Designing a Hybrid Controller:

Task Objectives:
The end desire is to have a robot that will start from a dangling pendulum, swing itself up and balance indefinitely. A different controller will be used for each task in this process, together forming a hybrid controller. Because the hybrid controller will be comprised of multiple different sub controllers, it is necessary to first design the control logic for this hybrid controller that selects between different sub controllers based of the state of robot.

Background:

Hybrid Controllers:
The controller necessary to accomplish function with the robot will vary depending on the function and the state of the model. For example when attempting to balance the robot, the controller needs to be fast, accurate and it needs to account for real world physics and the parameters of the specific robot being used. Things like internal friction and other pesky factors will demand the use of a more complex controller for the balance task. However more simple tasks like swinging the arm up or resetting the robot back to its starting point can make use of much more simplistic controllers that do not account for the impacts of physics and friction etc.

Much like a CPU is composed of datapath and control elements (where different datapath functions all operate in parallel but the selected output is determined by control logic), a basic hybrid controller is comprised of a set of system controllers that run in parallel while a digital controller selects between them based on some control logic.

High Level Overview of Hybrid Controller:
It is best to start with a high level picture of what the controller will look like before defining details about each section. A digital controller will analyze the current states of the robot, and determine which controller fits the situation. It will then send a control signal to a multiport switch to functionally select the output of the appropriate controller. Meanwhile, all controllers are calculating their outputs... but it is only the selected controllers output that makes it through the mux and acts as the control voltage for the system.

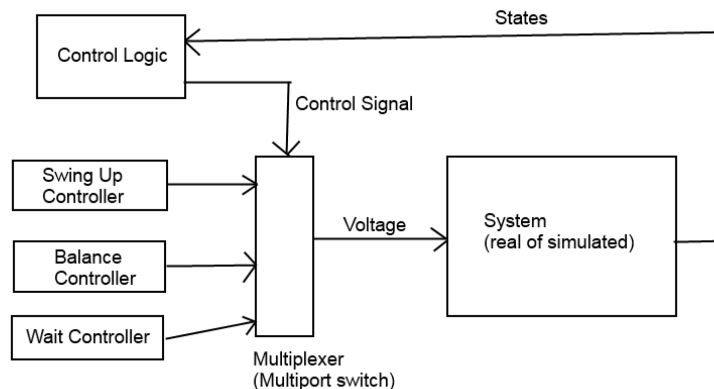

Figure 1: Overview of Hybrid Controller.

Outlining the Entire Hybrid Controller Schematic:
Before going into the details of implementing each element, here is a schematic showing the overview of this specific implementation of the hybrid controller.

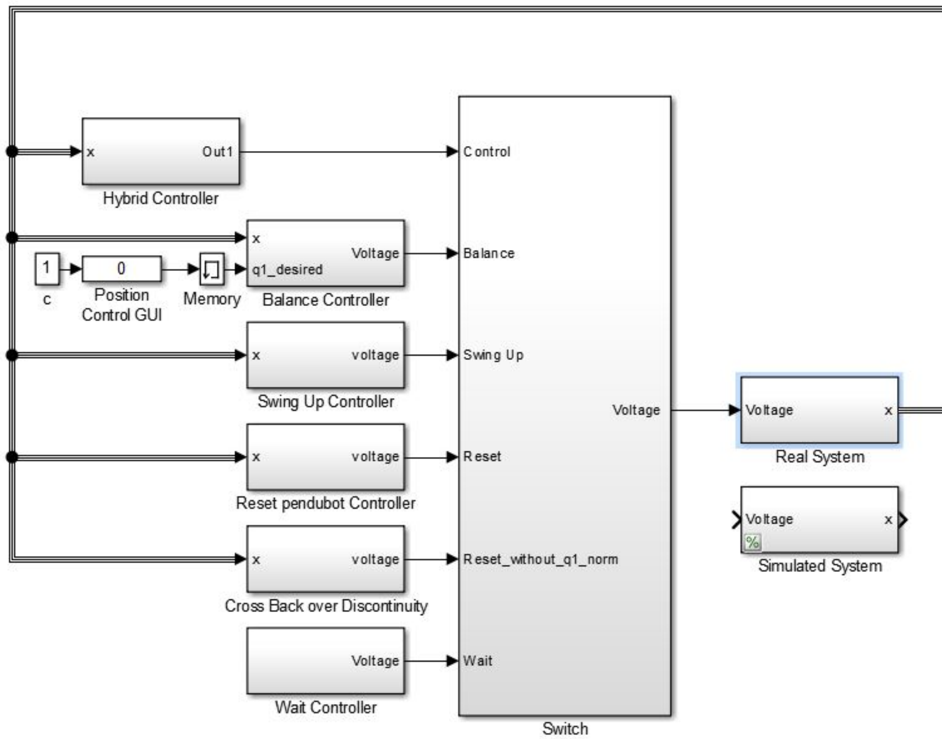

Figure 2: Schematic for the Entire Hybrid Controller.

Implementing a Digital Controller and Control Logic to Switch Between Sub-Controllers:
The first step in creating the digital controller that selects between system controllers in the hybrid, is identifying some criteria on which to base the control logic for the selection. In this case, the criteria will mostly be defined by the current state of q_2. To make it even more simple, define three regions for q_2 that will require different controllers.

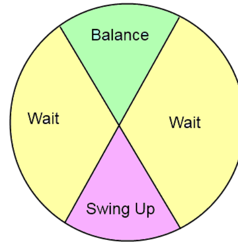

Figure 3: Definable controller regions.

A set of rules defining what changes in controllers are permitted keeps the logic very simple. For the most basic hybrid controller the 3 states Balance, Swing Up and Wait abide by the following logical flow path:

1. Swing Up can only go to Balance
2. Balance can only go to Wait
3. Wait can only to to Swing Up

This simple 3 region design gets the basic functionality described previously. However to design a more robust controller with some extended functionality, it was necessary to include some extra control logic and a few more sub controllers for very specific tasks like resetting the robot and unwinding cables. The final mux for this is shown below:

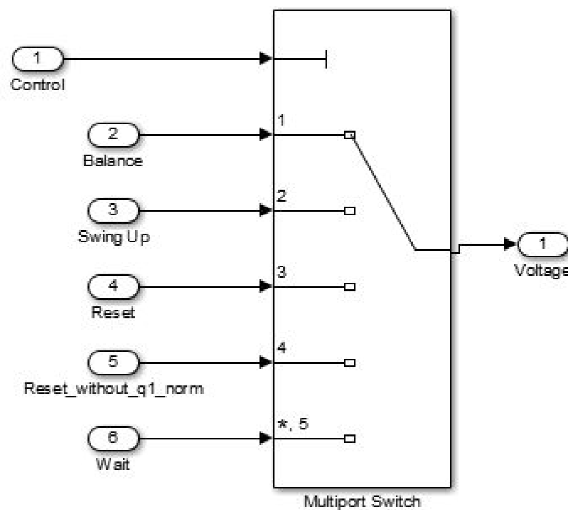

Figure 4: Multi-Port Switch.

