,2}(\theta_1)^2 + a_{1,3}(\theta_1)^3\right] \\ \theta_2 - \left[-\theta_1 + \left[a_{2,0} + a_{2,1}(\theta_1) + a_{2,2}(\theta_1)^2 + a_{2,3}(\theta_1)^3\right]\left(\theta_1 + \theta_1^d\right)\left(\theta_1 - \theta_1^d\right)\right] \end{bmatrix} \tag{8.5}$$

8.2 Defining the Optimization Problem

Now you may be wondering, "hey, where does optimization fit into all of this?" To which I ask you this question in response: how do we define values for the coefficients $a_{1,0}, a_{1,1}, a_{1,2}, a_{1,3}, a_{2,0}, a_{2,1}, a_{2,2}$, and $a_{2,3}$? They could be anything really. There are a bajillion different combinations that we could try. But somewhere among all of those combinations is a set of values for those coefficients that would result in the least amount of torque input compared to the rest of the combinations.

... you can probably see where I am going with this ...

Yes, we want to set up an optimization problem such that our objective function is minimizing torque input and the parameters that are being optimized are those cubic function coefficients. For clarity, let's stick those coefficients into a vector and call that vector a:

$$a = \begin{bmatrix} a_{1,0} \\ a_{1,1} \\ a_{1,2} \\ a_{1,3} \\ a_{2,0} \\ a_{2,1} \\ a_{2,2} \\ a_{2,3} \end{bmatrix} \tag{8.6}$$

NOTE: I am defining a as a vector rather than the matrix

$$a = \begin{bmatrix} a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

because when we go to using the function `fmincon` in Matlab, everything will have to be defined in vector form for syntax.

It is all the same, really. I just want to be consistent with my notation.

And just for the heck of it, I am also going to throw in a nonlinear constraint! I want to make the steps periodic, meaning that I want the biped to end a step with the exact same q and $\dot{q}$ values as it did for the beginning of the step. This constraint will allow our biped to essentially walk forever.

So the overall optimization problem will look like this

$$\begin{aligned} \underset{\text{cost}(a)}{\text{minimize}} \quad & \text{cost}(a) = \int_0^{t_f} u^T(\tau) u(\tau) d\tau \\ \text{subject to} \quad & \left|\left| x_o - \Delta(x_f) \right|\right| = 0 \end{aligned} \tag{8.7}$$

where u is the torque input, τ is time, x_o are the initial x values (recall that $x = [q; \dot{q}]$) and $\Delta(x_f)$ are the x values after impact and after swapping legs ("impact" is designated by the Δ).

Something that is not immediately obvious by this setup is that in addition to optimizing a, we are also optimizing the initial conditions x_o. This is because of the nonlinear constraint that we added to the problem. Since that constraint depends on x_o, we must also look for an optimal set of x_o that satisfies the optimization problem. If you neglect to try to optimize x_o using the function `fmincon` in Matlab, you will likely run into Matlab saying that it is unable to converge to a feasible solution. I learned this the hard way.

8.3 Setting up the Optimization Problem in Matlab

And now for the fun part: incorporating the optimization problem in Matlab. My advice would be to create a duplicate folder containing of all the Matlab files that we have created so far and name the new folder something like "Optimization 3-LinkWalker" or something, assuming that the original folder is called "3-LinkWalker." This way you won't lose your pre-optimization code.

Anyway, we will be introducing two new Matlab functions

- `costFunction.m` – Defines the optimization cost function.
- `myconstraints.m` – Defines the nonlinear constraints.

Additionally, we will also be modifying the following Matlab files:

- `SymbolicModelForThreeLinkRobot.m` – Update definition of `h`.
- `ControlFunctions.m` – Update `h` and Lie derivative definitions. Change `th3desired` to `th1desired`.
- `ThreeLinkWalker_ODE45.m` – Separate the `f` function into its own Matlab file (which we will call `ThreeLinkWalker_dynamics.m`. Create variable `dt` which will be used in `costFunction.m`. Replace `th3desired` and `stepSizeAngle` with `th1desired` (and move where you definie it to `ThreeLinkWalker_sim`. Move all definitions of "options" to `ThreeLinkWalker_sim.m` file.
- `ThreeLinkWalker_sim.m` – Define "options" (which was previously defined in `ThreeLinkWalker_ODE45.m`). Define `a0` (initial values for vector a). Define `th1desired`. Perform optimization problem using `fmincon`. Apply optimized values of a and x_o to calculations for "First Step" and "Next Step." Update calls to `ThreeLinkWalker_ODE45.m` to include "option" input parameter.
- `ThreeLinkEndStepEvents.m` – Replace `stepSizeAngle` with `th1desired`.

Let's modify/add to the code in the order that it will actually makes sense in.

8.3.1 `SymbolicModelForThreeLinkRobot.m`

This is the perfect place to start. The very first thing that we did in this optimization section was update our virtual constraints $h(x)$, so it makes sense that we want to update our symbolic notation to incorporate this new definition.

The only modification that we are making to this code is changing $h(x)$ to be the new $h(x)$ that we defined in Eq. 8.5. Do not forget to create new symbolic variables for each of cubic coefficients as well as for θ_1^d.

```
syms a10 a11 a12 a13 a20 a21 a22 a23 th1desired

h0 = [th3; th2];
hd = [a10 + a11*th1 + a12*th1^2 + a13*th1^3; ...
    -th1 + (a20 + a21*th1 + a22*th1^2 + ...
    a23*th1^3)*(th1+th1desired)*(th1-th1desired)];
h = h0 - hd;
```

8.3.2 `ControlFunctions.m`

Since we updated the symbolic definition of $h(x)$, we now need to make sure to update all the places where $h(x)$ (as well as other function dependent on $h(x)$) is defined. The only place where this happens is in `ControlFunctions.m`.

Here, you want to first update your input arguments. We no longer use θ_3^d. Instead we use θ_1^d (since our control functions are now a function of the stance leg angle). Additionally, we need to add the vector a to our input arguments.

```
function [h,Lfh,L2fh,LgLfh] = ControlFunctions(q,dq,parameters,th1desired,a)
```

Next, where you are defining your inputs at the top of the code, you need to also add definitions for each of the individual cubic coefficient variables.

```
a10 = a(1);
a11 = a(2);
a12 = a(3);
a13 = a(4);
a20 = a(5);
a21 = a(6);
a22 = a(7);
a23 = a(8);
```

Finally, run `SymbolicModelForThreeLinkRobot.m` and update your matrix definitions for `h,Lfh,L2fh,` and `LgLfh`.

8.3.3 `ThreeLinkWalker_sim.m`

This is where things gets interesting. We will perform and implement the optimization problem using `fmincon` in this top level code. I strongly recommend reviewing [8] if you are unfamiliar with how `fmincon` works in Matlab.

As with all optimization problems, you need a set of initial conditions to start off the optimization. Since we are optimizing both the vector a and initial conditions x_o, that means that our overall initial condition vector will consist of both of these vectors. We will define this initial condition vector as `conditions0`.

NOTE to clarify a possible point of confusion: I talked about two **different** types of "initial conditions" in the previous paragraph: a set of initial conditions for my optimization problem (`conditions0`) and the set of initial conditions that describe the biped walker (x_o).
All optimization problems require you to define initial conditions for the parameters that you are optimizing. It just so happens that some of those parameters that I am optimizing are the initial conditions that describe the initial states for the biped walker ($x_o = [q;\dot{q}]$). In other words, since I am optimizing a and x_o, I need to define initial values for a and x_o (which we will call a_o and x_{o_o} respectively) in a single vector which I will call `conditions0`, where

$$\texttt{conditions0} = \begin{bmatrix} x_{o_o} \\ a_o \end{bmatrix} \tag{8.8}$$

Let's keep our definition of `x00` the same as we had for the definition of `x0` in our original code. After defining `x00`, we will define `a0` and `conditions0` and then leave your definition of `t_end` as is. Furthermore, we will choose to define θ_1^d here as well. Normally, we would define this in the `f` function inside of `ThreeLinkWalker_ODE45.m` file, but we shall move this variable definition to this file to help streamline the code a bit more and consolidate the places in the code where there are variables that we may choose to modify.

```
x00=[-pi/8;pi/8;pi/6;1.5;-1.5;0]; %[th1, th2, th3, dth1, dth2, dth3]
a0 = [pi/6 0 0 0 0 0 0 0]'; %[a10 a11 a12 a13 a20 a21 a22 a23]
conditions0 = [x00; a0];

th1desired = pi/8;
t_end=1; % length of simulation
```

These initial conditions for `a0` were given to me by Prof. Jessy Grizzle.

