EECS 206B Lab 3 Report: Nonholonmic control with Turtlebots

Aakash Parikh, Federik Warburg, Jun Zeng

April 2019

1 Introduction

In this project, we develop controllers for a second order nonholonomic system using two methods. We first try to control the system directly on its Lie algebra with a Bang-Bang controller. Then we utilize the steering with sinusoids method [1] to do more efficient open and closed loop control. We use these controllers to move a Turtle Bot that is constrained to the bicycle model.

A video of Turtlebot being controlled can be found on YouTube. We also include our GitHub repository here.

2 Approach and Methods

2.1 Bang-Bang Control

The first controller we implemented was the bang bang controller. This involves alternating between maximum and minimum controller input. Using the lie algebra of the system to guide our intuition, we designed basic controllers for the various maneuvers.

2.1.1 Translation in x

This control was fairly straightforward since we have direct control of this motion:

Result: Translate X
goal := [x, 0, 0, 0];
while *goal[0] - state[0]* $\geq \varepsilon$ **do**
    command(x, 0);
    sleep(1);
end
command(0,0);
sleep(1);

2.1.2 Translation in y

Here we utilized a strafing motion to generate translation in y.

Result: Translate Y
goal := [0, y, 0, 0];
command(1,1);
while *goal[1] - state[1]* $\geq \varepsilon$ **do**
    command(0.5, -1);
    sleep(1);
    command(-0.5, 1);
    sleep(1);
    command(-0.5, -1); sleep(1);
    command(0.5, 1);
    sleep(1);
    command(0,0);
end
command(0,0);
sleep(1);

2.1.3 Yaw rotation

This involves repeated back and forth turning rotation, and the drift results in a rotation.

Result: Yaw Rotation
goal := [0, 0, angle, 0];
while *goal[2] - state[2]* $\geq \varepsilon$ **do**
    command(0.3, 1);
    sleep(1);
    command(-0.3, -1);
    sleep(1)
end
command(0,0);
sleep(1);

2.1.4 S Turn

We built this function as a composition of the previous maneuvers.

Result: S Turn
goal := [0.5, 0.5, 0, 0];
translateX (0.25) ;
rotate($\pi/2$) ;
translateX (0.5) ;
rotate($-\pi/2$) ;
translateX (0.25) ;
command(0,0) ;

2.2 Sinusoidal Control

Our implementation for this controller was very straightforward based on Sastry and Murray's paper [1] and the math given in the lab document.

The approximated car model we use is as shown:

$$\begin{aligned} \dot{x} &= v_1 & v_1 &= \cos(\theta)u_1 \\ \dot{\phi} &= v_2 & v_2 &= u_2 \\ \dot{\alpha} &= \frac{1}{l}\tan(\phi)v_1 & \alpha &= \sin(\theta) \\ \dot{y} &= \frac{\alpha}{\sqrt{1-\alpha^2}}v_1 \end{aligned}$$

2.2.1 Steer x

This method was given and is simple since we have direct control over x.

Result: Steer x
Δ_x := goal[0] - start[0];
v1 = Δ_x / Δ_v;
v2 = 0;
path := [];
$t = t_0$;
while $t < t_0 + \Delta_t$ **do**
    path.append($[t, v1, v2]$);
    $t = t + dt$;
end
return vPathToUPath(path, start, dt);

2.2.2 Steer ϕ

This method is done similarly to steer x.

Result: Steer ϕ
Δ_ϕ := goal[1] - start[1];
v1 = 0;
v2 = Δ_ϕ / Δ_t;
path := [];
$t = t_0$;
while $t < t_0 + \Delta_t$ **do**
    path.append($[t, v1, v2]$);
    $t = t + dt$;
end
return vPathToUPath(path, start, dt) ;

2.2.3 Steer α

This method is more complicated as α cannot be controlled by g_1 or g_2, but instead needs to be controlled by g_3. Here we find a_1 and a_2 such that the inputs $v_1 = a_1 sin(\omega t)$ and $v_2 = a_2 cos(\omega t)$ change α without modifying ϕ or x

