Jimmy Paul

Kevin Lynch/Nelson Rosa

NxR lab Gibbot

June 11, 2013

Gibbot Vision System and Balancing Controller

Summary

Throughout this quarter I integrated the Gibbot's vision system with the old Gibbot, helped design a simulation that mimicked the Gibbot's dynamics, and designed a balancing controller for the robot. I was able to successfully demo the balancing controller on the computer simulation of the robot however was never able to test either the swing up or balancing controller on any physical robots. Unfortunately the electronic system of the new robot is not yet completed and the old Gibbot is not functioning properly. Nelson and myself have spent a significant amount of time debugging the old Gibbot and have yet to track down the issue. Our findings are detailed later in this paper.

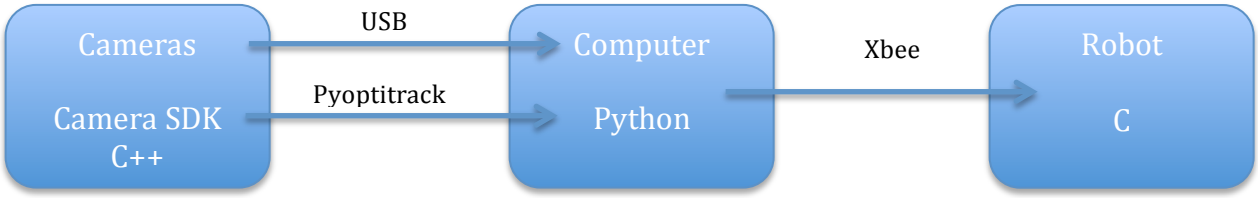
Table of Contents

Vision System

Overview:

The vision system we created uses two OptiTrack Flex 13 infrared cameras to cover a 6 by 8 foot frame. Factory settings leave the cameras running at 120 fps and use on board LEDs to track the positions of reflective objects. However, we found that relying on passive lighting often led to noise in the system (The cameras often picked up reflections from the floor as objects). In order to prevent this we turned off the camera's LEDs and used active lighting on both the steel frame and robot to track it's positioning (Wide Angle IR 850 nm LEDs). We programmed our system to pick up each LED as an object and send them through a perspective transform using the outside markers as landmarks in order to give us real world coordinates. In order to differentiate between the robots links we placed three LEDs over one of the robots magnets, two LEDs over the other magnet, and one LED in the center of the Gibbot. We then ran a grouping algorithm that ignores the 6 LEDs representing the outer edges of the metal frame and groups, averages, and labels the remaining LEDs according to which nodes they represent on the robot. The final output of our vision system is real world (x, y) coordinates for each of the three nodes of the robot.

Vision System Communication:



Our vision system consists of three main parts: the cameras, our computer and the robot. The cameras run USB cables through an OptiHub that then runs a cord to the computer's USB port. The OptiHub assists with both power regulation and camera syncing. Pixel data can be obtained from the cameras by using the Optitrack SDK provided online by Optitrack. One issue here is that we are running our code using python 2.7.4 and the SDK is written in C++. Neal has created a module called pyoptitrack that outputs a pyoptitrack.pyd file and can be used to extract any useful data we need from the SDK therefore making it simple to get object data from the cameras. From there I have downloaded and set up pyserial on the computer so that we can communicate via Xbee to the robot. In the vision folder you can find an example script called CommandRobot.py that sets up a serial connection to COM3 and turns magnet one of the robot on. Finally the robot's code can be changed in MPLAB and wirelessly downloaded to the robot.

Next Steps

The next steps in this portion of the lab are to change the robot's code to accept data in packets that include 1 bit for direction, 11 bits for torque, and 2 bits representing whether or not each magnet is on. This will minimize the packet sizes and therefore maximize speed. We will also need to calculate the angles q_1 and q_2 from the real world data collected by our vision system. We can then insert these state values into our controllers to calculate our next torque inputs to the system.

