Chris Bonney

ESE 447

4/28/20

Part 1: Digital Selector

The goal for this section is to create the digital part of a hybrid controller that will balance the pendulum arm of the robot. The purpose of the digital controller is to select the mode of the analog controller.

There will be three modes: swing up, wait, and balance. Swing up will attempt to get the pendulum arm close to vertical, where the balance controller can kick in; this is important because the balance controller will be based on a linearized model of the system, which will only be accurate close to the vertical fixed point. The balance mode will obviously attempt to balance the arm. And finally, the wait mode will feed a 0 input through to the motor, allowing the arm to settle back again. The controller can be made in Simulink using a multiport switch, as shown in Figure 1.

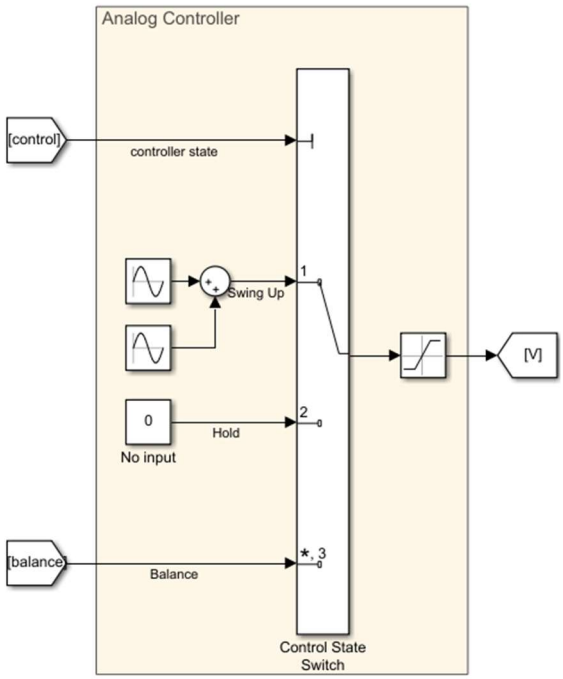

Figure 1: Multiport Switch for Analog Controller

Now, all we have to do is design logic for the controller. For this we will use a Matlab Function block. The block will take in the previous time step's control state, as well as a normalized version of the system's states. This is shown in Figure 2. The logic is as follows: the controller can only go between states in order of

$$\text{Balance } (3) \rightarrow \text{Hold } (2) \rightarrow \text{Swing up } (1) \rightarrow \text{Balance } (3)$$

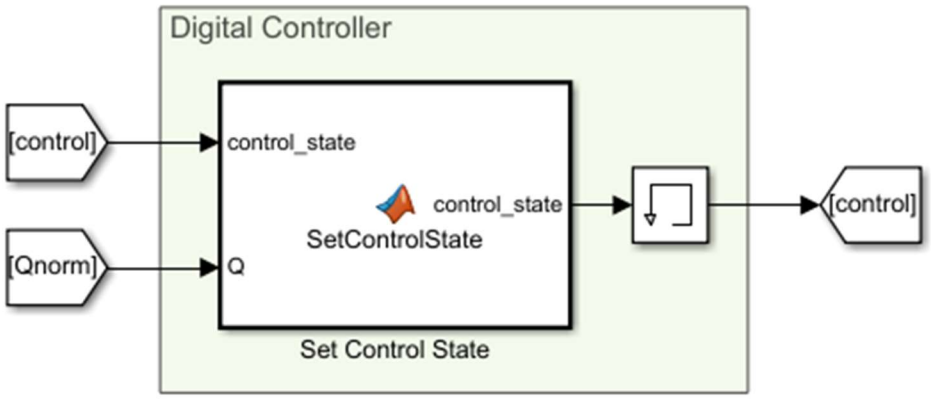

Figure 2: Setting the Control State

and depends on the second joint's position and velocity:

- When in balance mode (3), switch to wait (2) if the second joint is more than 25° away from vertical
- When in wait (2), switch to swing up (1) if the second joint is within 10° of straight down, and its angular velocity is less than 0.01 rad/sec
- When in swing up (1), switch to balance (3) if the second joint is within 20° of vertical.

The code for this function is shown in Figure 3.

```
function control_state = SetControlState(control_state, Q)

q2 = Q(2);
q2dot = Q(4);

q2 = 2*atan2(tan(q2/2),1);

switch control_state

    case 1
        if (abs(q2) < 20*pi/180)
            control_state = 3;
        end
    case 2
        if (abs(q2) > pi-5*pi/180 && abs(q2dot) < 0.01)
            control_state = 1;
        end
    case 3
        if abs(q2) > 25*pi/180
            control_state = 2;
        end
end
```

Figure 3: Setting the Control State

Lastly, we have to normalize the q_2. The goal is to limit the joint angle to between $[-\pi, \pi]$, with switching between the two endpoints. This is important because the controller uses the angle of the joints relative to 0 for its calculations. To do this, I used the following equation:

$$q_{norm} = 2\arctan\left(\tan\left(\frac{q}{2}\right)\right)$$

I created a graph in Desmos to plot this in a Desmos graph in Figure 4, and confirm that it matches the behavior desired. This normalization also works for normalizing q_1, which will be important for the analog controller, even though it is not needed for the digital selector.