Result: Steer α
$\Delta_\alpha := goal[2] - start[2]$;
$\omega = 2\pi/\Delta_t$;
$a_2 = \min(1, \Delta_\phi \omega)$;
$f = lambda(phi) : tan(\phi)/l$;
$\phi_{fn} = lambda(t) : a_2 sin(\omega t)/\omega + start[1]$;
$integrand = lambda(t) : f(\phi_{fn}(t)) sin(\omega t)$;
$\beta_1 = \omega quad(integand, 0, \Delta_t)[0]/\pi$;
$a_1 = \Delta_\alpha \omega/(\pi\beta_1)$;
$v1 = lambda(t) : a_1 sin(\omega t)$;
$v2 = lambda(t) : a_2 cos(\omega t)$;
path := [];
$t = t_0$;
while $t < t_0 + \Delta_t$ **do**
    path.append($[t, v1 \cdot (t - t_0), v2 \cdot (t - t_0)]$);
    $t = t + dt$;
end
return vPathToUPath(path, start, dt);

2.2.4 Steer y

This method is more complicated as y cannot be directly controlled by g_1, g_2 or even g_3, but by g_4. Here we find a_1 and a_2 such that the inputs $v_1 = a_1 sin(\omega t)$ and $v_2 = a_2 cos(2\omega t)$ change y without modifying ϕ, x, or α. We utilize a binary search on α_1 after defining α_2 in order to do this.

Result: Steer y
$\Delta_y := goal[3] - start[3]$;
$\omega = 2\pi/\Delta_t$;
$a_2 = min(2, \Delta_\phi \omega)$;
$a_{1min} = 0.0$;
$a_{1max} = 5.0$;
$error_{tol} = 0.01$;
while $\|error\| > error_{tol}\Delta_y$ **do**
    $a_{1mid} = (a_{1min} + a_{1max})/2$;
    $f = lambda(\phi) : tan(\phi)/l$;
    $g = lambda(\alpha) : \alpha/sqrt(1-\alpha^2)$;
    $\phi_{fn} = lambda(t) : a_2 sin(2\omega t)/(2\omega)$;
    $integrand_1 = lambda(t) : f(\phi_{fn}(t))a_1 sin(\omega t)$;
    $integrand_2 = lambda(t) : g(quad(integrand_1, 0, t)[0])sin(\omega t)$;
    $\beta_1 = (\omega/\pi)quad(integrand_2, 0, 2\pi/\omega)[0]$;
    if $error \geq 0$ **then**
        $a_{1min} = a_{1mid}$;
    end
    else
        $a_{1max} = a_{1mid}$;
        break ;
    end
end
$v1 = lambda(t) : a_1 sin(\omega t)$;
$v2 = lambda(t) : a_2 cos(2\omega t)$;
path := []
$t = t_0$
while $t < t_0 + \Delta_t$ **do**
    path.append($[t, v1 \cdot (t - t_0), v2 \cdot (t - t_0)]$);
    $t = t + dt$;
end
return vPathToUPath(path, start, dt);

2.3 Feedback Controller

Paths based on pre-computed plans with an open loop controller often doesn't work well in practice. Thus we add a Feedback controller, where we can compute gains in advance.

To design the controller, we follow the lab's instructions and calculate the Jacobians analytically A and B as below,

$$A = \begin{bmatrix} 0 & 0 & -\sin(\theta)u_1 & 0 \\ 0 & 0 & \cos\theta u_1 & 0 \\ 0 & 0 & 0 & \frac{1}{l}\sec(\phi)^2 u_1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ \frac{1}{l}\tan\phi & 0 \\ 0 & 1 \end{bmatrix}. \tag{1}$$

However, this feedback controller cannot work in practice due to singularity concerns. The singularity comes from the integral to get the values of H_c.

$$H_c(t) = \int_{t_0}^{t_1} e^{6\alpha(\tau - t)} e^{-A\tau} B(\tau) B(\tau)^T e^{-A\tau^T} d\tau. \tag{2}$$

Let's analyze the integral above firstly. Based on the definition of matrix exponential, $e^{-A\tau}$ and its transpose wouldn't be singular. However, the $B(\tau)B(\tau)^T$ could be very singular along the trajectory. Firstly and this comes from two reasons. Firstly, the value of θ is zero initially which reduces the rank. Moreover, the steering angle ϕ is very bounded and remains a very small value during quite a while. We try to solve this problem with increasing horizon number for the integral:

$$H_c(t) \approx \sum_{i=0}^{N} e^{6\alpha(t_i - t)} e^{-At_i} B(t_i) B(t_i)^T e^{-At_i{}^T} \Delta t \tag{3}$$

where we hope the accumulating the time-variant BB^T can help the system become full rank.