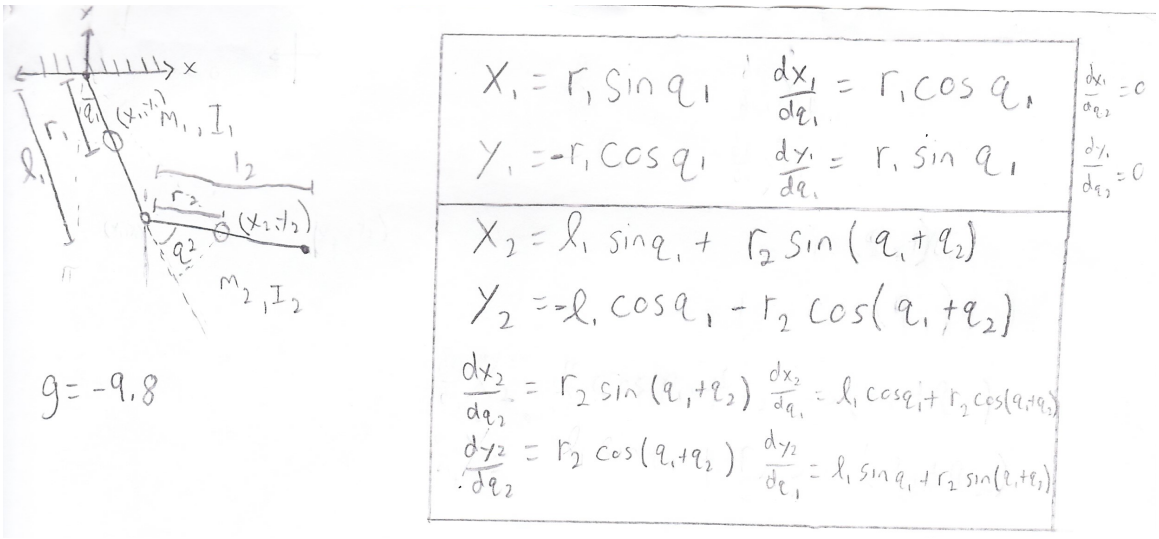
Lastly, with the new robot's encoders it may be useful to combine both the vision system's readings with the encoders for more accurate state measurements. One possible approach would be to use a Kalman filter.

Gibbot Simulation Dynamics:

The dynamics behind our robot were found by first setting up the system's kinematics, solving an Euler-Lagrange equation about the system, putting these results into standard form, and finally non-dimensionalizing these results. I have set up our robot to be represented by a system with one fixed node, 2 links, and one actuator between the robots links.

Kinematics:

The actual set-up and kinematics can be seen below:



m_1, m_2 = the mass of the links

l_1, l_2 = lengths of the links

r_1, r_2 = the lengths to COM

(x_1, y_1), (x_2, y_2) = position of COM

q_1, q_2= link angles

I_1, I_2 = moments of Inertia

Euler-Lagrange Equation

By solving Euler-Lagrange equations for this system, we are able to find our equations of motion. To compute the Lagrangian we solve for the following equation:

$$L = T - V$$

Where L is the Lagrangian, T is the systems kinetic energy and V is the systems potential energy. By substituting our system's kinematics into the definitions of kinetic energy ($\frac{1}{2}mv^2$) and potential energy do to gravity (mgy) we can begin to solve this equation.

$$T = \frac{1}{2}m_1(\dot{x_1}^2 + \dot{y_1}^2) + \frac{1}{2}I_1\dot{q_1}^2 + \frac{1}{2}m_2(\dot{x_2}^2 + \dot{y_2}^2) + \frac{1}{2}I_2(\dot{q_1} + \dot{q_2})^2$$

$$V = (m_1 g y_1) + (m_2 g y_2)$$

Lastly we find our Euler-Lagrange as follows:

$$\frac{dL}{dq_1} = \frac{d}{dt}\frac{dL}{d\dot{q_1}}$$

$$\frac{dL}{dq_2} = \frac{d}{dt}\frac{dL}{d\dot{q_2}}$$

Solving these equations about our kinematics gives us our dynamic equations. Lastly we subtracted $\frac{dL}{dq_2}$ to the other side to help us place these equations into standard form as can be seen in the next step:

$$\frac{d}{dt}\frac{dL}{d\dot{q}_1} - \frac{dL}{dq_1} = \left[m_1 r_1^2 + I_1 + m_2 l_1^2 + 2 m_2 l_1 r_2 \cos(q_2) + m_2 r_2^2 + I_2\right]\ddot{q}_1$$
$$+ \left[m_2 l_1 r_2 \cos(q_2) + m_2 r_2^2 + I_2\right]\ddot{q}_2$$
$$+ \left[-2 m_2 l_1 r_2 \sin(q_2)\right]\dot{q}_1\dot{q}_2$$
$$+ \left[-m_2 l_1 r_2 \sin(q_2)\right]\dot{q}_2^2$$
$$+ \left[m_1 r_1 \sin(q_1) + m_2 l_1 \sin(q_1) + m_2 r_2 \sin(q_1+q_2)\right]g$$

$$\frac{d}{dt}\frac{dL}{d\dot{q}_2} - \frac{dL}{dq_2} = \left[m_2 l_1 r_2 \cos(q_2) + m_2 r_2^2 + I_2\right]\ddot{q}_1 + \left[m_2 r_2^2 + I_2\right]\ddot{q}_2 + \left[m_2 l_1 r_2 \sin(q_2)\right]\dot{q}_1^2 + \left[m_2 r_2 \sin(q_1 + q_2)\right] g$$

Standard Form:

After solving for our Euler-Lagrange equation I put our equations into standard form. The reason for this is that once in standard form we can solve the equations for $\ddot{q}$ and utilize these equations to output the robot's state given the torque we input into the robots motor.

$$M(q)\ddot{q} = C(q)\dot{q} + g(q) - u$$
$$\ddot{q} = M^{-1}(q)(C(q)\dot{q} + g(q) - u)$$

M(q) = Mass Matrix
$C(q)\dot{q}$ = Coriollis Vector (describes the force in the rotating frame of reference)
g(q) = Gravity array
u = Torque

The matrices and arrays below were found by separating the different components of the Euler-Lagrange equations above.

$$M(q) = \begin{bmatrix} m_1 r_1^2 + I_1 + m_2 l_1^2 + 2 m_2 l_1 r_2 \cos(q_2) + m_2 r_2^2 + I_2 & m_2 l_1 r_2 \cos(q_2) + m_2 r_2^2 + I_2 \\ m_2 l_1 r_2 \cos(q_2) + m_2 r_2^2 + I_2 & m_2 r_2^2 + I_2 \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix}$$

$$C(q) = \begin{bmatrix} -m_2 l_1 r_2 \sin(q_2)\dot{q}_2 & -m_2 l_1 r_2 \sin(q_2)(\dot{q}_1 + \dot{q}_2) \\ m_2 l_1 r_2 \sin(q_2)\dot{q}_1 & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}$$

$$g(q) = g \begin{bmatrix} m_1 r_1 \sin(q_1) + m_2 l_1 \sin(q_1) + m_2 r_2 \sin(q_1 + q_2) \\ m_2 r_2 \sin(q_1 + q_2) \end{bmatrix}$$

Non-Dimensionalization

Lastly I non-dimensionalized the dynamics of our system so that they would be easier to work with. To do this I simply divided or multiplied the equations by the appropriate parameters until they no longer had units. My results can be seen below:

$$\beta = \frac{m_1 r_1 + m_2 l_1}{m_2 r_2} \qquad K_1 = \frac{1}{d_x}\left(\frac{I_2}{m_2 r_2} + r_2\right)$$