The contents of the "hybrid controller box" from the overview schematic (what is essentially the digital controller) is shown below:

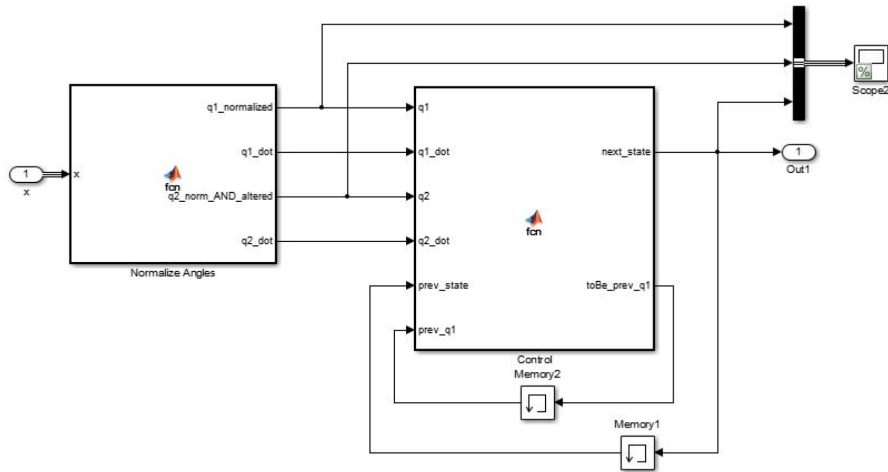

Figure 5: Digital Controller.

The normalization of the incoming angles takes place in a function block. Normalizing angles is important here because our controllers will use the values of q_1 and q_2 in a few equations so having a rolled over angle value will distort its impact and lead to broken controllers. Experimentation with different mod arrangements led to conditionally effective operation of the robot, however for best practice using atan2() function in matlab and feeding in the sine and cosine value of the current angle for the x and y arguments always provided a perfectly normalized result regardless of other circumstances. The contents are shown below:

```
function [q1_normalized,q1_dot, q2_norm_AND_altered,q2_dot] = fcn(x)
%#codegen
q1 = x(1);
q1_dot = x(2);
q2= x(3);
q2_dot = x(4);

x1 = cos(q1);
y1 = sin(q1);
x2 = cos(q2);
y2 = sin(q2);

q1_normalized = atan2(y1,x1);
q2_norm_AND_altered = atan2(y2, x2);

if q2_norm_AND_altered>0
    q2_norm_AND_altered = q2_norm_AND_altered-pi;
else
    q2_norm_AND_altered = q2_norm_AND_altered+pi;
end

end
```

The real control logic or brains of this digital controller (which takes place in the function block) is shown below:

```
function [next_state,toBe_prev_q1] = fcn(q1, q1_dot, q2, q2_dot, prev_state, prev_q1)

    state_reset = 3;
    state_crossBack = 4;
    state_wait = 5;
    state_swingUp = 2;
    state_balance = 1;

    toBe_prev_q1 = q1;

    if prev_state == state_reset
        %keep resetting until q1 and q2 are near 0 and the speeds are low
        if(abs(q1)<(pi/20) && abs(q2)<pi/10 && abs(q1_dot)<0.1 && abs(q2_dot)<0.2)
            next_state = state_swingUp;
        else
            next_state = state_reset;
        end
    elseif prev_state == state_crossBack
        %move q1 back over the discontinuity... then reset as normal
        if(abs(q1)>pi/8) && ((q1*prev_q1)<0)
            next_state = state_reset;
        else
            next_state = state_crossBack;
        end
    elseif prev_state == state_wait
        %keep waiting until q2 has settled back down in both position and speed
        if (abs(q1)>pi/8) && ((q1*prev_q1)<0)
            next_state = state_crossBack;
        else
            if(abs(q2)<pi/10 && abs(q2_dot)<0.35)
               next_state = state_reset;
            else
               next_state = state_wait;
            end
        end
    elseif (prev_state ~= state_swingUp) && (abs(q2)<(pi-35*pi/180))
        %below upper range and not swinging up
        next_state= state_wait;
    elseif prev_state == state_swingUp
        if abs(q1)>(pi/1.5)
            next_state = state_wait;
        elseif abs(q2)>(pi-27.5*pi/180)
            next_state = state_balance;
        else
            next_state = state_swingUp;
        end
    elseif (prev_state == state_balance)
        if (abs(q1)>pi/8) && ((q1*prev_q1)<0)
            next_state = state_crossBack;
        else
            next_state = state_balance;
        end
    else
        next_state = state_wait;
    end

end
```

Implementing the "Wait" Controller:
Implementing the "wait" controller is by far the easiest task in this whole hybrid design. The wait controller is only selected when the digital controller witnesses the robot get knocked out of the balance region. Therefore the only function of the wait controller is to cut all control voltage to the robot so that it can settle back to a hanging pendulum before swing up begins again. This is easily accomplished by simply outputting "0". The entirety of the wait controller looks like this:

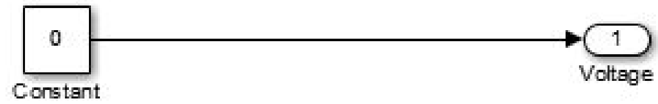

Figure 6: Entirety of the "Wait" Controller.