However these methods still doesn't solve the singularity problem. As I recall the examples shown in the Sastry's paper, I really feel we should have some less singular nominal trajectory. The parallel parking should be a less singular trajectory but it still doesn't work very well. I tried to add some noise on the $B(t_i)$, this can solve the singularity but it cannot hold a good control performance as the control on the steering angle is very delicate. Rescaling method that I have talked with Valmik works a little bit but the method will fail after some iterations, which is really out of my expectations. I attach my code and hope to hear some feedback about this question (rescaling method is excluded due to the bad performance).

```
def calculateGain(self, plan):
    gains = []
    #  x0(t) state (state.x, state.y, state.theta, state.phi)
    #  u0(t) cmd  (cmd.linear_velocity, cmd.steering_rate)

    # constant
    alpha = 0.1
    delta = 0.01
    gamma = 1

    cmd_list = []
    state_list = []
    for (t, cmd, state) in plan:
        cmd_list.append([cmd.linear_velocity, cmd.steering_rate])
        state_list.append([state.x, state.y, state.theta, state.phi])

    for (t, cmd, state) in plan:

        time_horizon = 100
        H = np.zeros((4,4))
        t_start = t
        for index in range(time_horizon):
            index_ss = (int)(index+ t/delta)
            A = np.matrix([[0, 0, -np.sin(state_list[index_ss][2])*cmd_list[index_ss][0],0 ],
            [0, 0, np.cos(state_list[index_ss][2])*cmd_list[index_ss][0], 0],
            [0, 0, 0, 1/l *( 1 + np.tan(state_list[index_ss][2])**2)*cmd_list[index_ss][0]],
            [0, 0, 0, 0]])

            B = np.matrix([[np.cos(state_list[index_ss][2]),0],
            [np.sin(state_list[index_ss][2]),0],
            [1/l*np.tan(state_list[index_ss][2]),0],
            [0,1]])

            func1 = lambda tau : np.matmul(np.transpose(B),np.transpose(expm(-A*tau)))
            func2 = lambda tau : np.matmul(B,func1(tau))
            func3 = lambda tau : np.matmul(expm(-A*tau),func2(tau))
            func4 = lambda tau : np.exp(6*alpha*(tau-t_start) * func3(tau))

            size = 20 # simpson method
            for ind in range(size):
                integral_delta_t = delta/size
                temp = func4(t_start + ind*integral_delta_t)
                for i in range(4):
                    for j in range(4):
                        H[i,j] += temp[i,j] * integral_delta_t
            time.sleep(0.5)
            print(H)
        P = np.linalg.inv(H)

        gain = gamma * np.matmul(np.transpose(B),P)

        gains.append(gain)

    return gains
```

Listing 1: Feedback controller

3 Discussion of Results

3.1 Bang-Bang Control

Overall, the Bang Bang Control produced very rough results. For the simple case like translate in x and for rotation, it worked fine, but we had substantial drift in the other cases. The graphs for these cases can be seen below.

1. **Translation in x [1,0,0,0]** can be seen in 1, 2
2. **Translation in y [0,0.5,0,0]** can be seen in 3, 4
3. **Translation in y [0,1,0,0]** can be seen in 5, 6
4. **Yaw rotation (u turn) [0,0,π,0]** can be seen in 7, 8
5. **Translating in x and y (s turn) [0.5; 0.5; 0; 0]** can be seen in 9, 10

3.2 Sinusoidal Control

This controller produced much cleaner results in simulation. However, when we tried our controller on the turtlebots, we had many difficulties reproducing the similar quality results as we found in simulation. As seen on the plots, in simulation we notice very little drift in any of our maneuvers. However, this drift is noticeable when executed on the turtlebots. However, this is expected due to variances in physical systems. More problematic is that the turtlebots did not execute the desired trajectories. We expect this to be connect the rospy.Rate as the controller seemed very dependt on a good choice of rate. Also, it might be connect to the choice of a_2. We expect that fine tuning of these parameters would help generate nicer results.

One key issue that we had to resolve was a singularity in our rotation at $\pi/2$. We do this by breaking up our rotation in θ into several sub paths for cases when $\theta > 1$. In particular, we attempt to first rotate by $\theta/3$. Then we reset our current state to 0 and attempt to rate by another $\theta/3$. This is repeated one last time to get a net rotation of θ. This works well for us, but we notice that the error accumulates.