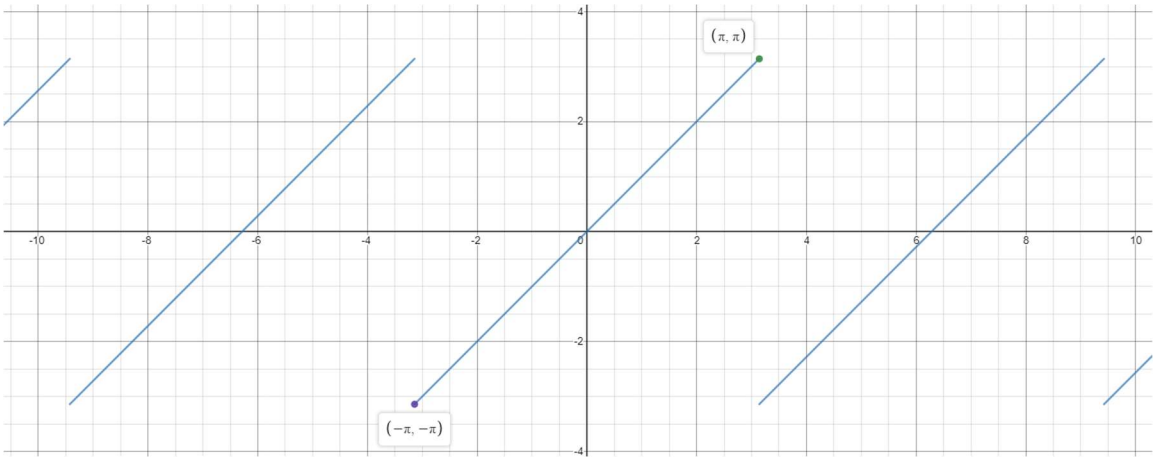

Figure 4: Normalized Joint Angles

I created a subsystem in Simulink to do this calculation, shown in Figure 5.

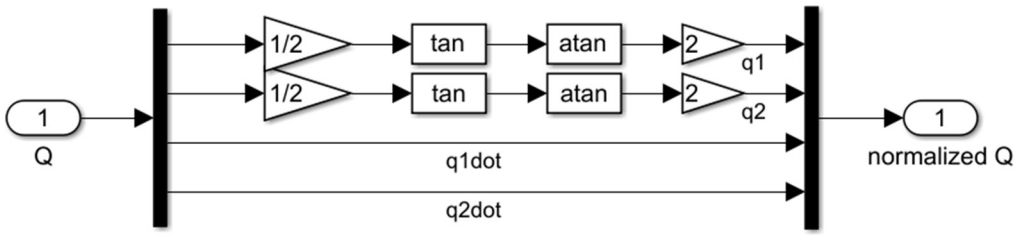

Figure 5: Trig Normalizer for Joint Angles

Using an inefficient swing up controller of summed sine functions, I can show that the behavior of the digital selector is as desired. I ran the full model, plotting the state q_2, the control state, and the input voltage. This is shown in Figure 6 in both a graph and video animation.

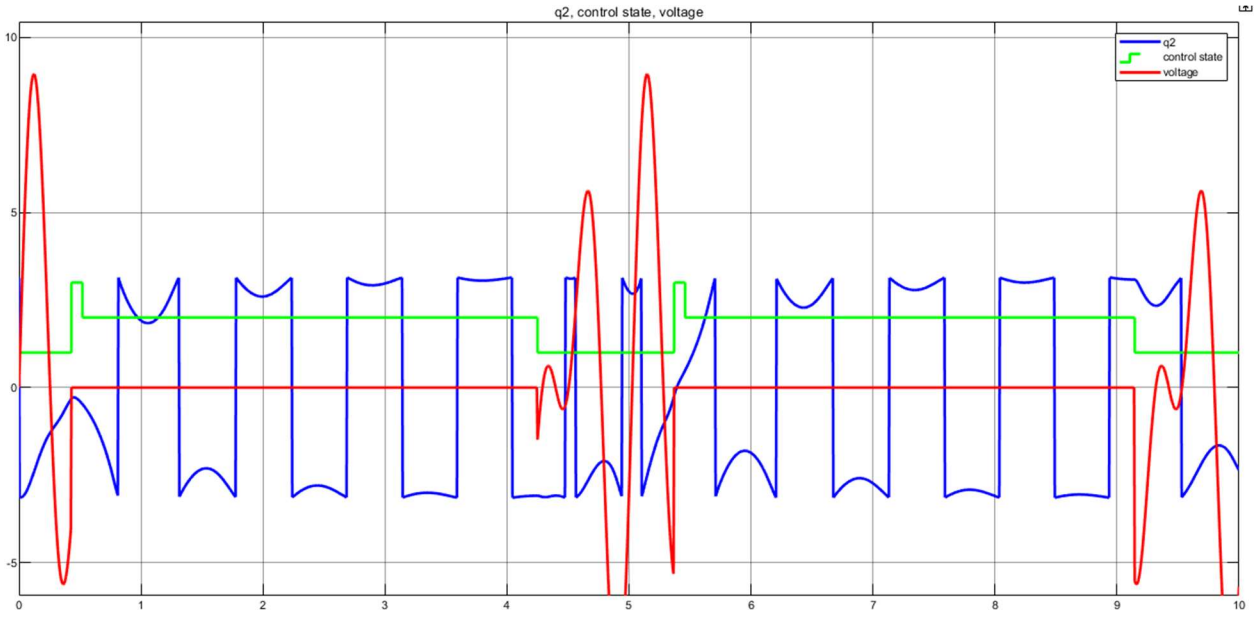

Figure 5: Simulation of Digital Selector with 0 Balance Input

We can see that at time 0.5, the state q_2 gets within 20° the control state switches to balance (3), until it falls outside that range, whereupon it goes into wait (2), until the arm has come close to being at rest, after this the process restarts. We can also observe the control input shutting off for states 2 and 3, because for this simulation I set the balance input to be 0, and the wait input should be 0 to allow the arm to swing to rest.

The full Simulink model is shown below in Figure 7.