Next, we are going to want to define our "options" variable for `ode45` and the `Events` function. In reality, we are simply moving where we defined these options in the

`ThreeLinkWalker_ODE45.m` file to this file. The reason we are doing this is in an effort to make our code a bit more efficient. There is no need to keep defining our options every time we call `ode45` – which, as a result of our optimization implementation method, will be quite a lot.

```
% set up options for event function
refine=4;
RelTol = 10^-4;
AbsTol = 10^-5;

% options withOUT event
options = odeset('Refine',refine, 'RelTol',RelTol,'AbsTol',AbsTol);

% options for with impact function
options = odeset('Events', @ThreeLinkEndStepEvents,'MaxStep',0.01);
```

Now, as you will hopefully begin to see as we implement this optimization strategy, this particular optimization method is going to take quite a bit of time to compute. Therefore, the purpose of this next set of code is to help reduce the computation time a bit as well as help visualize the calculations in real-time to monitor its status.

The first bit of code is to take advantage of your computer's hardware. If you have multiple cores like I do, you can use the `parpool` command to ensure that Matlab takes advantage of that. Before calling `parpool`, you will need to write `delete(gcp('nocreate'))`. Don't ask me why though – Prof. Grizzle gave me this hint. I am putting both of these command lines in an `if statement` because really you need to just run this bit of code once per coding session. So after running those two lines of code once, ignore them for subsequent runs of your code.

```
% Parallel stuff
if 0
delete(gcp('nocreate'))
parpool
end
```

Now for the visualization part. As I mentioned before, it is going to take quite a bit of time for our code to run, during which, it is going to be impossible to determine how our optimization is doing. What if the optimization problem blows up? We will not be able to tell except noticing that the code runs forever – not a very efficient method of tracking this. Thus, this next bit of code is defining a set of options that we are going to use when we call `fmincon`. It will plot our cost function value for each iteration of the optimization problem. You could also choose to plot your max constraint at each iteration using the same `PlotFcns` option. However you cannot do both. So pick one and comment out the other. I have also inserted some other options that do other stuff. Feel free to check out the MathWorks website to get a better understanding of the other options.

```
% define options for fmincon
options_fmin=optimset('fmincon');
options_fmin = optimset(options_fmin,'MaxFunEvals',3e6,'TolFun',1e-4,'TolCon',...
    1e-4,'TolX',1e-7,'Algorithm','sqp');
options_fmin = optimset(options_fmin,'PlotFcns',...
    'optimplotfval'); % display function value @ each iteration
% options_fmin = optimset(options_fmin,'PlotFcns', ...
    'optimplotconstrviolation'); % display max constraint violation
options_fmin = optimset(options_fmin,'UseParallel',1,'MaxIter',2e3);
```

Now we can do the actual optimization stuff. To make our code look better, we will use anonymous functions (see [9]) for our calls to `costFunction.m` (the cost function that we are minimizing) and `myconstraints.m` (the nonlinear constraints) (NOTE: we will define both of these functions later in this section) which we need when we use `fmincon`.

Note that I am defining the parameters that we are optimizing with the vector `conditions`, where

$$\texttt{conditions} = \begin{bmatrix} x_o \\ a \end{bmatrix} \tag{8.9}$$

I also want to define a set of upper and lower bounds for my optimization problem. After playing around a bit, I found some bounds that work quite nicely. Finally, I make the actual call to `fmincon` and use its results to define `conditions`.

```
% optimize
cost = @(conditions) costFunction(conditions,t_end,options,th1desired);
con = @(conditions) myconstraints(conditions,t_end,options,th1desired);
ub = [pi/4;pi/4;pi/4;4;4;4;...
    20*[pi; pi; pi; pi; pi; pi; pi; pi]];
lb = [-pi/4;pi/16;pi/16;-4;-4;-4;...
    20*[-pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi]];
[conditions,fval,exitflag,output] = ...
fmincon(cost,conditions0,[],[],[],[],lb,ub,con,options_fmin)
```

If all goes well, we should get some optimized values for a and x_o. Therefore, we will use those values to call `ThreeLinkWalker_ODE45.m` with these optimized values to compute our final optimized trajectory. Something to note here is that during optimization, we are restricting Matlab to optimize just θ_1 and **not** θ_2. We are doing this because we want θ_1 and θ_2 to be equal and opposite (because we are enforcing the mirror law at the beginning of the step). Thus, we want $\theta_2 = -\theta_1$. This will become more apparent when we write the code for `costFunction.m` and `myconstraints.m` later on. But for now, realize that I write the `x0(2)=-x0(1)` in the following lines of code for this reason.

Something else to note in the following lines of code is that my input arguments for my call to `ThreeLinkWalker_ODE45.m` are slightly different because I am changing my code for this file (as you will see in the next subsection).

```
% Calculate optimize trajectory
x0 = conditions(1:6);
x0(2)=-x0(1);
a = conditions(7:end);
[tout, xout, uout, hout, GRFout] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired);
```

Everything else in your code for this file will stay largely the same. We just need to modify one last line. In your "Next Steps" section of your code, you need to modify you call to `ThreeLinkWalker_ODE45.m` to reflect the change of input arguments. Something to note here is that yes, we are using the exact same optimized parameters for a. However, the "initial condition" for the states of the biped walker q and $\dot{q}$ will be defined by the final values from the end of the last step after impact and after swapping legs. Why? Because the beginning of the next step is defined by how the last step ended – you cannot optimize this directly.

```
% calculate  trajectory of next step
[tout_next,xout_next, uout_next, hout_next, GRFout_next] = ...
    ThreeLinkWalker_ODE45(t_end,x_plus,a,options,th1desired);
```

8.3.4 `ThreeLinkWalker_ODE45.m`

The first thing that you are going to want to do is update your input and output arguments. Your input arguments need to now include an argument for `a` (since we use this to define our virtual constraints) as well as `options` and `th1desired` since we are moving them from being defined in this code to being defined in our top level code `ThreeLinkWalker_sim.m`. Our output arguments needs to include one more argument for `dt` which we will need in our `costFunction.m` file.

```
function [tout, xout, uout, hout, GRFout, dt] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired)
```

Next, you are going to want to remove the line of code that defines stepSizeAngle. In reality, we are now defining "stepSizeAngle" as th1desired since these two variables serve the exact same purpose – they both define the final angle that the stance leg should be at the end of a step. So we will be replacing all instances of stepSizeAngle with th1desired.

Since we moved our definition for options to ThreeLinkWalker_sim.m, you may delete the definition that you have for it in this code.

Next, at the end of the code where you define the function f, remove your variable definition for th3desired. Then replace where you use th3desired for your input argument when you call ControlFunctions.m to th1desired (we no longer use θ_3^d for our control functions, and we define θ_1^d in the top level code). Additionally, in that call to ControlFunctions.m, you will need to add the input parameter a to your call.

```
[h,Lfh,L2fh,LgLfh] = ...
    ControlFunctions(q,dq,parameters,th1desired,a);
```

Next, you are going to completely extract your function f from this file and give it its own Matlab file. While we are at it, we will rename the function ThreeLinkWalker_dynamics. Therefore, the corresponding Matlab file will be ThreeLinkWalker_dynamics.m. Since the virtual constraints are dependent on a, you will need to add this as one of the input parameters to this function. Furthermore, you will replace the stepSizeAngle input parameter to th1desired.

```
function [dx,u,h,GRF] = ...
    ThreeLinkWalker_dynamics(t,x,th1desired,Kp,Kd,a)
```

Back to the ThreeLinkWalker_ODE45 file. In your call to the ode45 solver, replace the function f to ThreeLinkWalker_dynamics (since we changed the name). Also replace your input parameter stepSizeAngle with th1desired.

```
[tout, xout] = ode45(@ThreeLinkWalker_dynamics, [t_start t_end], ...
    x0, options, th1desired, Kp, Kd, a);
```

Next, we need to define the variable dt (in your ThreeLinkWalker_ODE45.m file, **not** the new ThreeLinkWalker_dynamics file). This variable is used in costFunction.m and it describes the the time between each solution point, the difference between the tout values. So, you will add a couple of lines of code in the section of the code where you do the reconstruction of uout, hout, and GRFout. Before the for loop, define dt as an empty vector.

```
dt = [];
```