Implementing the "Swing Up" Controller:
There are multiple strategies to implement a swing up controller. For example a controller could be designed to maximize potential energy. However an easier control scheme borrows heavily from physical intuition. If you think about how you'd swing up the pendulum by hand, you'd jerk q_1 back and forth while looking at q_2 and $\dot{q}_2$. Similarly we can create control scheme as follows:

$$q_{1_{desired}}(t) = K_1 q_2(t) + K_2 \dot{q}_2(t) \qquad \text{(Swing Up Control Scheme)}$$

Having calculated a desired target for q_1, a PV controller can help translate that desired value into a control target. Normally a PD control would compensate an error signal, but that would require adding other elements as a control. But a PV controller can modify existing elements as a method of control. The PV controller is a state-space controller and modifies states with P & V gain in a feedback path. To do so it first deconstructs the states, then modifies them and reconstructs them later.

The PV controller is implemented somewhat like the following, where Kp is the proportional gain:

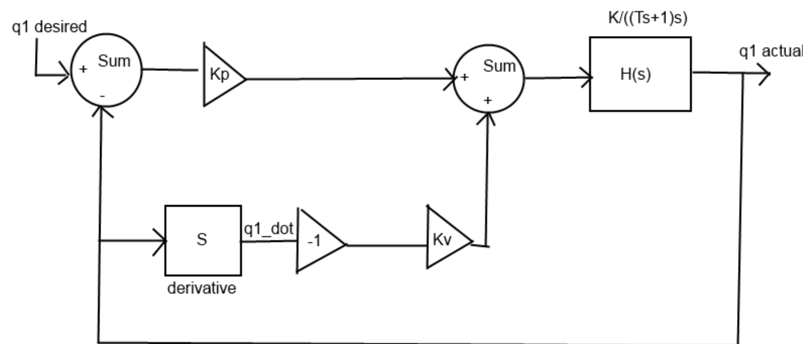

Figure 7: PV controller

$$\frac{q_{1actual}(s)}{q_{1desired}(s)} = \frac{\frac{KK_p}{\tau}}{s^2 + (\frac{1+KK_V}{\tau})s + \frac{KK_p}{\tau}} \tag{1}$$

Because the swingUp controller just needs to get the pendulum up into the balance region, without a terrible amount of accuracy, it is possible to ignore dynamics for the sake of simplicity. This avoids 4th order eqns and drastically simplifies the math involved.

Looking at things theoretically: Given the transfer function H(s) for the servo...

$$\frac{k}{(\tau s + 1)s} = \frac{q_1(s)}{V(s)} \qquad \text{(2nd order)}$$

$$\frac{k}{\tau s + 1} = \frac{\dot{q}_1(s)}{V(s)} \qquad \text{(1st order)}$$

First it is necessary to establish how loose the system can be. Choosing to permit a little bit of overshoot for a responsive system is a fair trade during swingUp. To describe this behavior quantitatively, the system will aim for the following:

$$t_p = 0.15 seconds \qquad \text{(time to peak)}$$

$$m_p = 5\% \qquad \text{(percent overshoot)}$$

The canonical form for a second order system is as follows:

$$H(s) = \frac{K\omega^2}{s^2 + 2\zeta\omega s + \omega^2} \quad \text{(Canonical form 2nd order system)}$$

where K is the system gain, ζ is the damping ratio, and ω is the natural frequency of the system. The damping ratio ζ is a real number that defines damping properties of the system. Damping is defined as "the inherent ability of the system to oppose the oscillatory nature of the system's transient response". A higher damping effect manifests as less percent overshoot and slower settling time. Therefore, larger ζ values produce transient responses with less oscillatory nature. If one finds exact values for ζ and ω, the system time responses can easily be plotted and stability can easily be checked.

So, taking the equations we had and moving them from the time domain into the Laplace frequency domain to match the Canonical 2nd order above, we get the characteristic equation below:

$$\frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad \text{(charactersitic equation)}$$

where ζ = damping ratio and ω_n = natural frequency.

Then we can define the following:

$$m_p = 100e^{-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}} \quad \text{(theoretical } m_p\text{)}$$

$$t_p = \frac{\pi}{\omega_n\sqrt{1-\zeta^2}} \quad \text{(theoretical } t_p\text{)}$$

Provided the values of K = 1.76 and $\tau = 0.0285$. Rearrange the theoretical t_p equation for ζ and rearrange the theoretical m_p equation for ω_n. Then plug in the target values for t_p and m_p defined previously. This yields:

$$\zeta = \frac{-\ln(0.05)}{\sqrt{\pi^2 + \ln^2(0.05)}} = 0.6901 \quad \text{(Damping Ratio } \zeta\text{)}$$

$$\omega_n = \frac{\pi}{0.15\sqrt{1-\zeta^2}} = 28.9398 \quad \text{(natural frequency } \omega_n\text{)}$$

Now comes the key step of finding K_V and K_p by symmetry. Comparing the characteristic equation and equation (1) from the PV controller...

$$\frac{\frac{KK_p}{\tau}}{s^2 + (\frac{1+KK_V}{\tau})s + \frac{KK_p}{\tau}} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (2)$$

...yields equations useful in solving for K_V and K_p.

$$\frac{1+KK_V}{\tau} = 2\zeta\omega_n \quad \text{(eqn to solve for } K_V\text{)}$$

$$\frac{KK_p}{\tau} = \omega_n^2 \quad \text{(eqn to solve for } K_p\text{)}$$

$$K_p = \frac{\omega_n^2 \tau}{K} = \frac{28.94^2 * 0.0285}{1.76} = 13.5620 \qquad \text{(solved for } K_V\text{)}$$

$$K_V = \frac{2\zeta\omega_n \tau - 1}{K} = 0.07862 \qquad \text{(solved for } K_p\text{)}$$

Implementing this controller becomes simple once we have the values of K_V and K_p. The schematic for the swingUp controller is shown below:

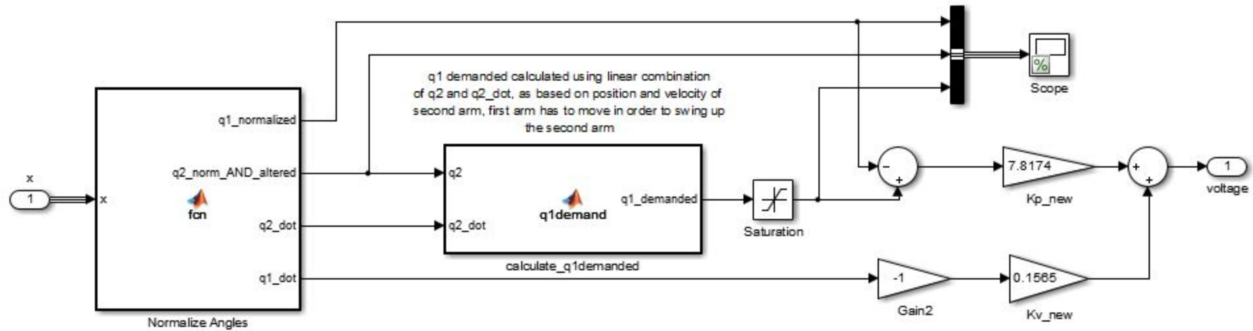

Figure 8: Swing Up Controller

The normalization of the incoming angles takes place in a function block. Here q_2 has to be altered such that 0 is centered at the bottom to avoid discontinuities in q_2 around the bottom where the swing up controller operates. The contents of the normalization block are shown below:

```
function [q1_normalized,q2_norm_AND_altered,q2_dot,q1_dot] = fcn(x)
%#codegen
q1 = x(1);
q1_dot = x(2);
q2= x(3);
q2_dot = x(4);

x1 = cos(q1);
y1 = sin(q1);
x2 = cos(q2);
y2 = sin(q2);

q1_normalized = atan2(y1,x1);
q2_norm_AND_altered = atan2(y2, x2);

if q2_norm_AND_altered>0
    q2_norm_AND_altered = q2_norm_AND_altered-pi;
else
    q2_norm_AND_altered = q2_norm_AND_altered+pi;
end

end
```

Additionally the calculation of $q_{1demanded}$ occurs in a function block whose contents are as follows:

```
function q1_demanded = q1demand(q2,q2_dot)
%Uses PD Controller to calculate demanded q1 using q2 and q2_dot
P = 1;
D = 0.02;
q1_demanded = P*asin(sin(q2))+D*q2_dot;
```

Implementing the "Balance" Controller:
The next task is implementing a balance controller to keep the inverted pendulum robot balanced. This controller will be selected only when the robot is coming from the swing up phase and q_2 is in the top "balance" region. The controller will be a little more complex than the others because the balancing act requires consideration of physics, friction, gravity etc. We will use the model of the system derived in previous sections and linearize it about a point (set of state values). The target values are actually trivial to define. For a balanced robot, q_2 will be 0 in a system where 0 is defined as the top of our coordinate space for q_2.
Starting from a simple state space model:

$$\dot{x} = Ax + Bu \qquad \text{(State Space Model)}$$

where "A" is a 4x4 matrix, "x" is a 4x1 matrix of the states, "B" is a 4x1 matrix and u is our input voltage.

The Jacobian A is defined as below:

$$A = \begin{bmatrix} \frac{\partial \dot{q}_1}{\partial q_1}\Big|_{x_0} & \frac{\partial \dot{q}_1}{\partial \dot{q}_1}\Big|_{x_0} & \frac{\partial \dot{q}_1}{\partial q_2}\Big|_{x_0} & \frac{\partial \dot{q}_1}{\partial \dot{q}_2}\Big|_{x_0} \\ \frac{\partial \ddot{q}_1}{\partial q_1}\Big|_{x_0} & \frac{\partial \ddot{q}_1}{\partial \dot{q}_1}\Big|_{x_0} & \frac{\partial \ddot{q}_1}{\partial q_2}\Big|_{x_0} & \frac{\partial \ddot{q}_1}{\partial \dot{q}_2}\Big|_{x_0} \\ \frac{\partial \dot{q}_2}{\partial q_1}\Big|_{x_0} & \frac{\partial \dot{q}_2}{\partial \dot{q}_1}\Big|_{x_0} & \frac{\partial \dot{q}_2}{\partial q_2}\Big|_{x_0} & \frac{\partial \dot{q}_2}{\partial \dot{q}_2}\Big|_{x_0} \\ \frac{\partial \ddot{q}_2}{\partial q_1}\Big|_{x_0} & \frac{\partial \ddot{q}_2}{\partial \dot{q}_1}\Big|_{x_0} & \frac{\partial \ddot{q}_2}{\partial q_2}\Big|_{x_0} & \frac{\partial \ddot{q}_2}{\partial \dot{q}_2}\Big|_{x_0} \end{bmatrix} \qquad (3)$$

Knowing that $\dot{q}_1$ only dependent on $\dot{q}_1$, the partial derivatives of $\dot{q}_1$ are simply 0 other than the partial with respect to itself which is 1. This turns the first row of the Jacobian into [0 1 0 0]. Similarly the third row is obviously just [0 0 0 1]. Additionally none of the parameters depend on q_1. This leaves the only values left to find shown as "?".

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & ? & ? & ? \\ 0 & 0 & 0 & 1 \\ 0 & ? & ? & ? \end{bmatrix} \qquad (4)$$

To find these "?" values, first recall that:

$$\big[m(q)\big]\begin{bmatrix}\ddot{q}_1 \\ \ddot{q}_2\end{bmatrix} + \big[c(q,\dot{q})\big]\begin{bmatrix}\dot{q}_1 \\ \dot{q}_2\end{bmatrix} + \big[f(\dot{q}_1)\big] + \big[g(q)\big] = \begin{bmatrix} v \\ 0 \end{bmatrix} \qquad (5)$$

Rearranging for $\ddot{q}_1$ and $\ddot{q}_2$ yields:

$$\begin{bmatrix}\ddot{q}_1 \\ \ddot{q}_2\end{bmatrix} = M^{-1}[-C\begin{bmatrix}\dot{q}_1 \\ \dot{q}_2\end{bmatrix} - F - g + \begin{bmatrix} v \\ 0 \end{bmatrix}] \qquad (6)$$

These values of $\ddot{q}_1$ and $\ddot{q}_2$ can be differentiated to find the "?" values for rows two and four of the Jacobian matrix.

Additionally the point to linearize about is easy to define, technically q_2 is the important parameter here and values for other parameters can differ but leaving them all 0 makes math easy.