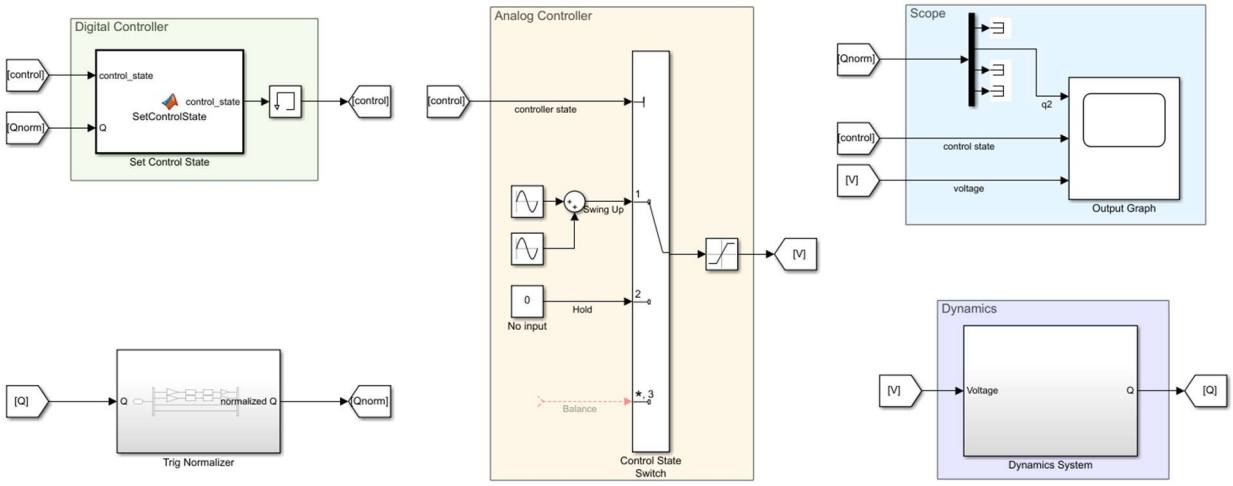

Figure 7: Full Simulink Model for Simulating the Digital Controller

Part 2: Swing Up Controller

The goal for this section is to design a controller for the system when the digital controller is in the first state, the purpose of this controller is to swing the arm up to a vertical position.

To start with, we designed a controller that used q_2 and $\dot{q_2}$ to calculate a desired position of q_1 and used PV or PD gains of the error of q_1 and $\dot{q_1}$ to calculate the input to steer to the desired value of q_1. If designed correctly, this steered the system to the pendulum arm being vertical, at which point we could let the balance controller kick in. The Simulink implementation of this is shown in Figure 1 where *P, V* are the calculated gains, and $q1d = \text{atan2}(q_2 + \pi)$. There were a few difficulties in I had in implementing this design which I will discuss next in the design process, and then some alternatives I designed that have different benefits and downsides.

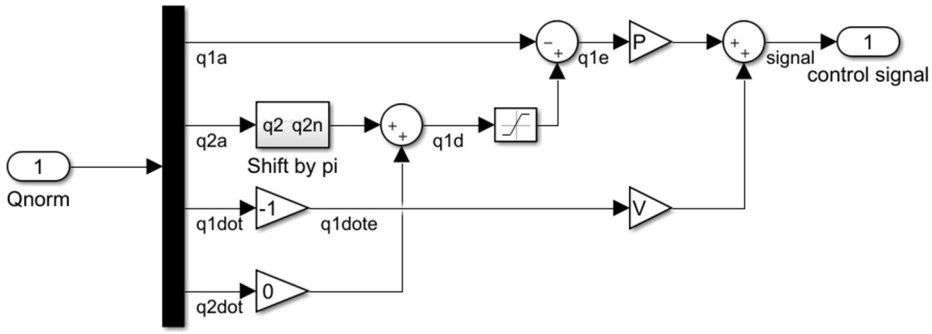

Figure 1: PV Swing Up Controller

The first part to designing this controller is calculating *P, V*. This was done by matching transfer functions. Our goal is to create the second order characteristic transfer function

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

We start by constructing the desired closed loop transfer function of the system given an input *q1d* using a black box transfer function. A model of the transfer function is shown in Figure 2.

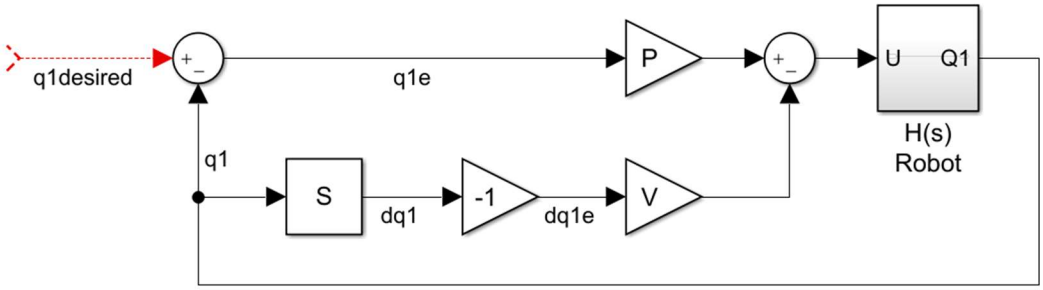

Figure 2: Closed Loop PV Transfer Function

From Figure 2, we can derive the transfer function of $q1a/q1d$, q_1 actual over q_1 desired.