In the `for loop`, define value of `dt` at point `i` as the difference between `tout` at point `i` minus `tout` at `i-1`.

```
dt(i) = tout(i)-tout(i-1);
```

Next, as you may have noticed, we need to change the start value of the `for loop`. Since `dt` is defined as the `tout` value at the current position minus the previous `tout` value, we will get an error at position `i = 1`. To fix this problem, we will start the `for loop` at `i = 2`.

```
for i = 2:length(tout)-1
```

Finally, also in the `for loop` you need to update your call to `f` to now be a call to `ThreeLinkWalker_dynamics`. Additionally, replace the input parameter `stepSizeAngle` to `th1desired`.

```
[~,uout(:,i),hout(:,i),GRFout(:,i)] = ...
ThreeLinkWalker_dynamics(t,x,th1desired,Kp,Kd,a);
```

8.3.5 `ThreeLinkEndStepEvents.m`

The changes to this code is simple. Since we replaced `stepSizeAngle` with `th1desired`, you are simply going to want to reflect those changes in this code. Thus, update your input parameter from `stepSizeAngle` to `th1desired`.

```
function [value, isterminal, direction] = ...
    ThreeLinkEndStepEvents(t, x, th1desired,Kp,Kd,a)
```

Finally, update the remaining two instances of `stepSizeAngle` to be `th1desired`.

8.3.6 `costFunction.m`

From Eq. 8.7 you see that our cost function is

$$\text{cost}(a) = \int_0^{t_f} u^T(\tau)u(\tau)d\tau$$

Rewriting this as a summation, we get

$$\text{cost}(a) = \sum_{k=1}^{\texttt{length(tout)-1}} u(k)^T u(k) dt(k) \tag{8.10}$$

where k refers to the k^{th} iteration of the optimization problem. So $u(k)$ is the resulting torque inputs along all time `tout` for the set of parameters `a` and `x0` corresponding to the k^{th} iteration of the optimization problem.

To write this cost function in a Matlab script, we first need to define the function along with its input and output arguments.

```
function [cost] = costFunction(conditions,t_end,options,th1desired)
```

Next, we need to break up the input argument conditions into separate vectors a and x0. Since we only want the optimization problem to optimize just the initial value for θ_1 rather than the initial values for both θ_1 and θ_2 (because we are enforcing mirror law at the beginning of the step), we will set the θ_2 initial term to equal to the opposite of the initial θ_1 term.

```
x0 = conditions(1:6);
x0(2)=-x0(1);
a = conditions(7:end);
```

Next, we make a call to ThreeLinkWalker_ODE45.m so we can see the resulting uout values for a given set of parameters a and x0.

```
[tout, xout, uout, hout, GRFout, dt] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired);
```

Finally, we use the results from ThreeLinkWalker_ODE45.m to do the cost function calculation that we established in Eq. 8.10.

```
cost_temp = [];

for k=1:length(uout)
    cost_temp(k) = uout(k)'*uout(k)*dt(k);
end

cost = sum(cost_temp);

end % end the function
```

8.3.7 myconstraints.m

Now to create the final function. We will define the nonlinear constraints as we established them in Eq. 8.7.

First, you need to set up your function definition as well as define your input and output arguments.

```
function [cin ceq] = myconstraints(conditions,t_end,options,th1desired)
```

Next, you need to use the conditions variable to define a and x0 (keeping in mind to set $\theta_2 = -\theta_1$ as we did in costFunction.m). Use these variables to call ThreeLinkWalker_ODE45.m.

```
x0 = conditions(1:6);
x0(2)=-x0(1);
a = conditions(7:end);

[tout, xout, uout, hout, GRFout, dt] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired);
```

Next, we need to use the last set of values from the calculated xout to find out the $x = [q;\dot{q}]$ values after impact and after swapping legs. Simply copy and past your code from ThreeLinkWalker_sim.m where you apply your impact and swap matrix.

```
% Define q and dq
q = xout(:,[1 2 3]);
dq = xout(:,[4 5 6]);

% Impact
q_minus = q(end,:)';
dq_minus = dq(end,:)';

[dq_plus, F_I] = Impact(q_minus,dq_minus);

% swap legs
q_plus = q_minus;

x_minus = [q_plus; dq_plus(1:3)];

swap = [0 1 0 0 0 0; ...
        1 0 0 0 0 0;...
        0 0 1 0 0 0;...
        0 0 0 0 1 0;...
        0 0 0 1 0 0;...
        0 0 0 0 0 1];

x_plus = swap*x_minus;
```

Finally, define your nonlinear constraints. We are only implementing an equality nonlinear constraint, so the inequality nonlinear constraint should just be an empty vector.

```
%% define constraints
```

```
cin = [];
ceq = norm(x0-x_plus);

end
```

8.3.8 You Did It!

Congratulations, you successfully implemented your first optimization code for the 3-link biped in Matlab. Go back to your `ThreeLinkWalker_sim.m` file and run it!

So what was the point of doing all of that optimzation stuff? At first glance, it probably appears that we didn't do much improvement. Indeed, if you use the same controller gains (K_p and K_d – which are determined by ω_n and ζ) as you used pre-optimization, you will notice that we have about the same error values but the motors are giving even more torque input. That sounds worse!
But, if you played around with you pre-optimization code you would notice that increasing your controller gains resulted in lower errors but would require an increasing amount of torque input. Using the optimization code that we just created, you will notice that increasing your gains not only decreases error but also decreases torque input (there's still a sweet spot to find though because torque input will start to increase after a point). As a reference, try comparing your non-optimized versus optimized virtual constraint error and torque input graphs with control parameters of $\omega_n = 17$ and $\zeta = 0.8$ versus $\omega_n = 70$ and $\zeta = 0.8$. See anything interesting?
Another plus of this new optimization code is that remember that problem that you may have encountered with your biped sinking into the floor with each subsequent step (especially with steps upwards of 6 or so)? Well, introducing that periodicity nonlinear constraint into our optimization problem resolved that issue! Your biped is now beginning and ending each step in the same position, and therefore it is staying on top of the ground!
Obviously, there is much to be desired in terms of more things we could do to further improve the biped walking. But clearly we are on the right path!

8.4 Reducing the Number of Optimization Parameters and Updating the Cost Function

So there is much to be desired with this optimization problem that we have set up. For one thing, it takes a while to run. The process is simply inefficient. We are having the code run that ode45 solver function so many times. The cost function calls it for each iteration of the optimization process. The constraints function also calls it. And then after the optimization problem has been finally solved, we call `ode45` yet again to establish the final trajectory which we plot and simulate. It is insane.

People way more smarter than me have developed software to make this process way more efficient, FROST being one of them. But this bootcamp is limited to just applications with standard Matlab functions, so we are stuck with this rather inefficient method of optimization.

But, there are still things we can do to improve the code! For one thing, you will notice that we are optimizing a grand total of 14 parameters – six for the initial conditions `x0` and eight for the cubic functions coefficients `a`.

What if I told you we could reduce the amount of parameters that we need to optimize from 14 to just 9? It is quite simple actually. So we still need to optimize the eight cubic function coefficients. However, we technically only need to know one of the six `x0` variables to define the rest of the `x0` variables. Thus, we would need to just optimize one (as opposed to six) `x0` variables.

Since we are dealing with initial condition variables `x0`, we know that time $t = t_o = 0$ at this point. We also know that with our virtual constraints, the control goal is to drive both constraints to equal zero. That is, we want

$$y_1 \equiv 0 \tag{8.11}$$

$$y_2 \equiv 0 \tag{8.12}$$

This means that at time $t_o = 0$, both y_1 and y_2 will both also equal to 0. Thus,

$$y_1(t_o) = 0 \tag{8.13}$$

$$y_2(t_o) = 0 \tag{8.14}$$

Consequently, the derivatives $\dot{y}_1$ and $\dot{y}_2$ will both also equal to 0 at this point (since the derivative of 0 is 0). Thus,

$$\dot{y}_1(t_o) = 0 \tag{8.15}$$

$$\dot{y}_2(t_o) = 0 \tag{8.16}$$

Recall our updated virtual constraints

$$h(x) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \theta_3 - \theta_3^d(\theta_1, a) \\ \theta_2 - \theta_2^d(\theta_1, a) \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

where both θ_3^d and θ_2^d are functions of both θ_1 and the cubic coefficient vector a. This means that if we know θ_1 at a time t and the vector a, we also would know both θ_3 and θ_2 at that same time t – they would equal to θ_3^d and θ_2^d at time t respectively.