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad (7)$$

The equations for $\ddot{q_1}$ and $\ddot{q_2}$ still need to be defined in terms of theoretical or measured system parameters $\theta_1 - \theta_6$. Recall the following were found previously:

$$M(q) = \begin{bmatrix} \theta_1 + \theta_2 \sin(q_2)^2 & \theta_3 \cos(q_2) \\ \theta_3 \cos(q_2) & \theta_2 \end{bmatrix} \tag{8}$$

$$C(q, \dot{q}) = \begin{bmatrix} 2\theta_2 \sin(q_2)\cos(q_2)\dot{q_1}\dot{q_2} & -\theta_3 \sin(q_2)\dot{q_2} \\ -\theta_2 \sin(q_2)\cos(q_2)\dot{q_1} & \theta_6 \end{bmatrix} \tag{9}$$

$$f(\dot{q_1}) = \begin{bmatrix} \theta_5 \dot{q_1} \\ 0 \end{bmatrix} \tag{10}$$

$$g(q) = \begin{bmatrix} 0 \\ \theta_4 g \sin(q_2) \end{bmatrix} \tag{11}$$

A simple Matlab script can help make short work of find the inverse of these matrices, and then differentiating them with respect to $q_2, \dot{q_1}, \dot{q_2}$ to yield symbolic equations for the partial derivatives replacing the "?'s" in the Jacobian matrix A.

```
syms theta1  theta2 theta3 theta4 theta5 theta6 q1 q2 q1_dot q2_dot grav q1_dd q2_dd v
M = [theta1+theta2*(sin(q2))^2 theta3*cos(q2);
    theta3*cos(q2) theta2];
C = [2*theta2*sin(q2)*cos(q2)*q2_dot -theta3*sin(q2)*q2_dot;
    -theta2*sin(q2)*cos(q2)*q1_dot theta6];
F = [theta5*q1_dot;
    0];
G = [0;
    theta4*grav*sin(q2)];
M1 = inv(M);
C1 = -M1*C;
F1 = -M1*F;
G1 = -M1*G;
V1 = M1*[v;0];

q1_dd = C1(1,1)*q1_dot + C1(1,2)*q2_dot + F1(1,1) + G1(1,1) + V1(1,1);
q2_dd = C1(2,1)*q1_dot + C1(2,2)*q2_dot + F1(2,1) + G1(2,1) + V1(2,1);

temp1 = diff(q1_dd, q1_dot);
temp1_1= subs(temp1, [q1 q1_dot q2 q2_dot], [0 0 0 0])
temp2 = diff(q1_dd, q2);
temp2_1= subs(temp2, [q1 q1_dot q2 q2_dot], [0 0 0 0])
temp3 = diff(q1_dd, q2_dot);
temp3_1= subs(temp3, [q1 q1_dot q2 q2_dot], [0 0 0 0])

temp4 = diff(q2_dd, q1_dot);
temp4_1= subs(temp4, [q1 q1_dot q2 q2_dot], [0 0 0 0])
temp5 = diff(q2_dd, q2);
temp5_1= subs(temp5, [q1 q1_dot q2 q2_dot], [0 0 0 0])
temp6 = diff(q2_dd, q2_dot);
temp6_1= subs(temp6, [q1 q1_dot q2 q2_dot], [0 0 0 0])
```

After solving symbolically with the previous script, the equations will be readable by eye and writing down a theoretical Jacobian matrix yields the following, where $d = \theta_1\theta_2 - \theta_3^2$.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-\theta_2\theta_5}{d} & \frac{g\theta_3\theta_4}{d} & \frac{\theta_3\theta_6}{d} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{\theta_3\theta_5}{d} & \frac{-g\theta_1\theta_4}{d} & \frac{-\theta_1\theta_6}{d} \end{bmatrix} \tag{12}$$

Similarly, factoring out voltage from one of the previous equations has lead to the B matrix:

$$\begin{bmatrix} 0 \\ \frac{\theta_2}{d} \\ 0 \\ \frac{-\theta_3}{d} \end{bmatrix} \tag{13}$$

Finally, subbing in the experimentally determined values for $\theta_1 - \theta_6$ can be accomplished with a secondary script.

```
theta1 = 0.0751;
theta2 = 0.0292;
theta3 = 0.0260;
theta4 = 0.1376;
theta5 = 0.5381;
theta6 = 0.0066;
syms grav q1 q2 q1_dot q2_dot q1_dd q2_dd v
M = [theta1+theta2*(sin(q2))^2 theta3*cos(q2);
    theta3*cos(q2) theta2];
C = [2*theta2*sin(q2)*cos(q2)*q2_dot -theta3*sin(q2)*q2_dot;
    -theta2*sin(q2)*cos(q2)*q1_dot theta6];
F = [theta5*q1_dot;
    0];
G = [0;
    theta4*grav*sin(q2)];
M1 = inv(M);    C1 = -M1*C;     F1 = -M1*F;     G1 = -M1*G;     V1 = M1*[v;0];

q1_dd = C1(1,1)*q1_dot + C1(1,2)*q2_dot + F1(1,1) + G1(1,1) + V1(1,1);
q2_dd = C1(2,1)*q1_dot + C1(2,2)*q2_dot + F1(2,1) + G1(2,1) + V1(2,1);

temp1 = diff(q1_dd, q1_dot);
temp1_1= subs(temp1, [q1 q1_dot q2 q2_dot grav], [0 0 0 0 -9.81])
temp2 = diff(q1_dd, q2);
temp2_1= subs(temp2, [q1 q1_dot q2 q2_dot grav], [0 0 0 0 -9.81]);
temp2_1=-3256176278324243/140737488355328
temp3 = diff(q1_dd, q2_dot);
temp3_1= subs(temp3, [q1 q1_dot q2 q2_dot grav], [0 0 0 0 -9.81])

temp4 = diff(q2_dd, q1_dot);
temp4_1= subs(temp4, [q1 q1_dot q2 q2_dot grav], [0 0 0 0 -9.81])
temp5 = diff(q2_dd, q2);
temp5_1= subs(temp5, [q1 q1_dot q2 q2_dot grav], [0 0 0 0 -9.81]);
temp5_1=4702669971195205/70368744177664
temp6 = diff(q2_dd, q2_dot);
temp6_1= subs(temp6, [q1 q1_dot q2 q2_dot grav], [0 0 0 0 -9.81])

A = [0 1 0 0;
    0 temp1_1 temp2_1 temp3_1;
    0 0 0 1;
    0 temp4_1 temp5_1 temp6_1];

B = [0; theta2/(theta1*theta2-theta3^2); 0; -theta3/(theta1*theta2-theta3^2)];
system_poles = eig(A)
K = place(A, B, [-5 -5.1 -5.2 -5.3])
```