$$\frac{q1a}{q1d} = \frac{PH}{(1+PH)+sHV}$$

We estimate the *H(s)* using a second order transfer function

$$H(s) = \frac{k}{s(\tau_s s + 1)}$$

and plug that into the transfer function in the previous equation and rewriting it in the form of the second order characteristic transfer function

$$\frac{q1a}{q1d} = \frac{PK/\tau_s}{s^2 + (1+VK)/\tau_s + PK/\tau_s}$$

This allows us to solve for the desired values of P and V,

$$P = \omega_n^2 \frac{\tau_s}{K}\ ,\ \ V = (2\zeta\omega_n\tau_s - 1)/K$$

In order to solve these, we need to know values for $\omega_n,\ \zeta,\ K,$ and τ_s. The later two are given as part of the description of the simplified transfer function of the robot *H(s)*, and the first two we can calculate from the desired transfer function *G(s)* by knowing the desired behavior. In this case, we used the peak time, t_p, and peak overshoot, M_p and the equations to get $\omega_n,\ \zeta$ from them,

$$\zeta = \sqrt{\frac{\ln(M_p)^2}{\pi^2 + \ln(M_p)^2}}\ ,\ \ \omega_n = \frac{\pi}{t_p\sqrt{1-\zeta^2}}$$

I made a MATLAB script to calculate these values using given values for $\omega_n,\ \zeta,\ K,$ and τ_s, shown in Figure 3.

```
k = 1.74;
tau = 0.025;

Mp = 0.05;
tp = 0.15;

zeta = sqrt(log(Mp)^2/(pi^2+log(Mp)^2));
omega = pi/(tp*sqrt(1-zeta^2));

P = omega^2*tau/k;
V = (2*zeta*omega*tau-1)/k;
```

Figure 3: Calculating P,V *for Swing Up Controller*

I calculated *P,V* to be

$$P = 12.0332\ ,\ \ V = -0.00082.$$

From this, we simplify our model somewhat. Because the V gain is so small, we can try eliminating the V component of the model since it is small enough to be insignificant, I accidentally had -V in one version of the controller and it still behaved as intended. This simplification is actually very convenient for implementing on the physical robot since we don't have measurements for the derivative of the states and would have to approximate them with a discrete derivative.

Lastly, we just need to set gains for calculating the desired value for q1. It turns out that just taking the value of q_2, using a slightly different normalization to what was shown in Part 1, worked, and the controller is not very sensitive to that parameter, so any gains near 1 also worked. This is again convenient that we don't have to use the derivative state. The simplified controller is shown in Figure 4.

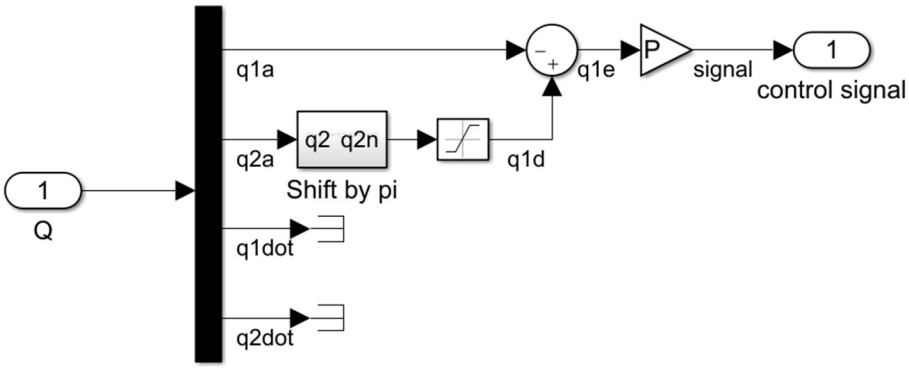

Figure 4: Simplified PV (P) Controller

The results of the full PV controller and the simplified P controller are shown below in graph and video in Figures 5 and 6. We can see that the graphs are nearly identical, so the controller performs equally well with the simplified model. Because they are so similar, I only included a video of the PV controller.