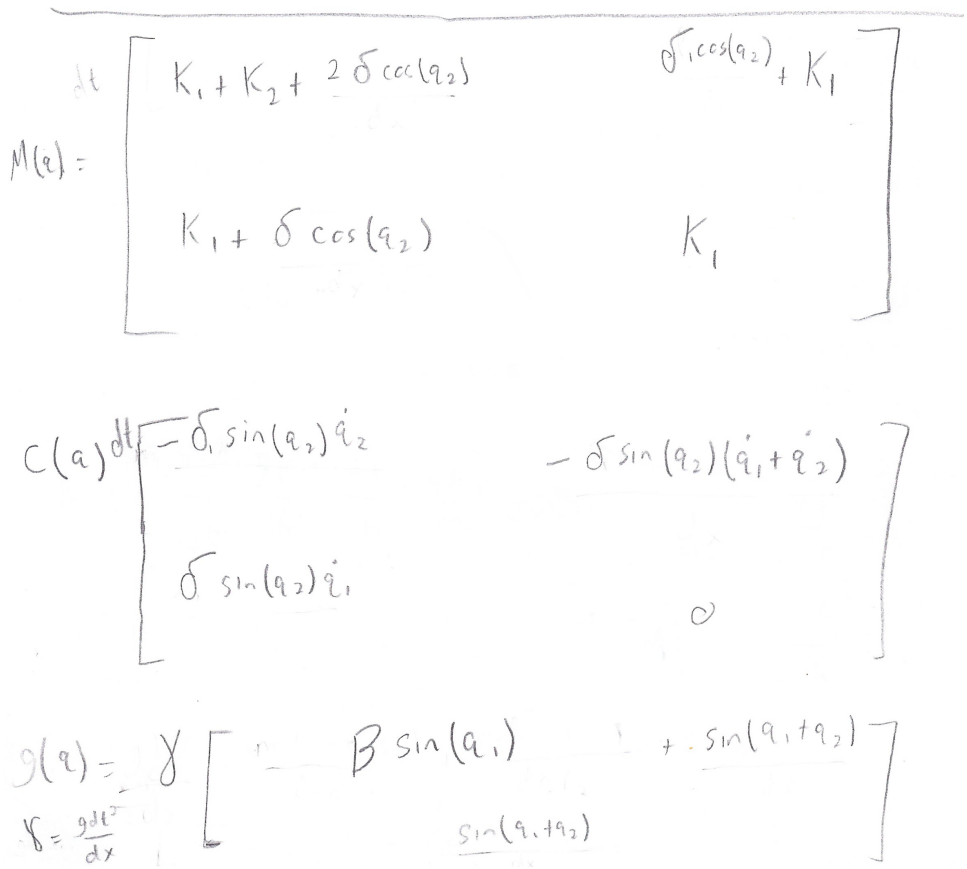
$$\gamma = g\, d_t^2 / d_x = 1 \qquad K_2 = \frac{1}{d_x}\left(\frac{I_1 + m_1 r_1^2}{m_2 r_2} + \frac{l_1^2}{r_2}\right)$$

$$\delta = \frac{l_1}{d_x}$$

Scaling

$$d_x = l_1 + l_2 \qquad d_v = g m_2 r_2$$

$$d_t = \sqrt{\frac{l_1 + l_2}{g}}$$



Nelson solved the calculations above separately and we both ended with the same results giving me confidence in the values above. These matrices are now used in the GibbotModel.py code that is called by the SimulatorApp.py script. When the SimulatorApp.py script is run you can see a simulation of the system using the dynamics solved for here and can call different controllers/change the parameters of the robot to get a sense for how they might affect the performance of the robot.

Gibbot Balancing Controller:

Overview

The balancing controller is only effective when the dynamics of the robot can be linearized about the vertical axis. This only occurs when the robot is very close to the upright orientation. The linear system takes the form:

$$\dot{x} = Ax + Bu$$

where,

$$F = \begin{bmatrix} \dot{q} \\ M^{-1}(q) * (u(t) - c(q) - g(q)) \end{bmatrix}$$

$$A = \frac{dF}{dx(t)} = \begin{bmatrix} \frac{dF}{dq_1} & \frac{dF}{dq_2} & \frac{dF}{d\dot{q}_1} & \frac{dF}{d\dot{q}_2} \end{bmatrix}$$

$$B = \frac{dF}{du(t)} = M^{-1} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

In these equations $\dot{x}$ represents our change in state, F represents our dynamics, and the A matrix/B array are used to solve for the K matrix below which maps state outcomes with torque inputs. In order to solve for the K matrix we must solve a linear quadratic regulator about matrix A, array B, and two weighting matrices Q and R. The Q term can be used to weight the importance of our trajectory to our final goal while the R term can be used to influence the emphasis we would like to place on the control effort of the system.

$$u(t) = -K * x(t)$$

$$K = lqr(A, B, Q, R)$$

The linear quadratic regulator is used to minimize the cost that maps our torques to our states. Since we input our goal state as being in the upright position and only use this controller when we can make an accurate linear approximation

about that goal we can solve for this K matrix (Our state feedback controller). The K matrix only needs to be solved for once and will never change given that the parameters of the robot remain unchanged.