We are most interested in the initial conditions, so the time t that we will look at is t_o. Thus,

$$y_1(t_o) = 0 \Rightarrow \theta_3(t_o) = \theta_3^d\left(\theta_1(t_o), a\right) \tag{8.17}$$

$$y_2(t_o) = 0 \Rightarrow \theta_2(t_o) = \theta_2^d\left(\theta_1(t_o), a\right) \tag{8.18}$$

And with the derivatives:

$$\dot{y}_1(t_o) = 0 \Rightarrow \dot{\theta}_3(t_o) = \left.\frac{\partial \theta_3^d}{\partial \theta_1}\right|_{\theta_1(t_o), a} \cdot \dot{\theta}_1(t_o) \tag{8.19}$$

$$\dot{y}_2(t_o) = 0 \Rightarrow \dot{\theta}_2(t_o) = \left.\frac{\partial \theta_2^d}{\partial \theta_1}\right|_{\theta_1(t_o), a} \cdot \dot{\theta}_1(t_o) \tag{8.20}$$

So our initial conditions vector x0 will look like

$$\begin{aligned}
\texttt{x0(1)} &= \theta_1(t_o) \\
\texttt{x0(2)} &= \theta_2^d\left(\theta_1(t_o), a\right) \\
\texttt{x0(3)} &= \theta_3^d\left(\theta_1(t_o), a\right) \\
\texttt{x0(4)} &= \dot{\theta}_1(t_o) \\
\texttt{x0(5)} &= \left.\frac{\partial \theta_2^d}{\partial \theta_1}\right|_{\theta_1(t_o), a} \cdot \dot{\theta}_1(t_o) \\
\texttt{x0(6)} &= \left.\frac{\partial \theta_3^d}{\partial \theta_1}\right|_{\theta_1(t_o), a} \cdot \dot{\theta}_1(t_o)
\end{aligned}$$

where you see we need to know just two variables θ_1 and $\dot{\theta}_1$ to define the rest of the variables.

"But, Wami. I thought you said that we were going to reduce the size x0 variables down to just one. It looks like we still need two here..."

Indeed! Excellent observation, my esteemed reader. We can, in fact, reduce the number of variables we need to just one. Let me ask you, what is the stance leg angle θ_1 at the end of a step? ... it is the same angle as it is at the beginning of the step, right? We established that when we imposed the mirror law in our second virtual constraint y_2.

Now, where have we defined what that final stance leg angle should be? Think way, way back to our first Matlab simulation.

Two words: `Event` function.

Indeed, we have already established what the final stance leg angle should be via the `Event` function. The final stance leg angle is equal to the initial stance leg angle at the beginning of the step. Thus, $\theta_1(t_o)$ is equal to that same angle that we defined for the `Event` function. In our new optimized code, we established that this `Event` function angle is also equal to the θ_1^d (`th1desired`) variable from our updated virtual constraints. Thus, in terms of the variable names that we have used in Matlab,

$$\texttt{x0(1)} = \theta_1(t_o) = \texttt{th1desired} \tag{8.21}$$

Therefore, the only `x0` variable that we need to optimize is $\dot{\theta}_1$. Thus, the updated conditions vector that we will be optimizing with `fmincon` will look like this:

$$\texttt{conditions} = \begin{bmatrix} \texttt{dth1} \\ \texttt{a} \end{bmatrix} \tag{8.22}$$

which consists of just nine variables (one from `dth1` and eight from `a`).

One more thing that I want to do before I show you how to update your Matlab code with this reduced number of optimization parameters is update your cost function.

The current cost function looks like this:

$$\text{cost}(a) = \int_0^{t_f} u^T(\tau)u(\tau)d\tau$$

I want to change it to look like this:

$$\text{cost}(a) = \int_0^{t_f} u^T(\tau)u(\tau)d\tau + \gamma\int_0^{t_f} y^T(\tau)y(\tau)d\tau \tag{8.23}$$

where γ is a weight (or penalty) variable and y is your output (which is equal to your virtual constraint vector).

So what I am doing with this new cost function is trying to tell the optimization problem that I also care about error. I want the virtual constraints vector to be as small as possible. In other words, I want θ_3 and θ_2 to be as close to θ_3^d and θ_2^d respectively as possible. The weight variable γ emphasizes how badly I want my virtual constraints vector to be small. A γ of, say, 1 implies that I care about minimizing the virtual constraints as much as I care about minimizing the torque input u. A γ of, say, 20 means that I care about minimizing the virtual constraints a lot more than minimizing the torque inputs – 20 times more, in fact!

8.5 Updating the Matlab Code

Once again, I am going to have to recommend that you save a copy of your current Matlab files and work from a new folder just so you can go back and compare your updated optimization code to your older optimization code.

We will be modifying the following files:

- `SymbolicModelForThreeLinkRobot.m` – Symbolically derive θ_2^d and θ_3^d.
- `ThreeLinkWalker_sim.m` – Update `conditions0` and `conditions` to reflect the updated optimization parameters. Update the upper and lower bounds of optimization problem.
- `costFunction.m` – Update how the `conditions` input variable is split into `x0` and `a`. Update cost function to new cost function defined at the end of previous section.
- `myconstraints.m` – Update how the `conditions` input variable is split into `x0` and `a`.

8.5.1 `SymbolicModelForThreeLinkRobot.m`

We will be adding just one line of code to this file. After your definition of `h` (the line written as `h = h0 - hd;`), insert the following line of code:

```
hd_dot = simplify(jacobian(hd,q)*dq); % dth3desired ; dth2desired
```

As the comment suggests, this line of code defines $\dot{\theta}_2^d$ and $\dot{\theta}_3^d$ symbolically (note that it is indeed `dth3desired` on top, not `dth2desired` because of how we ordered our virtual constraint vector `h`). You will need this when go to update your `x0` vector definition so that you do not have to compute these derivatives by hand.

8.5.2 `ThreeLinkWalker_sim.m`

Since we updated our parameters vector `conditions`, we will also need to update our initial conditions to these parameters `conditions0`. Specifically, get rid of your definition of `x00` and replace it with

```
dth10 = 1.5;
```

and modify your definition of `conditions0` to be

```
conditions0 = [dth10; a0];
```

Next, you will need to modify your upper and lower bounds for the optimization problem to reflect the fact that you now have 9 rather than 14 parameters to optimize.

```
ub = [4;   20*[pi; pi; pi; pi; pi; pi; pi; pi]];
lb = [-4;   20*[-pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi]];
```

Finally, where you split the `conditions` vector to `x0` and `a` – after you call `fmincon` and before you call `ThreeLinkWalker_ODE45.m` – you will need to update this. Specifically, the first component of `conditions` is $\dot{\theta}_1(t_o)$ – which you will need to define the other variables in `x0` – and the last eight components of `conditions` are your cubic coefficients.

Use `SymbolicModelForThreeLinkRobot.m` to help you define what $\dot{\theta}_2$ (`x0(5)`) and $\dot{\theta}_3$ (`x0(6)`) should be – use that new line of code that we just added.

```
% Calculate optimize trajectory
a = conditions(2:end);
a10 = a(1);
a11 = a(2);
a12 = a(3);
a13 = a(4);
a20 = a(5);
a21 = a(6);
a22 = a(7);
a23 = a(8);

x0 = zeros(6,1);
x0(1) = -th1desired;
x0(2) = -x0(1);
x0(3) = a(1)+a(2)*x0(1)+a(3)*x0(1)^2+a(4)*x0(1)^3;
x0(4) = conditions(1);
x0(5) = x0(4)*(5*a23*x0(1)^4 + 4*a22*x0(1)^3 - 3*a23*x0(1)^2*th1desired^2 + ...
    3*a21*x0(1)^2 - 2*a22*x0(1)*th1desired^2 + 2*a20*x0(1) - ...
    a21*th1desired^2^2 - 1);
x0(6) = x0(4)*(3*a13*x0(1)^2 + 2*a12*x0(1) + a11);
```