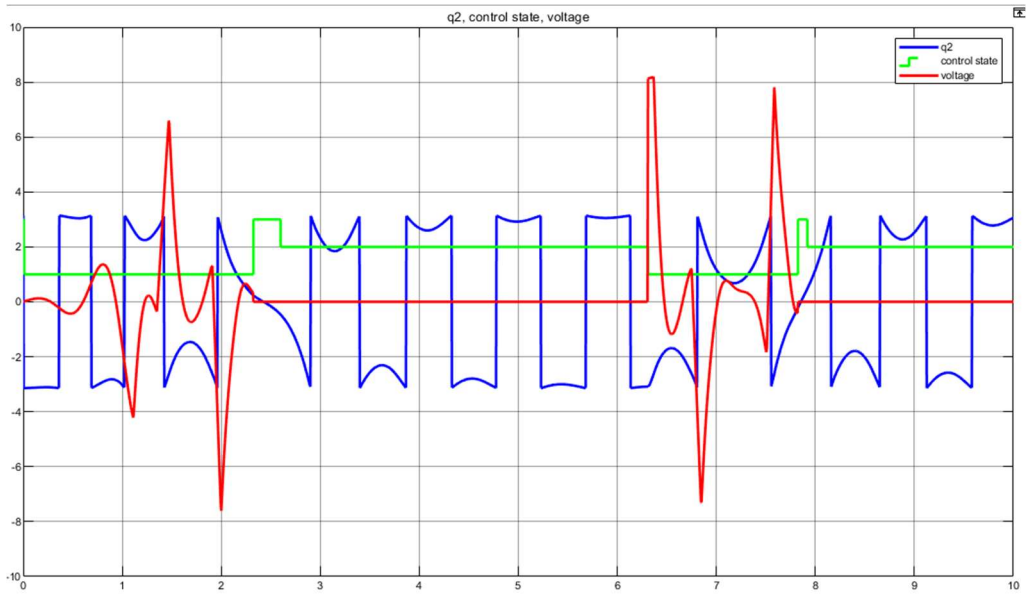

Figure 5: PV Swing Up Controller

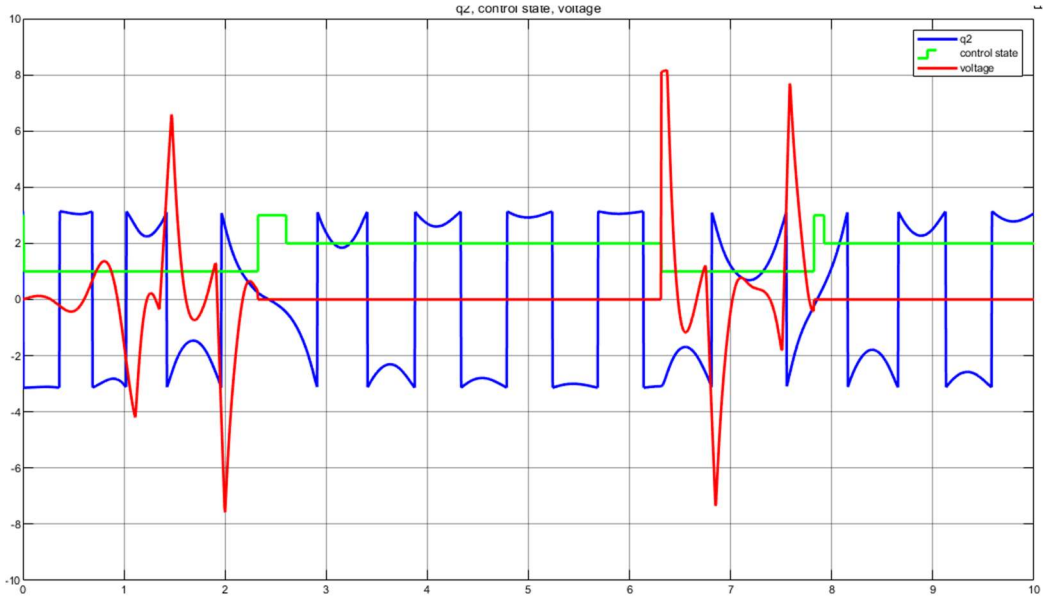

Figure 6: Simplified P Swing Up Controller

There were two main difficulties I had in designing this controller that didn't have to do with calculating the controller gains. Those were renormalizing q_2 and starting the simulation at 0, which are related.

First, based on the normalization scheme defined in in Part 1, when the pendulum arm is at rest on the down position, $q_2 = \pm\pi$, and has a discontinuity. This is bad for two reasons. First, a slight movement can drastically changes the value, and switches the sign, of q_2, and second, we cross this discontinuity often when in the process of swinging the arm up. The solution to this is instead renormalizing q_2 so that it is 0 at the bottom instead of π and goes to $\pm\pi$ at vertical. We

don't care about the discontinuity when it is vertical, because the balance mode of the controller will have taken over by that point. If you were to open a scope to graph the control voltage, it would be switching back and forth from ± 10, the saturation input.

This creates the second problem, however. If we calculate $q1d$ when the arm is straight down, we get 0, and with $q1a$ also 0, there will be no control signal input, and the arm will just sit there. Therefore, we need to give it a "kick" as part of the initial condition. This can come in the form of giving any of the states or state derivatives a non-zero value (or non-pi value in the case of q_2), which allows the controller to kick in.

The last part of this controller is the saturation block on $q1d$, which is optional. The purpose of the saturation block is to make the swings shorter and faster, which will swing q_2 up faster, while preventing $q1$ from straying too far from 0.

The last swing up design I made was a bang-bang controller that just input saturation one way until the arm was slightly above horizontal and then switched directions. The controller is shown below in Figure 7.

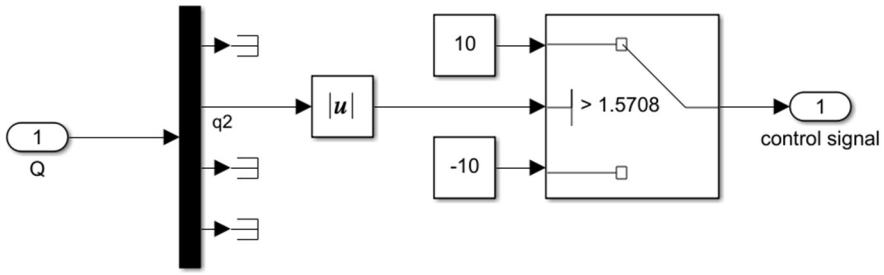

Figure 7: Bang-Bang Swing Up Controller

This controller had the best time-performance, as is expected by the Bang-Bang Theorem. By the same theorem, it requires one control input switch. The bang-bang controller is shown working in graph and video in Figure 8.