Results:

The simulation I ran used the parameters of the old Gibbot model:

$$m_1 = m_2 = 1.0\ (Kg)$$

$$l_1 = l_2 = 0.267\ (m)$$

$$r_1 = .107\ (m) \qquad r_2 = 0.156\ (m)$$

$$I_1 = m_1 * \left(\frac{l_1^2}{3} + \frac{0.077^2}{12}\right) = .0243 \quad I_2 = m_2 * \left(\frac{l_2^2}{3} + \frac{0.077^2}{12}\right) = .0243$$

$$g = 9.81\ (\frac{m}{s^2})$$

These parameters result in the values for A and B below and I chose to use the identity matrix for Q and a value of 1 for R in order to weight everything evenly.

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1.7997808 & -1.00157211 & 0 & 0 \\ -1.6281611 & 3.57439674 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ 0 \\ -2.17005086 \\ 5.7444476 \end{bmatrix}$$

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R = 1$$

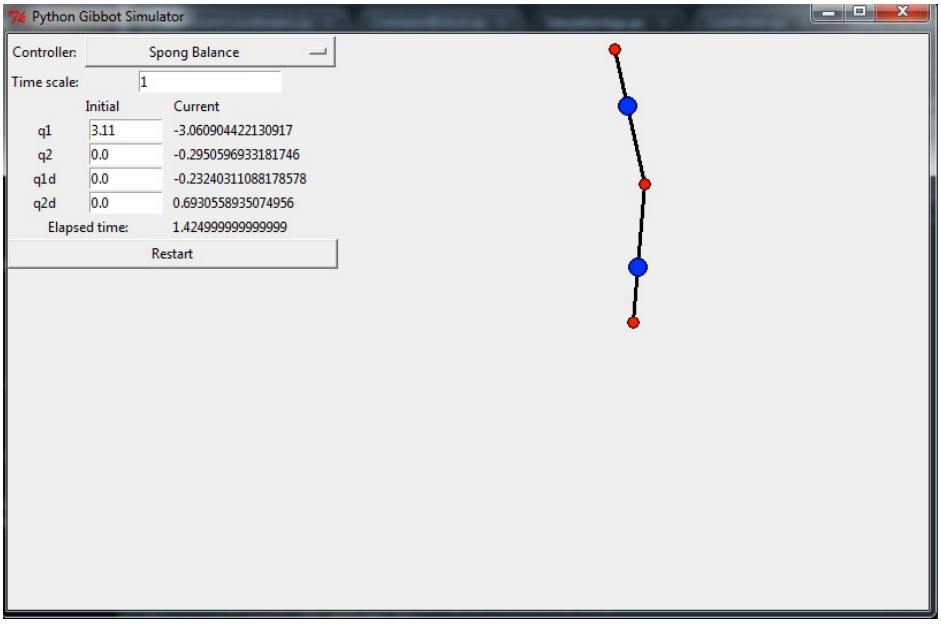
On the next pages are two pictures showing the actual simulation along with a graph plotting the error of angles q1 and q2 with respect to the upright vertical position.