8.5.3 `costFunction.m`

The changes to this file is pretty minimal. First, update how you split the `conditions` input variable into `x0` and `a` using the exact same definitions that you used in `ThreeLinkWalker_sim.m`.

```
% Calculate optimize trajectory
a = conditions(2:end);
a10 = a(1);
a11 = a(2);
a12 = a(3);
a13 = a(4);
a20 = a(5);
a21 = a(6);
a22 = a(7);
a23 = a(8);

x0 = zeros(6,1);
x0(1) = -th1desired;
x0(2) = -x0(1);
x0(3) = a(1)+a(2)*x0(1)+a(3)*x0(1)^2+a(4)*x0(1)^3;
x0(4) = conditions(1);
x0(5) = x0(4)*(5*a23*x0(1)^4 + 4*a22*x0(1)^3 - 3*a23*x0(1)^2*th1desired^2 + ...
    3*a21*x0(1)^2 - 2*a22*x0(1)*th1desired^2 + 2*a20*x0(1) - ...
    a21*th1desired^2^2 - 1);
x0(6) = x0(4)*(3*a13*x0(1)^2 + 2*a12*x0(1) + a11);
```

Finally, update your cost function to include the modified version that we established at the end of the previous subsection.

```
gamma = 1;
y = hout;
cost_temp = [];
for k=1:length(uout)
    cost_temp(k) = uout(k)'*uout(k)*dt(k) + gamma*(y(k)'*y(k)*dt(k));
end

cost = sum(cost_temp);
```

8.5.4 `myconstraints.m`

In this file, update how you split `conditions` into `x0` and `a` using the exact same code that I showed for both the updated `costFunction.m` and updated `ThreeLinkWalker_sim.m` files.

8.5.5 You Did It!

There you have it! You have once again updated your Matlab code. Go back to `ThreeLinkWalker_sim.m` and hit "Run." Do you notice your code running faster?

An interesting exercise that you can try is setting both your K_p and K_d gains to 0 and seeing how your biped reacts. If the biped walks, then you know that it is just using feed forward control law alone, showing that our periodicity constraint is quite robust.

8.6 Adding more Constraints to Optimization Problem: Speed and Friction

We are slowly improving our optimization problem. For this next iteration, I would like to introduce a few more constraints.

Up until this point, we have been letting the biped move at whatever speed it wanted to satisfy the problem constraints. Now, we are going to set that speed.

Recall method 2 of computing the ground reaction forces where we used the center of mass. In that process, we defined the velocity at the center of mass of the biped (Eq. 7.16). We can control the velocity of the biped by imposing a nonlinear equality constraint on the x-component of this variable. For example, if we wanted the biped to move a 1 m/s, the corresponding nonlinear equality constraint `ceq` would be

$$\texttt{ceq} = \texttt{vcm(1) - 1} \tag{8.24}$$

where `vcm(1)` is the x-component of the velocity vector at the center of mass. Something to note is that the x and y components of velocity at the center of mass of the biped is not constant throughout a step, as can be seen in Figure 7. The x-component of velocity is greatest at the beginning and end of the step and smaller in the middle of the step. This is because at the middle of a step, the biped is essentially moving more in the y-direction as it pivots on the stance foot.

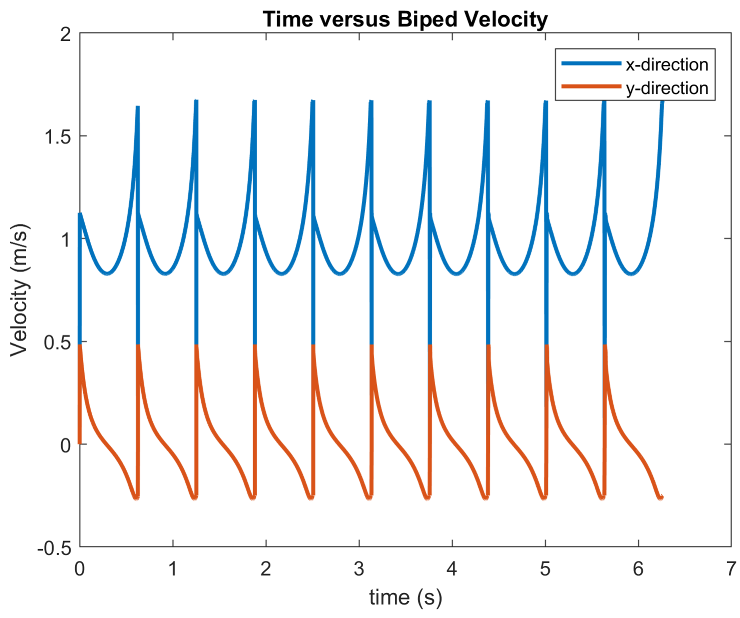

Figure 7: Time versus Velocity at the Center of Mass

Therefore, when setting up the velocity constraint, you must choose how to best represent the x-component of the velocity at the center of mass. You may choose to use the x-component velocity in the middle of the step, at the ends of the step, or you may choose to take the average x-component velocity across the entire step to represent the velocity of the biped. In my Matlab implementation (which I will show in the next subsection), I choose to use the average velocity across the entire step.

Something that you may notice as you implement this velocity constraint is that your torso link may hang straight down. This could be due to `fmincon` choosing that the torso link in this down position best minimizes the optimization problem. To prevent this from happening, you will need to impose nonlinear inequality constraints that forces the torso link angle to remain in a specified range. For example, if you wanted to keep the torso link angle between $-\frac{\pi}{6}$ and $\frac{\pi}{6}$, those constraints would look like this:

$$\texttt{cin(1)} = max(\theta_3) - \frac{\pi}{6} \tag{8.25}$$

$$\texttt{cin(2)} = -min(\theta_3) - \frac{\pi}{6} \tag{8.26}$$

Another constraint to play around with is the friction cone. What if you wanted to simulate your biped walking on various surface roughness (eg, dirt versus ice)? What if you wanted to control how easy it was for your stance foot to lift off of the ground? To impose these constraints, you will put nonlinear inequality constraints on the ground reaction forces. These constraints will look like this:

$$min(F_y) \geq 0.4 \cdot M_{total} \cdot g \tag{8.27}$$

$$max\left(\frac{|F_x|}{F_y}\right) \leq 0.6 \tag{8.28}$$

where F_y is the ground reaction forces in the y-direction, F_x is the ground reaction forces in the x-direction, M_{total} is the total mass of the biped, and g is gravity.

Eq. 8.27 controls the vertical friction component. Adjusting the decimal in this equation (right now, I have it set to 0.4) will determine how easily you want your stance foot off of the ground during a step. Too high of a centripetal force (setting the y-direction friction component too low) and you will find that your stance foot will lift off the ground during a step, meaning that you will transition to "running."

Eq. 8.28 controls the horizontal friction component. A decimal value of 0.6 is about the friction of dirt while a decimal of 0.1 would be about the friction of ice.

Something to note is that Matlab requires these inequality constraints to be in the form of `cin` ≤ 0. Thus, you must rearrange Eq. 8.27 and Eq. 8.28 to be in the form

$$\texttt{cin(3)} = 0.4 \cdot M_{total} \cdot g - min(F_y) \leq 0 \tag{8.29}$$

$$\texttt{cin(4)} = max\left(\frac{|F_x|}{F_y}\right) - 0.6 \leq 0 \tag{8.30}$$

8.7 Updating the Matlab Code

Now that we have identified a few more nonlinear constraints that we want, let's implement them in Matlab! We will be making modifications to the following Matlab files:

- `ThreeLinkWalker_dynamics.m` – Make `vcm` an output.
- `ThreeLinkWalker_ODE45.m` – Reconstruct `vcmout` from `ThreeLinkWalker_dynamics.m` and make it an output argument.
- `myconstraints.m` – Add new constraints. Update call to `ThreeLinkWalker_ODE45.m` to get `vcmout` vector.
- `ThreeLinkWalker_sim.m` – Make a plot of center of mass velocity over time.