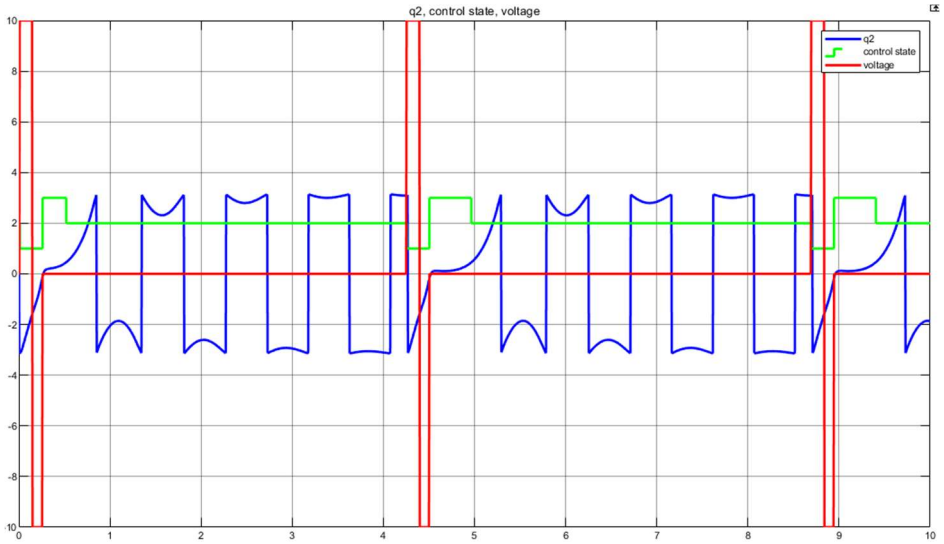

Figure 8: Bang-Bang Swing Up Controller

The downside to this last controller, and the sine function controller is they do not restrict the position of q_1, which could cause trouble for the pole-placement balance controller in the next part. However, as can be seen in the video of the bang-bang controller, it ends the swing up in close to the same q_1 position it started in, so if it starts at $q_1 = 0$, it would end there, which is ideal for the balance controller in the next section.

For reference, the full system up to this point is shown in Figure 9.

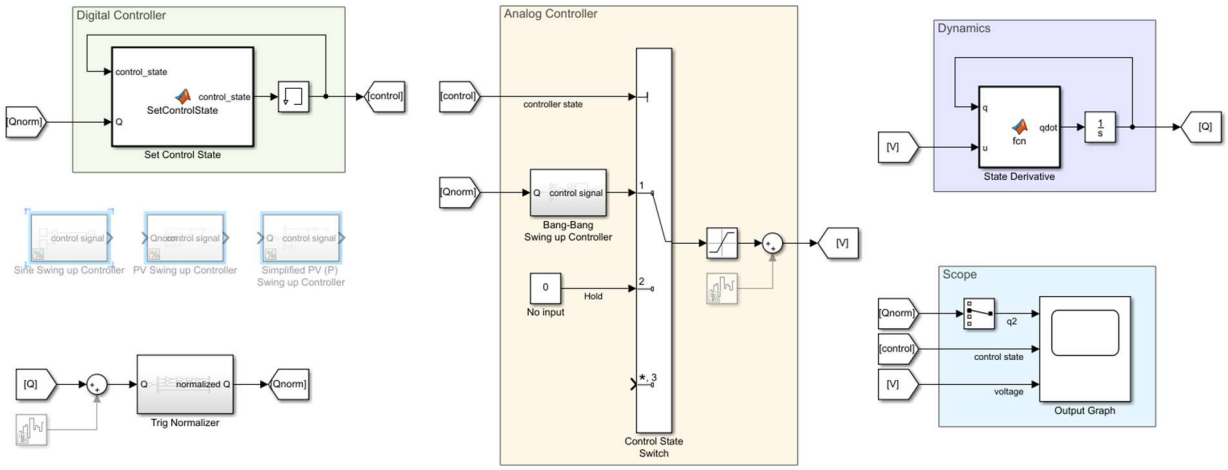

Figure 9: Full System with Swing Up

Part 3: Balance Controller

The goal of this part of the lab is to design a balance controller to keep the pendulum arm vertical once it has been swung up into position.

We did this by using pole-placement state feedback of the system linearized about the vertical stationary point. We calculated a controller gain *K* by placing the poles of the system

$$\dot{x} = (A - BK)x$$

Then, to make the controller, we just feed back the normalized state multiplied by *-K* as the control input to the system. This is shown in Figure 1.

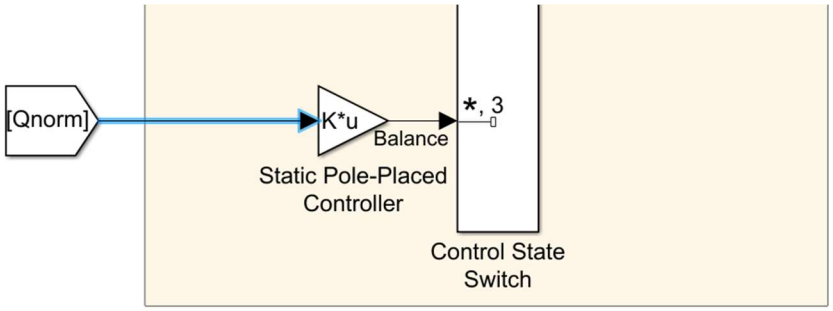

Figure 1: State Feedback Controller

As can be seen from how straightforward the controller design is in Figure 1, the hard part is in calculating the controller gain *K*. To do this we start by linearizing the system. We start with the nonlinear system

$$\dot{x} = f(x, u)$$

and our first step is to take the Jacobian of f for x and u, I will call J_A and J_B. The symbolic version of these matrices is too large and LaTeX editors get mad at me when I try to make them show it. However, we can note the important fact that both Jacobians are related only to the system parameters θ, calculated in a previous lab, and the states Q. This means that if we evaluate them at a point, the resulting matrices only depend on the system parameters, which makes sense. We want to evaluate the Jacobians at the point that we are trying to stabilize it about: [0 0 0 0]. When we do this, we get the *A* and *B* matrices of the linearized system:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{g\,\theta_3\,\theta_4}{(\theta_1\,\theta_2-\theta_3{}^2)} & -\frac{\theta_2\,\theta_5}{\theta_1\,\theta_2-\theta_3{}^2} & \frac{\theta_3\,\theta_6}{\theta_1\,\theta_2-\theta_3{}^2} \\ 0 & \frac{g\,\theta_1\,\theta_4}{(\theta_1\,\theta_2-\theta_3{}^2)} & \frac{\theta_3\,\theta_5}{\theta_1\,\theta_2-\theta_3{}^2} & -\frac{\theta_1\,\theta_6}{\theta_1\,\theta_2-\theta_3{}^2} \end{pmatrix} B = \begin{pmatrix} 0 \\ 0 \\ \frac{\theta_2}{\theta_1\,\theta_2-\theta_3{}^2} \\ -\frac{\theta_3}{\theta_1\,\theta_2-\theta_3{}^2} \end{pmatrix}$$