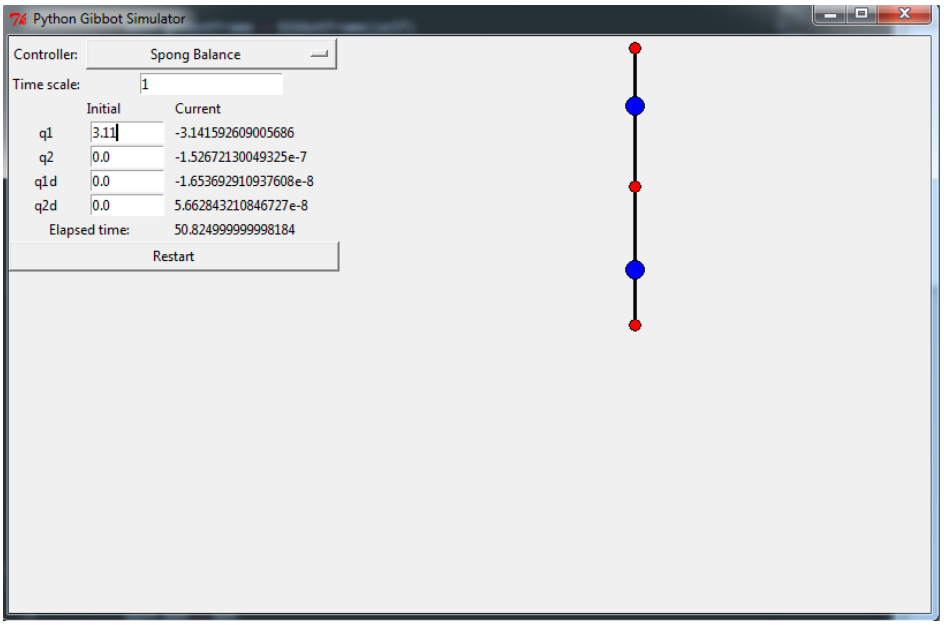
Initial Conditions (radians):

$q_1 = 3.11$ $\qquad$ $q_2 = 0$

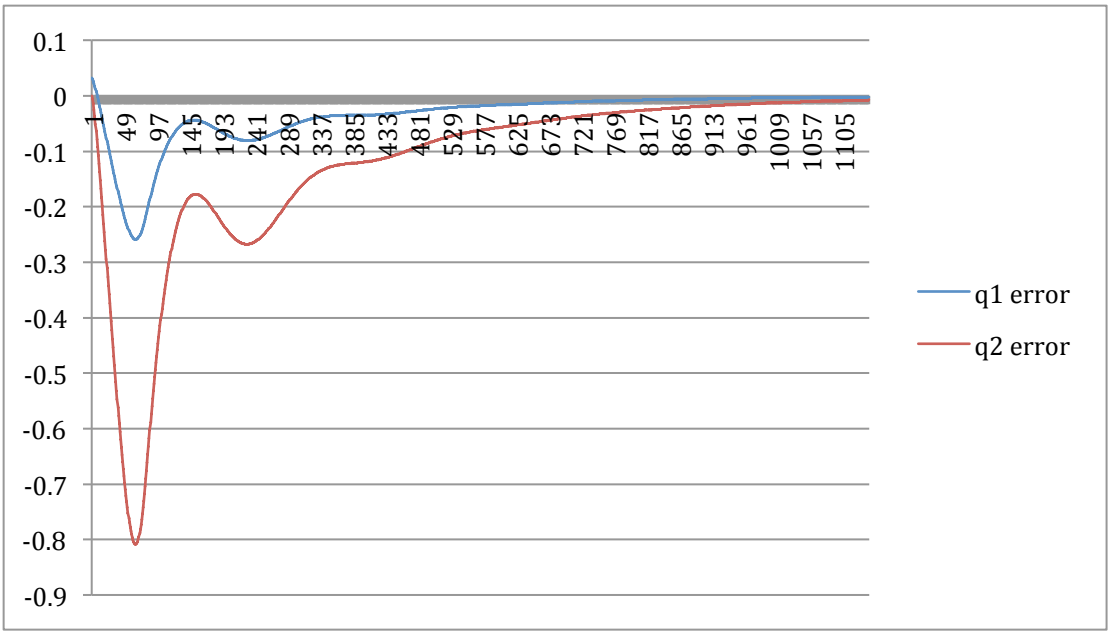
$\dot{q_1} = 0$ $\qquad$ $\dot{q_2} = 0$

$e_1 = 0.0315926535898\ (\approx 5.69°)$ $\qquad$ $e_0 = 0$





***NOTE** The user interface is quite simple. Initial conditions for q_1, q_2, $\dot{q_1}$, and $\dot{q_2}$can be typed in the GUI and the controller desired can be selected through the drop down menu. The restart button will restart the simulation with your inputs.*