8.7.1 `ThreeLinkWalker_dynamics.m`

Before we can update our constraints, we need to make sure that we have all the required variables that we need to impose the constraints. The `myconstraints.m` file will need `GRF` (for the friction constraints) and `vcm` (for the velocity constraint). We are already outputting `GRF`, so all we need to do is add an output argument for `vcm`.

```
function [dx,u,h,GRF,vcm] = ...
    ThreeLinkWalker_dynamics(t,x,th1desired,Kp,Kd,a)
```

8.7.2 `ThreeLinkWalker_ODE45.m`

While our `ThreeLinkWalker_dynamics.m` is now outputting `vcm`, we know that Matlab's `ode45` function does not store the `vcm` values as it solves the ODE. Therefore, we must reconstruct the `vcm` vector (as we have done for `uout`, `hout`, and `GRFout`). Outside the reconstruction `for loop`, add an empty vector definition for `vcmout`.

```
vcmout = [];
```

Inside the `for loop`, update your call to `ThreeLinkWalker_dynamics.m` to include `vcmout`.

```
[~,uout(:,i),hout(:,i),GRFout(:,i),vcmout(:,i)] = ...
    ThreeLinkWalker_dynamics(t,x,th1desired,Kp,Kd,a);
```

Finally, update your output arguments for `ThreeLinkWalker_ODE45.m` to include `vcmout`.

```
function [tout, xout, uout, hout, GRFout, dt, vcmout] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired)
```

8.7.3 `myconstraints.m`

Now we can start implementing our constraints. First, you will need to update your call to `ThreeLinkWalker_ODE45.m` to include `vcmout` as an output.

```
[tout, xout, uout, hout, GRFout, dt, vcmout] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired);
```

Next, somewhere in the code (for example, right after you do your impact calculations) make a call to `model_params_stiff_legs` to get your parameter values (technically, you just need `Mtotal` and g).

```
%% define parameters
[r,m,M_H,M_T,l,g,p_st_foot,Mtotal] = model_params_stiff_legs();
```

Then you can update your constraints to include the new constraints. Something to note here is that for the x and y components of the ground reaction forces (`Fx` and `Fy`), I have chosen to ignore the first 10 values in both vectors. This is because at the beginning of each step (starting with step 2), there will be an impact. This impact results in large spikes in the x and y components of the ground reaction forces. We want to ignore these spikes when imposing our constraints because they effectively act as outliers.

```
%% define constraints
Fx = GRFout(1,10:end);
Fy = GRFout(2,10:end);

cin = [];
cin(1) = max(xout(:,3))-pi/6; % keep th3 upright
cin(2) = -min(xout(:,3))-pi/6; % keep th3 upright
cin(3) = 0.4*Mtotal*g - min(Fy); % affects centripetal force
cin(4) = max(abs(Fx)/Fy) - 0.6 % friction constraint


ceq = [];
ceq(1) = norm(x0-x_plus); % periodicty constraint
ceq(2) = mean(vcmout(1,:))-1 % constrain average velocity
```

8.7.4 ThreeLinkWalker_sim.m

An optional thing that you may wish to do is make a plot of your center of mass velocity. To do this, first update your call to ThreeLinkWalker_ODE45.m when you compute the optimize trajectory (after the call to fmincon) for the "First Step."

```
[tout, xout, uout, hout, GRFout, dt, vcmout] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired);
```

Do the same for the "Next Step."

```
[tout_next, xout_next, uout_next, hout_next, ...
    GRFout_next, dt_next, vcmout_next] = ...
    ThreeLinkWalker_ODE45(t_end,x_plus,a,options,th1desired);
```

Next, append the vcmout vector from the "Next Step" to the vcm from the "First Step" (you did something similar for tout, xout, p_foot, uout, hout, and GRFout.

```
vcmout = [vcmout vcmout_next];
```

Finally, in the "Plots" section, add a plot for time versus center of mass velocity.

```
figure
plot(tout(1:end-numSteps-1),vcmout,'LineWidth',2)
title('Time versus Biped Velocity')
xlabel('time (s)')
ylabel('Velocity (m/s)')
legend('x-direction','y-direction')
```

8.7.5 You Did It!

Another iteration done! Go back to your top level code and watch that bad boy run!

8.8 Introducing Bézier Curves to the Virtual Constraints

We started off this optimization section by introducing the cubic function to our virtual constraints. This was a good start, relatively straightforward to conceptualize. But the problem with cubic functions is that they are rather rigid. You cannot modify any aspect of the curve without completely changing the entire cubic function.

What if I told you there was a way to change up, say, the middle part of a cubic curve without affecting its end points? Or what if I could change up a small section of any higher order curve without it affecting the rest of the curve?

Ladies and gentlemen, let me introduce to you **bézier curves**!

A bézier curve is a curve that is defined by two or more "control points." These control points may be located on the curve itself or outside of the curve.

The curve's order is determined by the number of control points that the curve has. Specifically, a bézier curve's order is equal to the number of its control points minus one. For example, a linear curve (ie, a straight line) has two control points; a quadratic curve (ie, a parabola) has three control points; a cubic curve has four control points; and so on.

Regardless of how many control points a bézier curve has, the curve is always located inside the "convex hull" of its control points. Convex hull here refers to the minimum enclosed space if you were to draw an encompassing region through all of the control points.

For a (better written, plus with pictures!) general introduction to bézier curves, see [10]. For a more in depth discussion of literally everything that you could possibly want to know about bézier curves (and then some!), see [11].

Chances are that you have already encountered bézier curves through graphics programs such as Photoshop.

But enough with the overview. Let's get to the math.

A one-dimensional bézier polynomial of degree M is defined as follows:

$$b_i(s) := \sum_{k=0}^{M} \alpha_k^i \frac{M!}{k!(M-k)!} s^k (1-s)^{M-k} \tag{8.31}$$

where $s \in [0,1]$, i is the i^{th} bézier curve (for example, if you are writing two bézier curves, you would refer to the first one as $b_1(s)$ and the second one as $b_2(s)$), M is the order of the bézier curve, and α_k^i is the k^{th} coefficient for the i^{th} bézier curve (there are $M+1$ coefficients for each bézier curve).

When you want to use a bézier curve, a given function $\theta(q)$ of generalized coordinates will not, in general, take values that span only across the unit interval. Since the function that defines a bézier curve, $b(s)$, requires that its independent variable s span a unit interval $[0,1]$, we often have to normalize $\theta(q)$ using the following formula

$$s(q) := \frac{\theta(q) - \theta^+}{\theta^- - \theta^+} \tag{8.32}$$

where θ^- is the smallest allowable value for θ and θ^+ is the largest allowable value for θ.

Something interesting to note is that the partial derivative of the bézier curve with respect to s is

$$\frac{\partial b_i(s)}{\partial s} = \sum_{k=0}^{M-1} \left(\alpha_{k+1}^i - \alpha_k^i\right) \frac{M!}{k!(M-k-1)!} s^k (1-s)^{M-k-1} \tag{8.33}$$

Figure 8 depicts a bézier curve of order 5. As expected, it has six control points (the number of control points that a bézier curve has is its order plus one) and the curve itself fits within the convex hull of the six control points. The control points are exactly the bézier curve coefficients α_k^i from Eq. 8.31 and thus these two terms (bézier curve coefficients and bézier curve control points) can be used interchangeably. The beginning of the curve starts at $b(0) = \alpha_0$, and the end of the curve is at $b(1) = \alpha_5$. This is no accident. For all bézier curves, $b_i(0) = \alpha_0^i$ and $b_i(1) = \alpha_M^i$; that is, the i^{th} bézier curve starts at the first coefficient α_0^i and ends at the last coefficient α_M^i.