MATLAB code to create these matrices using the dynamics calculated in a previous assignement is shown below in Figure 2.

```
%% define syms
syms q1 q2 q1dot q2dot u g
states = [q1;q2;q1dot;q2dot];
inputs = [u];

linear_state = [0;0;0;0];
linear_input = [0];
%% constants

syms theta1 theta2 theta3 theta4 theta5 theta6

%% make matrices
m =    [...
```

```
        (theta1+theta2*sin(q2)^2)    (theta3*cos(q2)) ;...
            (theta3*cos(q2))             (theta2)     ;...
];


c =     [...
        (2*theta2*sin(q2)*cos(q2)*q2dot)     (-theta3*sin(q2)*q2dot) ;...
         (-theta2*sin(q2)*cos(q2)*q1dot)              (0)           ;...
];


f =     [...
         theta5*q1dot;...
         theta6*q2dot;...
];

g =     [...
                 0         ;...
        -theta4*g*sin(q2)  ;...
];

v = [u; 0];

%% get accleration
acceleration = -m\(c*[q1dot;q2dot]+f+g-v);

%% get state derivative
qdot = [q1dot;q2dot;acceleration];

%% get jacobian of system for inputs
J_A = jacobian(qdot, states);
J_B = jacobian(qdot, inputs);

%% get linear system
% A = double(round(subs(J_A, [states; inputs], [linear_state;
linear_input])*1000)/1000);
% B = double(round(subs(J_B, [states; inputs], [linear_state;
linear_input])*1000)/1000);
A = subs(J_A, [states; inputs], [linear_state; linear_input])
B = subs(J_B, [states; inputs], [linear_state; linear_input])
```

Figure 2: MATLAB Calculations of Linearized System

From there we simply substitute in the calculated values of θ to get the A and B matrices we will use in calculating our controller gain K:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{3783}{50} & -\frac{51077}{1000} & \frac{81}{100} \\ 0 & \frac{12471}{100} & \frac{51077}{1000} & -\frac{267}{200} \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 0 \\ \frac{10464}{125} \\ -\frac{10464}{125} \end{pmatrix}$$

An important note is that the linearized system does not depend on q_1, which is a good sanity check, because our nonlinear dynamics system also did not depend on q_1 so we should expect that to be the same for the linearized model.

The last step for this is to solve for controller gain that places the eigenvalues of $(A - BK)$ where we desire them. First we should confirm that the system is in fact controllable, which we can do by evaluating the rank of the controllability matrix. We can do this in MATLAB by using the "*ctrb*" command and passing *A, B.* We confirmed that the rank is 4, so the system is controllable. Next, we just use the "*place*" command, passing in *A, B* and the desired poles. I chose the poles $[-5, -6, -7, -8]$ because the poles aren't to large for the controller to handle, but large enough that the system will quickly converge to 0 error. It is also convenient to have the poles close to each other to recreate something similar to critical damping.

The MATLAB code for confirming the controllability and calculating the controller gain is shown below in Figure 3.

```
%% confirm controllability

Ctrb = ctrb(A,B);

rank(ctrb) % rank 4

%% make controller
n = -6;
desired_poles = [n:-1:n-3];

K = place(A,B,desired_poles);

eig(A-B*K) %[-5;-6;-7;-8]
```

Figure 3: Calculating Controller Gain, K

We then just input the controller gain *-K* multiplied by the state as the controller to complete state feedback. The controller working is shown in Figure 4 in graph and video.

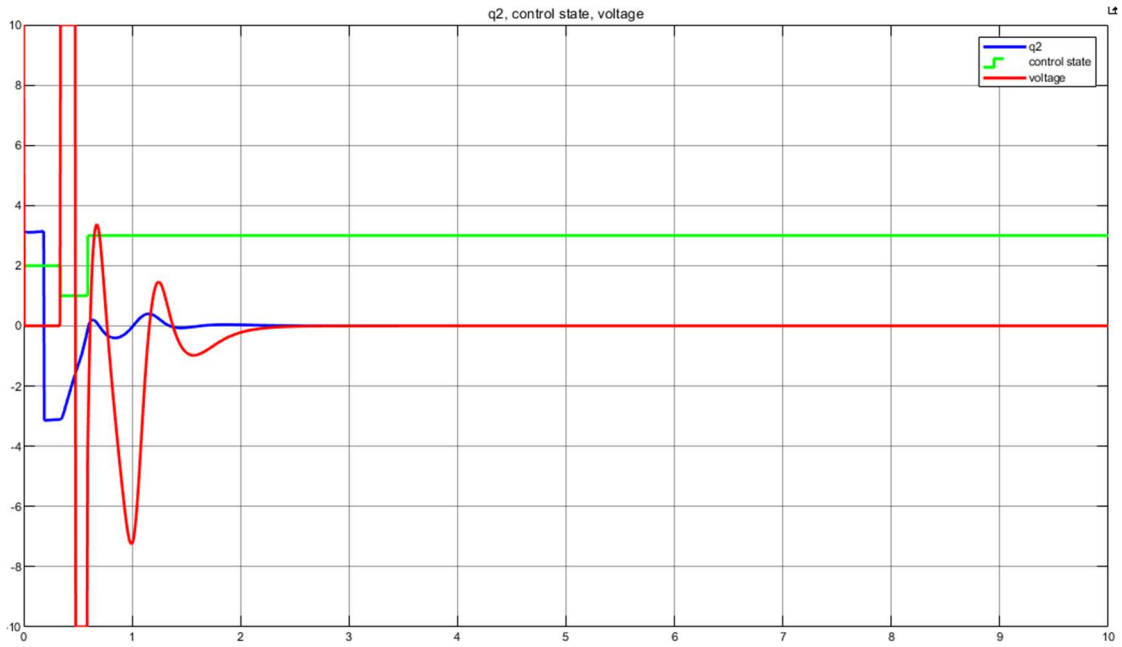

Figure 4: Static State Feedback Controller

We can also note the controller's "priorities" from the gain K:

$$K = [-0.736 \ -6.443 \ -1.020 \ -0.752]$$

where the size of K in each row corresponds to the strength at which it responds to an error in the corresponding state. Therefore, we can see that it responds strongest to q_2, and weakest to q_1. This makes sense because the most important part of balancing the arm is confirming that the arm is vertical.

Part 4: Controlling q_1's Position

The goal of the last part of this lab is to change the controlled position of q_1.

This task is trivial because the dynamics do not depend on q_1 so we can use the same controlled designed in Part 3. The only difference we need to make is instead of feeding back -Q as the error, we feedback $-Q + [q_{1\,desired}; 0; 0; 0]$ as the error. This is done with a simple sum and constant block with slider gain, shown in Figure 1.

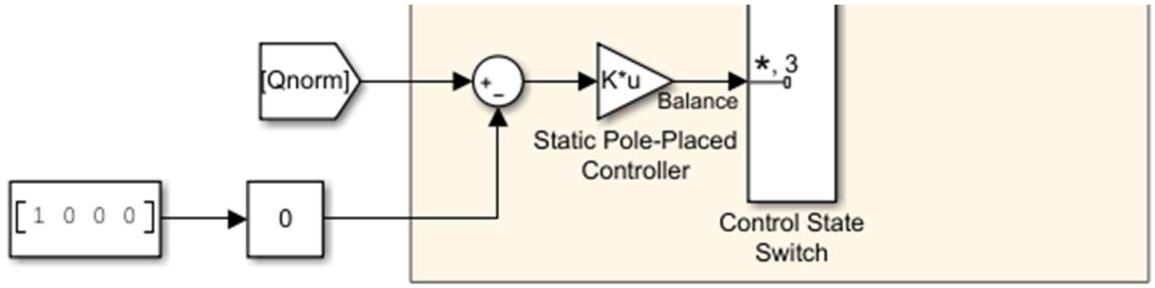

Figure 1: Variable q_1 *Controller*

The controller is shown working in video and graph of a simulation in Figure 2

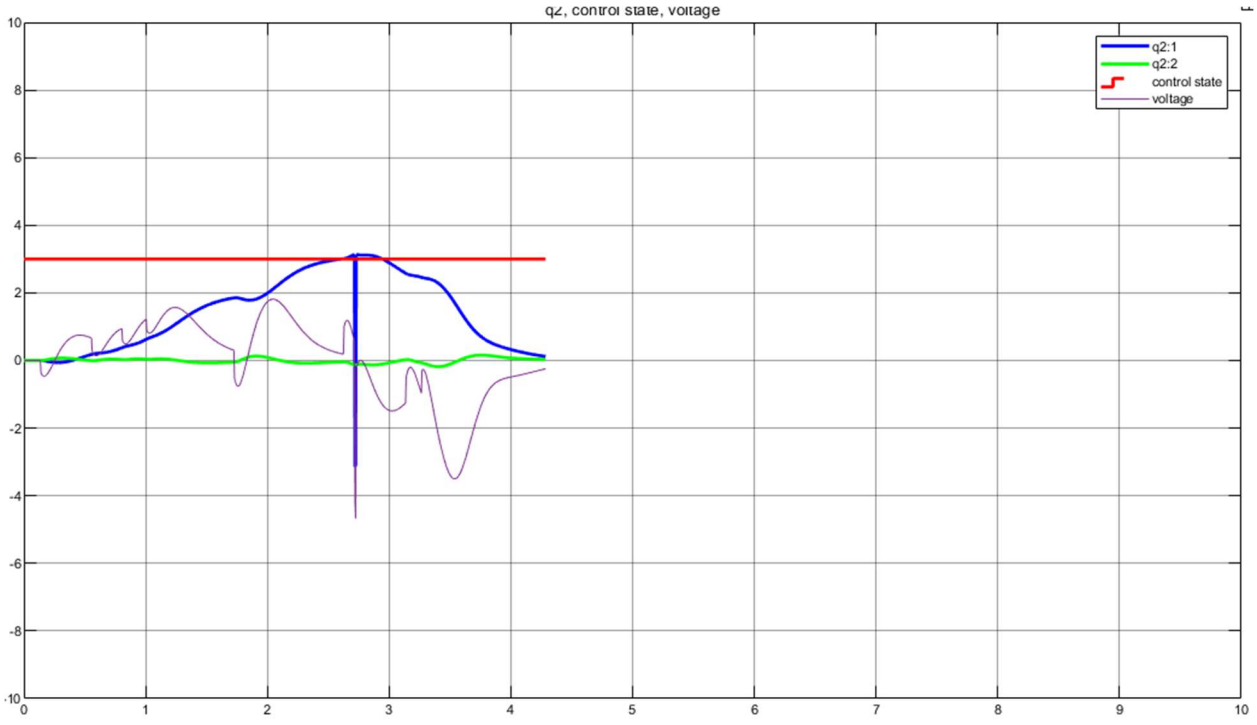

Figure 2: Controlling Position of q_2