Recall that the system in use is modeled with $\dot{x} = Ax + Bu$. The system poles can be found by finding the eigen values of the Jacobian matrix A just defined. Matlab has a handy function we can use, and a simple command "eig(A)" yields the following poles:

$$0, \ -12.8005, \ -5.1492, \ 7.2648 \qquad (\dot{x} = Ax + Bu \text{ system poles})$$

A quick observation will catch a potential problem. The last pole (7.2648) is positive. e^x will blow up if x is positive and converge if x is negative. Without all negative poles the system is an unstable open loop. It will not converge. To close the loop it is possible to functionally change A to a different matrix A-Bk which places all the poles in the left hand plane so that the system is not unstable. The new system will look like this:

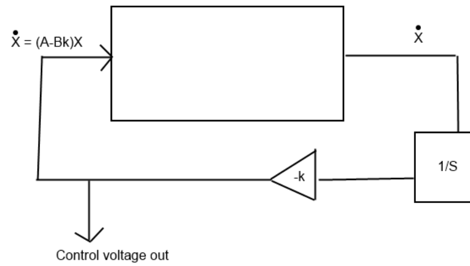

Figure 9: Altered System to shift poles.

There is a wonderful Matlab function called "place" which makes this process remarkably simple. The function provides a gain k (shown in the system above) so that the poles will be shifted to a specific place. In this case the shifted locations are set to [-5,-5.1,-5.2,-5.3]. The "place" function yields the following k vector.

$$k = \begin{bmatrix} -0.7898, & -1.557, & -14.2257, & -1.8764 \end{bmatrix} \qquad \text{(k vector)}$$

The states $X = [q_1, \dot{q_1}, q_2, \dot{q_2}]$ are fed directly into this gain. The magnitude of each element in the k vector indicates significance of the elements in X. Logically the third value, corresponding to q_2 is has the largest magnitude because the balance controller focuses mostly on q_2 when balancing.

The final schematic for the balance controller is shown below:

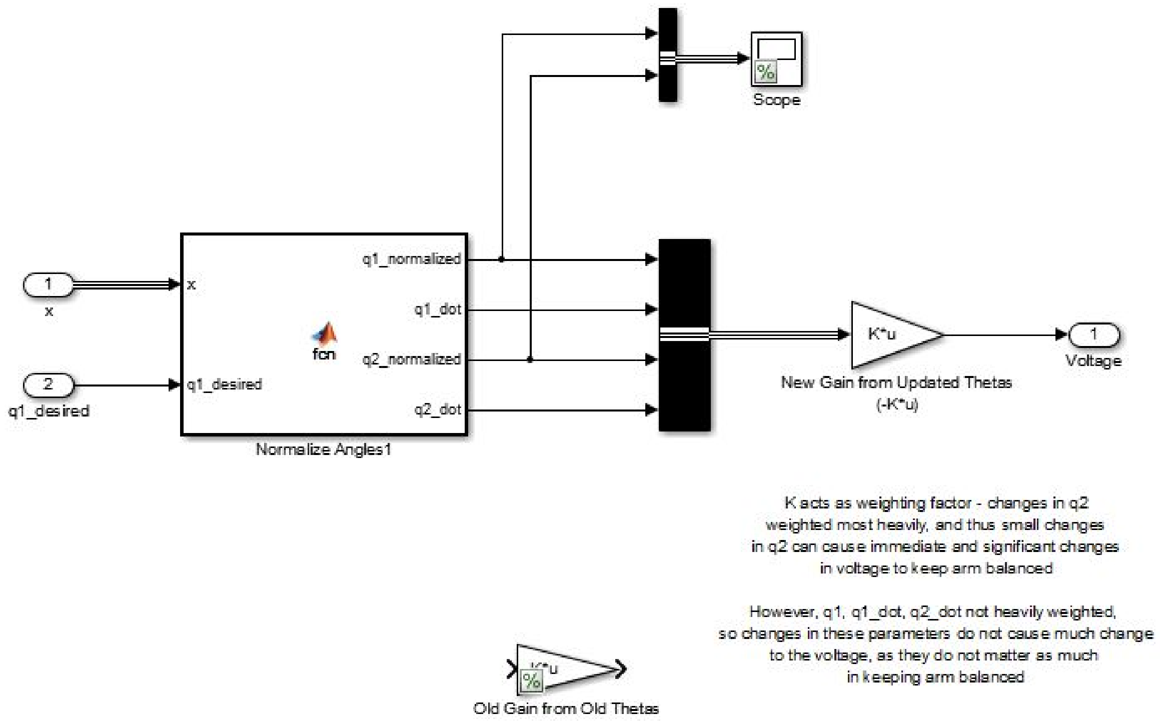

Figure 10: Final Balance Controller.

The contents of the normalize block are shown below:

```
function [q1_normalized,q1_dot,q2_normalized,q2_dot] = fcn(x, q1_desired)
%#codegen
q1 = x(1);
q1_dot = x(2);
q2= x(3);
q2_dot = x(4);

x1 = cos(q1);
y1 = sin(q1);
x2 = cos(q2);
y2 = sin(q2);

q1_normalized = atan2(y1,x1)-q1_desired;
q2_normalized = atan2(y2, x2);
end
```

Experimenting with Pole Locations:
A pole is a point of impossibility in a system where if things approach that point, the value of the system approaches infinity. Mathematically a pole is defined as a value of s where the denominator of the transfer function becomes 0. The locations of the poles, and the values of the real and imaginary parts of the pole determine the response of the system. Real parts correspond to exponentials, and imaginary parts correspond to sinusoidal values. For a stable system, all poles must be negative (ie: in the Left Half Plane) because an exponential function with positive poles makes the system approach infinity. We want it to converge to 0 for stability. The order of the poles do not matter. In this case, the Matlab "place" function cannot support identical pole placements. Because of this the poles need to be different (can't support multiplicities).