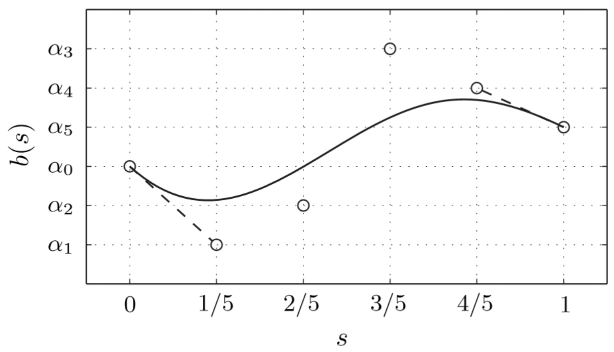

Figure 8: A Bézier Curve of Order 5 [4]

In our application for the biped, we will also be using a bézier curve of order 5 to model the desired trajectory for both θ_3^d and θ_2^d. Thus, our updated virtual constraints will be of the form

$$h(x) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \theta_3 - \theta_3^d(\theta_1, \alpha) \\ \theta_2 - \theta_2^d(\theta_1, \alpha) \end{bmatrix} = \begin{bmatrix} \theta_3 - b_1(s, \alpha) \\ \theta_2 - b_2(s, \alpha) \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{8.34}$$

Using Eq. 8.31, we derive θ_3^d to be

$$\theta_3^d = b_1(s) = \alpha_0^1 \frac{5!}{0!(5-0)!} s^0 (1-s)^{5-0} + \alpha_1^1 \frac{5!}{1!(5-1)!} s^1 (1-s)^{5-1} + \alpha_2^1 \frac{5!}{2!(5-2)!} s^2 (1-s)^{5-2} + \alpha_3^1 \frac{5!}{3!(5-3)!} s^3 (1-s)^{5-3} + \alpha_4^1 \frac{5!}{4!(5-4)!} s^4 (1-s)^{5-4} + \alpha_5^1 \frac{5!}{5!(5-5)!} s^5 (1-s)^{5-5} \tag{8.35}$$

which reduces to

$$b_1(s) = \alpha_0^1 (120)(1-s)^5 + \alpha_1^1 (5) s (1-s)^4 + \alpha_2^1 (10) s^2 (1-s)^3 + \alpha_3^1 (10) s^3 (1-s)^2 + \alpha_4^1 (5) s^4 (1-s) + \alpha_5^1 s^5 \tag{8.36}$$

Similarly, θ_2^d would be

$$\theta_2^d = b_2(s) = \alpha_0^2 (120)(1-s)^5 + \alpha_1^2 (5) s (1-s)^4 + \alpha_2^2 (10) s^2 (1-s)^3 + \alpha_3^2 (10) s^3 (1-s)^2 + \alpha_4^2 (5) s^4 (1-s) + \alpha_5^2 s^5 \tag{8.37}$$

We normalize θ_1 (which has a maximum value of θ_1^d and a minimum value of $-\theta_1^d$) for both of these curves to fit the unit interval of s. Thus,

$$s = \frac{\theta_1 - \theta_1^d}{-\theta_1^d - \theta_1^d} \tag{8.38}$$

8.9 Updating the Matlab Code

There are a few new things that we want to add to the Matlab code. 1) We want to update our virtual constraints to include the bézier curves. 2) We want to modify our code such that we plot the velocity of the swing foot (for analysis purposes). 3) We want to add a zero dynamic-like hack such that we force our output function y and its derivative $\dot{y}$ to both equal 0.

To make these modifications, we will need to modify the following Matlab files:

- `SymbolicModelForThreeLinkRobot.m` – Update definition of `hd`. Define `s` and `dh`.

- `ControlFunctions.m` – Update definition of `a`, `h`, and the other control functions. Define new vector for `dh`.
- `myconstraints.m` – Update definitions for `a` and `x0`. Add new equality constraints.
- `costFunction.m` – Update definitions for `a` and `x0`.
- `ThreeLinkWalker_sim.m` – Update definitions for `a` and `x0`. Update `a0`, `ub` and `lb`. Add plots for `v_sw_foot`.
- `ThreeLinkWalker_ODE45.m` – Define and add output argument for `dhout`.
- `ThreeLinkWalker_dynamics.m` – Add output argument for `dh`.
- `Points_ThreeLinkWalker.m` – Define and add output argument for `v_sw_foot`. Define `dth1`, `dth2`, and `dth3`.

8.9.1 `SymbolicModelForThreeLinkRobot.m`

The first step is to redefine our virtual constraint vector `h` to incorporate our new θ_3^d and θ_2^d definitions that uses the bézier curve. Recall that we are using a 5th order bézier curve for both values which we defined in Eq. 8.36 and Eq. 8.37. Also note that we will need to define `s` as the normalized θ_1, which we defined in Eq. 8.38. Finally, we will need to make sure to update the variables that we define with `syms`.

```
syms a10 a11 a12 a13 a14 a15 a20 a21 a22 a23 a24 a25 th1desired

s = (th1 - th1desired)/(-th1desired - th1desired);

h0 = [th3; th2];
hd = [120*a10*(1-s)^5+5*a11*s*(1-s)^4+10*a12*s^2*(1-s)^3+...
    10*a13*s^3*(1-s)^2+5*a14*s^4*(1-s)+a15*s^5; ...
    120*a20*(1-s)^5+5*a21*s*(1-s)^4+10*a22*s^2*(1-s)^3+...
    10*a23*s^3*(1-s)^2+5*a24*s^4*(1-s)+a25*s^5];
h = h0 - hd;
```

One last thing to add to this code is taking the derivative of `h` using the jacobian. We will need this for when we impose the nonlinear equality constraint that sets the derivative of the output function $\dot{y} = 0$ (recall that $y = h$ and thus $\dot{y} = \dot{h}$).

```
dh = simplify(jacobian(h,q)*dq);
```

Just to clarify, we have a variable hd which we use to define the desired values of θ_3 and θ_2 where

$$\text{hd} = \begin{bmatrix} \theta_3^d \\ \theta_2^d \end{bmatrix} \tag{8.39}$$

And we now have the variable dh which is used to define the derivative of h where

$$\text{dh} = \frac{\partial h}{\partial q}\dot{q} \tag{8.40}$$

8.9.2 `ControlFunctions.m`

The first adjustment to this code is to update your definitions for the extraction for the values in the input vector a.

```
a10 = a(1);
a11 = a(2);
a12 = a(3);
a13 = a(4);
a14 = a(5);
a15 = a(6);
a20 = a(7);
a21 = a(8);
a22 = a(9);
a23 = a(10);
a24 = a(11);
a25 = a(12);
```

Next, update your definition for h by using the new symbolic definition for h that you can generate by running the updated `SymbolicModelForThreeLinkRobot.m` file.

```
h(1)= ...
h(2)= ...
```

Next, define a new vector for dh and set its values equal to the symbolic definition that you generated in `SymbolicModelForThreeLinkRobot.m`.

```
dh = zeros(2,1);
dh(1)= ...
dh(2)= ...
```

Update the definitions for the remaining control functions `Lfh`, `L2fh`, and `LgLfh` with the new symbolic definitions that you generated in the updated `SymbolicModelForThreeLinkRobot.m` file.

```
Lfh(1)= ...
Lfh(2)= ...

L2fh(1)= ...
L2fh(2)= ...

LgLfh(1,1)= ...
LgLfh(1,2)= ...
LgLfh(2,1)= ...
LgLfh(2,2)= ...
```

Finally, update your output arguments for this function (`ControlFunctions.m`) to include `dh`.

```
function [h,Lfh,L2fh,LgLfh,dh] = ...
    ControlFunctions(q,dq,parameters,th1desired,a)
```

8.9.3 `myconstraints.m`

First, update your definition for the a vector. Something to note is that we want the first and last bézier curve coefficients for the bézier curve that defines θ_3^d to equal to each other. This is to ensure that we meet periodicity. Recall that $b_i(0) = \alpha_0$ and $b_i(1) = \alpha_M$, that is, that every bézier curve starts at the value of its first coefficient α_0 and ends at the value of its last coefficient α_M. Thus to impose our periodicity constraint, we are setting the first and last coefficients equal to each other. Similarly, we will be imposing the mirror law for the bézier curve that defines θ_2^d be setting its first and last coefficients to equal to each other.

```
a = conditions(2:end);
a10 = a(1);
a11 = a(2);
a12 = a(3);
a13 = a(4);
a14 = a(5);
a15 = a10; % constraint torso link to start and end at same angle
a20 = a(7);
a21 = a(8);
```

```
a22 = a(9);
a23 = a(10);
a24 = a(11);
a25 = a20; % impose mirror law
```

Next, update your definition for the x0 vector. We still want the initial values for θ_1 (x0(1)) and θ_2 (x0(2)) to equal to each other because of mirror law. We want the initial value of θ_3 (x0(3)) to equal to the first bézier curve coefficient (or the last coefficient since they are the same) for θ_3^d since this coefficient represents where θ_3 will start. The initial value of $\dot{\theta}_1$ (x0(4)) is given from the input arguments. You can generate the symbolic definitions for the initial values of $\dot{\theta}_2$ (x0(5)) and $\dot{\theta}_3$ (x0(6)) by running your SymbolicModelForThreeLinkRobot.m code and looking at the values of hd_dot (where hd_dot(1)=x0(6) and hd_dot(2)=x0(5)).

```
x0 = zeros(6,1);
x0(1) = -th1desired;
th1 = x0(1);
x0(2) = -x0(1);
x0(3) = a10;
x0(4) = conditions(1);
dth1 = x0(4);
x0(5) = ...
x0(6) = ...
```

Next, update your call to ThreeLinkWalker_ODE45.m to include the new dhout output variable (which we will add in a subsequent subsection).

```
[tout, xout, uout, hout, GRFout, dt, vcmout, dhout] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired);
```

Next, where you define your equality constraints, first define your output function y and its derivative $\dot{y}$ to equal to the virtual constraint matrix h and its derivative $\dot{h}$.

```
y = hout;
dy = dhout;
```

Using these definitions, you can add the new following constraints

```
ceq(3) = max(norm(y)^2);
ceq(4) = max(norm(dy)^2);
```

...except I have not actually gotten the code to quite work with these new constraints yet, so maybe just comment them out when you run the code.