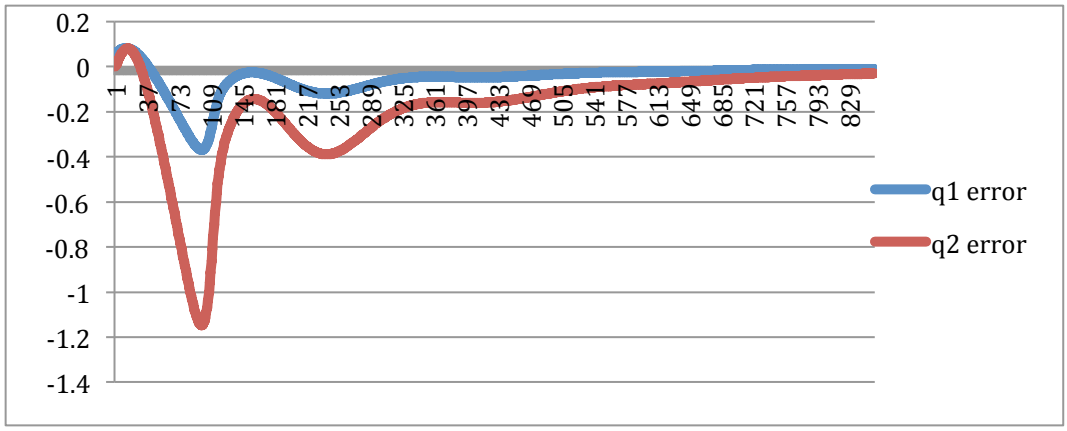
As can be seen in this graph q_1 and q_2 converge towards 0 after about 1000 time steps. Because our simulation is unit less the x-axis should not be thought of as seconds but rather as time steps. I also found that by initializing our simulator with a velocity about q1 the controller was able to stabilize at errors that doubled that of above:

Initial Conditions (radians):

$q_1 = 3.08$ $q_2 = 0$

$\dot{q_1} = .1$ $\dot{q_2} = 0$

$e_1 = 0.0615926535898 \ (\approx 11.37°)$ $e_0 = 0$



Finally, I attempted to experiment with the weighting value R while solving for the K matrix. By either decreasing this number I found that the resulting controller began calling for very large torque values that went beyond the max power of our old Gibbot robot. Currently we have our old Gibbot robot sending a maximum torque of 5 so anytime the controller asks for more than this value it is actually only being sent a value of 5 and the controller becomes very limited. The max torque parameter can be set in the GibbotModel.py and will need to be changed when running simulations on the new Gibbot.

Next Steps:

I have created a script called SolveForKMatrix.py that when given a set of parameters for the robot will spit out a K matrix. It uses both the scipy and numpy libraries for Python 2.7. I have downloaded both of these onto the computer in the NxR lab. I found that the controller generated was only satisfied when the robot was extremely close to the upright position. In order to widen the window of success the controller has someone will have to play with the Q and R gain matrices in this script. This will place different weights on different components of either the A or B matrices and spit out new K matrices. The resulting K matrix can then be placed in the spongBalanceController function within Controllers.py. To test the new matrix try running the controller in the simulator. It is also possible that the parameters of the new robot will spit out a new and more effective K matrix and you may also find that with different initial velocities the controller may perform better.

Old Gibbot Debugging:

While attempting to test our controllers with the old Gibbot robot I ran into many hardware issues. The first issue was that we found the DC/DC converter on the high power board of the robot had fried and was heating to extremely high temperatures. We have since replaced the part but still find that it gets very hot at times and eventually we turn off the robot. When re-attaching the lid of robot I once experienced that by pushing down on the components something shorted with the motor and powered it on even though the power switch was off. I believe the short had to do with the H-bridge because the component is very exposed and was beneath the spot I was putting pressure. I suggest this component is cased or at the very least covered with electrical tape before the robot is put back in use. Lastly we still have not tracked down this issue but when the robot is hanging on to the wall and is issued a torque command the motor shakes back and forth as if it has been sent an oscillating signal. I do not know if this is due to a mechanical fault and slippage in the motor or if a bad signal is being sent. We have attempted to send commands to the robot while it is lying flat and it moves its limb correctly but very weak and slowly. I believe the motor is burning out and needs replaced. Nelson helped me debug this system and should be able to help answer any questions regarding the robot's current state.