In order to observe the effects of various pole locations, the following K vectors were generated and the physical robot was observed using the K vector in the balance controller.

```
K(1:4) = place(A, B, [-5 -5.1 -5.2 -5.3])
%Resultant K:  -0.7827   -1.1504  -13.7885   -1.8155

K(1:4) = place(A, B, [-2 -2.1 -2.2 -2.3])
%Resultant K:  -0.0237   -0.5819   -5.3698   -0.5114

K(1:4) = place(A, B, [-10 -10.1 -10.2 -10.3])
%Resultant K:  -11.8172   -5.2659  -53.0948   -7.4857

K(1:4) = place(A, B, [-1 -3 -5 -7])
%Resultant K:  -0.1169   -0.7343   -8.8055   -1.0968

K(1:4) = place(A, B, [-5 -7 -9 -11])
%Resultant K:  -3.8589   -2.6633  -29.7584   -4.1281

K(1:4) = place(A, B, [-7 -5 -3 -1])
%Resultant K:  -0.1169   -0.7343   -8.8055   -1.0968

K(1:4) = place(A, B, [-5+1i -5-1i -5 -5.1])
%Resultant K:  -0.7384   -1.1184  -13.3572   -1.7520
```

The observations are listed below:

- With the smaller pole values (-2, -2.1, -2.2, -2.3), the system could not counter quick enough to overcome the initial jerk up.
- With the bigger pole magnitudes (-10,-10.1,-10.2,-10.3) Unexpected behavior. The system was very unstable. Looking at the output K, the value corresponding to q1 is weighed in substantially now. This might be why the system is so unstable, because q1 shouldn't have much effect at all.
- With varied poles (-1,-3,-5,-7), the system was able to balance, but it requires a larger q1 distance for counter balance maneuvers. This might be because q1 is 7 times less weighted relatively than it was in the original.
- With varied poles (-5,-7,-9,-11) the system was not stable at all.
- With varied poles (-5+i, -5-i, -5, -5.1), The system balanced every time, which was expected given the identical real portions of the poles to the original. However the system oscillated a little more than the original which was expected with the presence of the sinusoidal component influence from the imaginary.

Implementing a "Reset" Controller:

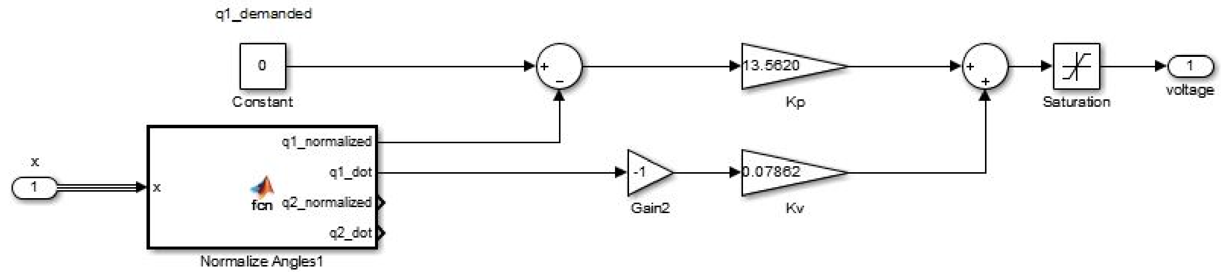

Figure 11: Reset Controller.

The contents of the normalize block are shown below:

```
function [q1_normalized,q1_dot,q2_normalized,q2_dot] = fcn(x)
%#codegen
q1 = x(1);
q1_dot = x(2);
q2= x(3);
q2_dot = x(4);

x1 = cos(q1);
y1 = sin(q1);
x2 = cos(q2);
y2 = sin(q2);

q1_normalized = atan2(y1,x1);
q2_normalized = atan2(y2, x2);
end
```

Preventing Cable Winding and q_1 Rollover:

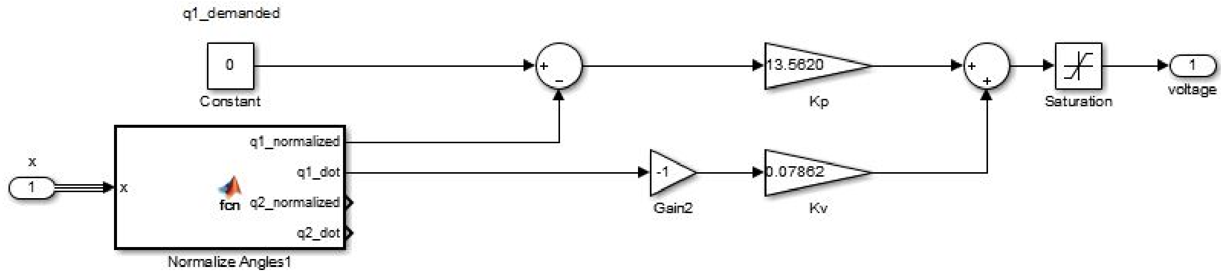

Figure 12: Crossover Discontinuity Controller to Unwind Cables.

The contents of the normalize block are shown below. For this controller, most of the logic happens here.

```
function [q1_normalized,q1_dot,q2_normalized,q2_dot] = fcn(x)
    %#codegen
    q1 = x(1);
    q1_dot = x(2);
    q2 = x(3);
    q2_dot = x(4);

    x1 = cos(q1);
    y1 = sin(q1);
    x2 = cos(q2);
    y2 = sin(q2);

    q2_normalized = atan2(y2,x2);
    q1_normalized = atan2(y1,x1);

    %ignore discontinuity, permit q1 to be between -2pi:2pi instead of -pi:pi
    if(q1_normalized < 0)
       q1_normalized = pi - q1_normalized;
    else
        q1_normalized = -pi - q1_normalized;
    end
end
```

Controlling the Robot with a Kinect Sensor

What better fun is there, than controlling a robot by waving your hands. There were a few design considerations that had to be fleshed out however.