8.9.4 `costFunction.m`

The changes to this code is simple. Simply copy and paste your new definitions for `a` and `x0` from `myconstraints.m` into this file, replacing the old definitions.

8.9.5 `ThreeLinkWalker_sim.m`

Update your vector for `a0` to account for the additional number of coefficients that we are using with this bézier curve application for the virtual constraints.

```
a0 = [pi/6 1 1 1 1 1 1 1 1 1 1 1]';
```

Similarly, update your upper and lower bounds (`ub` and `lb`) definitions for your call to `fmincon` to account for the additional variables.

```
ub = [4;   10*[pi; pi; pi; pi; pi; pi; pi; pi; pi; pi; pi; pi]];
lb = [-4;   10*[-pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi; -pi]];
```

Update your definitions for the vectors `a` and `x0` after you perform `fmincon` to be the same new definitions that we defined in `myconstraints.m` and `costFunction.m`.

At the end of the "First Step" section, add a vector that defines the velocity of the swing foot at every data point using a call to `Points_ThreeLinkWalker.m` (which we will be updating the file for in a later subsection).

```
v_sw_foot = [];
for i=2:length(q)-1
    [~,~,~,~,v_sw_foot(:,i)] = Points_ThreeLinkWalker(q,dq);
end
```

Do the same in the "Next Steps" section

```
v_sw_foot_next = [];
for i=2:length(q)-1
    [~,~,~,~,v_sw_foot_next(:,i)] = Points_ThreeLinkWalker(q,dq);
end
```

Append the velocity values from the next step to the previous step (similar to what you did for `tout`, `xout`, `p_foot`, `uout`, `hout`, `GRFout`, and `vcmout`).

```
v_sw_foot = [v_sw_foot v_sw_foot_next];
```

Now you can plot the velocity of the swing foot with respect to time.

```
figure
plot(tout(1:end-numSteps-1), v_sw_foot,'LineWidth',2)
title('Time versus swing foot velocity')
xlabel('time (s)')
ylabel('Velocity (m/s)')
legend('x-direction','y-direction')
```

8.9.6 `ThreeLinkWalker_dynamics.m`

There is only two minor changes that needs to be made to this file. First, add `dh` as an output argument for the function.

```
function [dx,u,h,GRF,vcm,dh] = ThreeLinkWalker_dynamics(t,x,th1desired,Kp,Kd,a)
```

Next, add `dh` as an output to your call to `ControlFunctions.m`.

```
[h,Lfh,L2fh,LgLfh,dh] = ControlFunctions(q,dq,parameters,th1desired,a);
```

8.9.7 `ThreeLinkWalker_ODE45.m`

In this Matlab file, we are going to first add an output argument for `dhout`.

```
function [tout, xout, uout, hout, GRFout, dt, vcmout, dhout] = ...
    ThreeLinkWalker_ODE45(t_end,x0,a,options,th1desired)
```

Next, we need to define `dhout` by reconstructing it from `ThreeLinkWalker_dynamics.m`. So first, outside of the reconstruction `for loop`, define an empty vector for `dhout`.

```
dhout = [];
```

Then, inside the reconstruction `for loop`, populate the `dhout` variable by including it in the output from your call to `ThreeLinkWalker_dynamics.m`.

```
    [~,uout(:,i),hout(:,i),GRFout(:,i),vcmout(:,i),dhout(:,i)] = ...
        ThreeLinkWalker_dynamics(t,x,th1desired,Kp,Kd,a);
```

8.9.8 `Points_ThreeLinkWalker.m`

We use this file to generate the velocity of the swing foot at a given point. So first add `v_sw_foot` as an output argument for the function.

```
function [p_st_foot,p_hip,p_torso,p_sw_foot,v_sw_foot] = ...
    Points_ThreeLinkWalker(q,dq,p_foot)
```

Since the velocity will include $\dot{q}$ terms, we need to define `dth1`, `dth2`, and `dth3` using the input arguments (since this was not done before).

```
dth1 = dq(1);
dth2 = dq(2);
dth3 = dq(3);
```

Finally, you may define the velocity of the swing foot by using the symbolic notation that you can generate from the `SymbolicModelForThreeLinkRobot.m` file.

```
v_sw_foot = [dth1*r*cos(th1) - dth2*r*cos(th2);...
    dth2*r*sin(th2) - dth1*r*sin(th1)];
```

8.9.9 You Did it!

Congratulations! You completed another update of your code. Now go see your hard work pay off.

9 Thus Concludes Biped Bootcamp

Well done pupil. You made it through Biped Bootcamp!

Of course, there is still much, much more to learn about all things bipeds. In fact, this does not even cover all the material that I learned through my own "bootcamp." I did not get to show you how to use FROST or how to transition to a 5-link biped model. But this guide does cover everything that I learned in the summer before my first year as a Robotics PhD student. It is an accurate depiction of my thought process as I was learning all of this material for the first time.

Looking back at my code and thought processes as a now rising second year Robotics PhD student, I see many things that I would like to have changed for the better. However, I made the decision to keep this document largely the same as when I first wrote it. I did this because I see value in a document written by a novice for fellow novices in the topic. There are places where I went into lengthy detail because I wanted to address topics that I struggled with at the time. If I re-wrote the document today, I would not likely go into the same level of detail because these concepts are now somewhat second nature to me, and I would not know how to explain them in a manner that would make sense to someone hearing it for the first time. As a novice understanding a particular concept for the first time, I knew exactly how to explain it in a way that I wish that I had heard it for faster understanding. Still, there were many topics that I wish that I wrote about in this document that I did not fully understand well enough to talk about. Perhaps there might be a second edition of this document that I can write sometime in the future to address the shortcomings in this first edition...

In any case, I hope to write a similar document that goes through how to model and control a 5-link biped walker and that explains how to use FROST to generate the equations of motion and kinematics of any biped. I also wish to turn this first edition Biped Bootcamp document into a video series and/or a series of lesson plans that goes over the same material for more accessible learning. So be on the look out for those future projects!

But with that, that is all that I have for you for now. Thank you for tuning in. I will see you in the next one!

About the Author

Oluwami Dosunmu-Ogunbi, also known as Wami Ogunbi to her peers, is a first year Robotics PhD student at the University of Michigan. She is advised by Professor Jessy Grizzle in the Biped Robotics Lab. Her current research focus is in controls with applications in bipedal locomotion.

She has an interest in effectively disseminating complex engineering and robotics concepts to a wide audience, and so one of her long-term goals is to become a professor.

Learn more about Wami by visiting her website at wamiogunbi.com.

Figure 9: Wami standing next to Agility Robotics' Digit

References

[1] https://en.wikipedia.org/wiki/Lagrangian_mechanics

[2] R., Murry, Z., Li, and S. Sastry. A Mathematical Introduction to Robotic Manipulation CRC Press. 1994.

[3] J., Grizzle, G., Abba, F., Plestan. "Asymptotically Stable Walking for Biped Robots: Analysis via Systems with Impulse Effects." *Automatic Control*. 46:1. Jan 2001. p 51-64,

[4] E., Westervelt, J., Grizzle, C., Chevallereau, J., Choi, B., Morris. Feedback Control of Dynamic Bipedal Robot Locomotion. *CRC Press*. 2007.

[5] https://www.mathworks.com/help/matlab/ref/ode45.html

[6] https://www.mathworks.com/help/matlab/math/ode-event-location.html

[7] https://en.wikipedia.org/wiki/Feedback_linearization

[8] https://www.mathworks.com/help/optim/ug/fmincon.html

[9] https://www.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html

[10] https://javascript.info/bezier-curve

[11] https://pomax.github.io/bezierinfo/