- The robot hardware was all setup with a computer running Matlab r2012b which makes implementing data acquisition for the kinect sensor a much more tedious process involving 3rd party solutions. Matlab r2013+ introduced simpler implementations.
- I had access to a computer running Matlab r2013b, but conducting the data acquisition on a different PC meant communicating that information over a network.
- Assuming the network connection was operational in Matlab, there were still two functions that needed to execute simultaneously... 1) Simulink model controlling the robot... and 2) the client script receiving updated info from the server. Also the robot controller would still have to run lightning fast, whereas the updated commands from the server could come in relatively slowly (20/second). Without being able to put the call in the control loop itself, the solution solution would have to involve two asynchronously parallel function. Of course Matlab doesn't lend itself to simple asynchronous parallelism. Although there are some workarounds.[I later discovered this was unnecessary, as the model was running externally and a simpler solution would suffice]

Drafting a Server to Acquire Kinect Data

```
%%%%%%%%      Rough SERVER Draft    %%%%%%%%%%%%
close all; clear all; clc;

%define data to send over network as a 1x1 array (for simple command code)
dataPackage = zeros(1,1);
s = whos('dataPackage')

%define server and open it
tcpipServer = tcpip('0.0.0.0',55000,'NetworkRole','Server');
set(tcpipServer,'OutputBufferSize',s.bytes);
display('Initialized Server');
fopen(tcpipServer);
display('Acquired Client, beginning Kinect Setup');

%setup path for kinect stuff
utilpath = fullfile(matlabroot, 'toolbox', 'imaq', 'imaqdemos', ...
    'html', 'KinectForWindows');
addpath(utilpath);

% The Kinect for Windows Sensor shows up as two separate devices in IMAQHWINFO.
hwInfo = imaqhwinfo('kinect');

% Create the VIDEOINPUT objects for the depth streeam, which is the 2nd device
depthVidOBJ = videoinput('kinect',2);

depthVidOBJ.TriggerRepeat = Inf;
depthVidOBJ.FrameGrabInterval = 1;

% Get the VIDEOSOURCE object from the depth device's VIDEOINPUT object.
depthSrc = getselectedsource(depthVidOBJ);

% Turn on skeletal tracking.
depthSrc.TrackingMode = 'Skeleton';

%turn on the stream
start(depthVidOBJ)

%we will always be dealing with the first frame since we deal with them one at a time here
frameInQuestion=1;

acquireCount = 50; %determines how long the data acquisition will continue for
fprintf('Initiating Data Acquisition\n\n');
while(1)%acquireCount >=0)
        %acquire the data
        [frameDataDepth, timeDataDepth, metaDataDepth] = getdata(depthVidOBJ);

        anySkeletonsTracked = any(metaDataDepth(frameInQuestion).IsSkeletonTracked ~= 0);
        if(anySkeletonsTracked)
            % See which skeletons were tracked.
            trackedSkeletons = find(metaDataDepth(frameInQuestion).IsSkeletonTracked);

            % Skeleton's joint indices with respect to the color image
            jointIndices = metaDataDepth(frameInQuestion).JointImageIndices(:, :, trackedSkeletons);

            %define specific joint Indices
            rightHand= jointIndices(8,:);
            leftHand= jointIndices(12,:);


            %check for control gestures (currently simple 2 hand control gestures)
            if(abs(rightHand(2)-leftHand(2))>25) %are the hands around the same heigh
                if(rightHand(2)>leftHand(2))
                    fwrite(tcpipServer,dataPackage(:),'double');
                    display('Move Robot Right');
                    dataPackage(1)=2;
                else
                    display('Move Robot Left');
```

```
                    dataPackage(1)=1;
                end
            else
                display('Keep Robot In Place');
                dataPackage(1)=0;
            end
            fwrite(tcpipServer,dataPackage(:),'double');
        else
            display('No skeleton detected');
        end
    acquireCount = acquireCount-1;
end

stop(depthVidOBJ)
fprintf('\nDone Acquiring Data\n');


fclose(tcpipServer);
display('Successfully closed server');
```

Drafting an Asynchronous Parallel Client to Retrieve Data After spending a great deal of time experimenting with asynchronous parallelism in Matlab (including getting into some java threading behind the scenes) it dawned on me that the controller was actually being run externally. This meant providing instructions to the model transformed into a dramatically simple task. To take care of the updates I used a parameter set command in the client that targeted a variable in the model and executed on update from the server.

```
%%%%%%%%%%%%%%%    CLIENT     %%%%%%%%%%%
clear all; close all; clc;
tcpipClient = tcpip('172.16.1.71',55000,'NetworkRole','Client') %for
% server running on kinect computer

set(tcpipClient,'InputBufferSize',8);
set(tcpipClient,'Timeout',1000);
fopen(tcpipClient);
while(1)
    rawData = fread(tcpipClient,1,'double');
    display(rawData);
    if rawData == 0
        set_param('Hybrid_Controller/c','Value','0');
    elseif rawData == 1
        set_param('Hybrid_Controller/c','Value','1.57');
    else
        set_param('Hybrid_Controller/c','Value','-1.57');
    end
end
fclose(tcpipClient);
display('successfully retrieved data from server');
```

Task Chart:

Task	Unresolved Issues
Simulink Pallet Familiarization	None
Read from Encoders	None
Use sine input w/ good frequency to make sinusoid Scope Reading	None
Low Pass Filter between the gain and the scope	None
Oscillate between - and + 45 degrees with switching sign on a constant voltage whenever reaching the barriers of 45 degrees	None
Oscillate Movement via PID	None
Create simple PID controller with a simple repeating step input	None
Oscillate the system with various waveforms	None
Move the link to any given position and hold that position	None
Created a GUI with 3 selectable modes with controls to move to 0, to position X or to oscillate between + and - X.	None
Derive math for transforming coordinates between frames.	None
Defined all points of the robot in their initial respective frames	None
Draw the base of the robot in Frame-0	None
Draw the first link of the robot in Frame 1	None
Convert Points Representing the First Link	None
Convert Points Representing the Second Link	None
Draw the entire robot in Frame 0	None
Simulate and Animate the Robot Given an input theta value	None
Define the work envelope	None
Animate the robot w a work envelope Overlay	None
Simulate and animate the end effector path	None
Test the Simulation	None

Task Chart Continued:

Derive Matrices (M,C,f and g)	None
Derive equations for robot parameters theta1-theta6 provided physical parameters for the robot.	None
Conduct previous simulations with updated physics considerations	None
Generate Data for Measured Voltage and States	None
Find experimental theta values from measured robot state data	None
Test experimental parameters	None
Implement a digital controller and control logic to switch between sub-controllers (glorified multiplexer)	None
Implement a wait controller with 0 voltage input	None
Derive Kv and Kp for the swing up controller	None
Implement a swing up controller	None
Derive A and B matrices for the Balance Controller	None
Shift the poles to the LHP with a functional gain K	None
Implement a Balance Controller	None
Experiment with Pole Placements	None
Implement a Reset Controller	None
Prevent Cable Winding with a q1 rollover	None
Control the Robot with a kinect sensor	None
Elaborate on Gesture Control and Network Functionality and Reliability	None