1. **Translation in x [1,0,0,0]** can be seen in 11, 12, 13
2. **Translation in y [0,0.5,0,0]** can be seen in 14, 15, 16
3. **Translation in y [0,1,0,0]** can be seen in 17, 18, 19
4. **Yaw rotation (u turn) [0,0,π,0]** can be seen in 20, 21, 22
5. **Translating in x and y (s turn) [0.5; 0.5; 0; 0]** can be seen in 23, 24, 25

4 Summarization of two papers

4.1 A dynamic programming approach for nonholonomic vehicle maneuvering in tight environments [2]

In this paper [2], Schildbach and Borrelli tackles the problem of generating obstacle-avoiding trajectories for vehicles in real-time in tight or cluttered environments. They propose an algorithm that performs tree-search on a discrete state space with dynamic programming. They show in simulation and with several experiments that even complicated paths can be computed efficiently. The algorithm works in the following steps: First an infinite feasible configuration space is discretized. Then, a search tree is constructed backwards from the set of target points, which results in candidate path segments (lines or arcs). The segments becomes a path if the angular orientation at the arc falls within a tolerance and the car does not collide with an obstacle.

The Steering with Sinusoid paper have contributed to this paper as they try to solve similar problems. However, Schildbach et al. points out that the Steering with Sinusiod controller cannot keep the trajectory out of obstacles placed in the configuration space.

4.2 Steering on SO(3) with Sinusoidal Inputs [3]

In this paper [3], Wang et al. present steering control strategies for kinematic systems where the input is restricted to a class of sinusoidal functions. They present an exact analysis using a closed-form solution of the kinematic equations and establish a controllable kinematic system with sinusoidal input restrictions.

Compared to Murray et Sastry's paper [1], Wang et al. constraint the input, but do not have the nonholonomic constraint. Thus, they only draw on the sinusoidal controller and not the nonholonomic model.

5 Conclusion

5.1 Lab Documentation (Bonus)

References

[1] R. M. Murray and S. S. Sastry. Nonholonomic motion planning: steering using sinusoids. *IEEE Transactions on Automatic Control*, 38(5):700–716, May 1993.

[2] G. Schildbach and F. Borrelli. A dynamic programming approach for nonholonomic vehicle maneuvering in tight environments. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 151–156, June 2016.

[3] S. Wang, J. B. Hoagg, and T. M. Seigler. Steering on so(3) with sinusoidal inputs. In *2017 American Control Conference (ACC)*, pages 604–609, May 2017.

6 Appendix

6.1 Bang Bang Controller Graphs

Note that the graphs with 4 subplots are $[x, y, \theta, \phi]$ vs time. Where the red line is desired and the green is the actual state. The graphs with 4 subplots are $[x, y]$ plots, where the red line is desired and the blue line is the actual state.

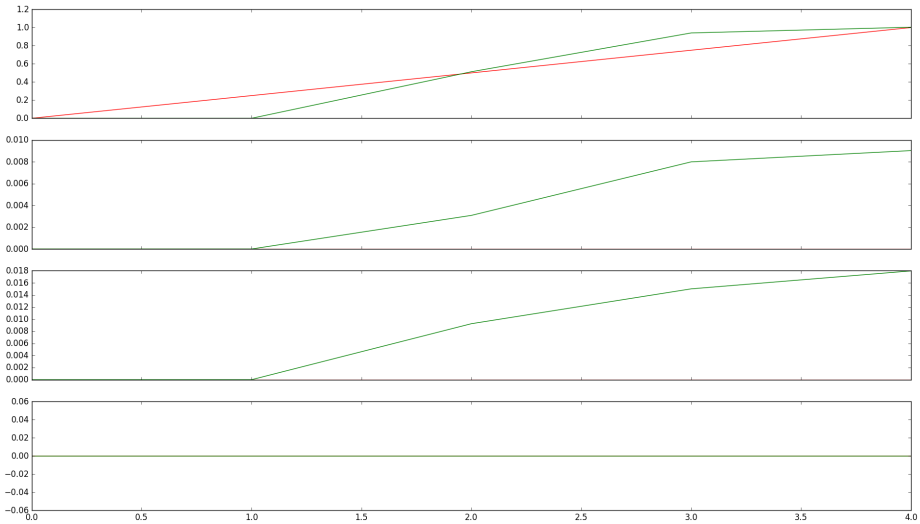

Figure 1: BangBang Controller translateX(1) Desired vs Real for each state over time over time

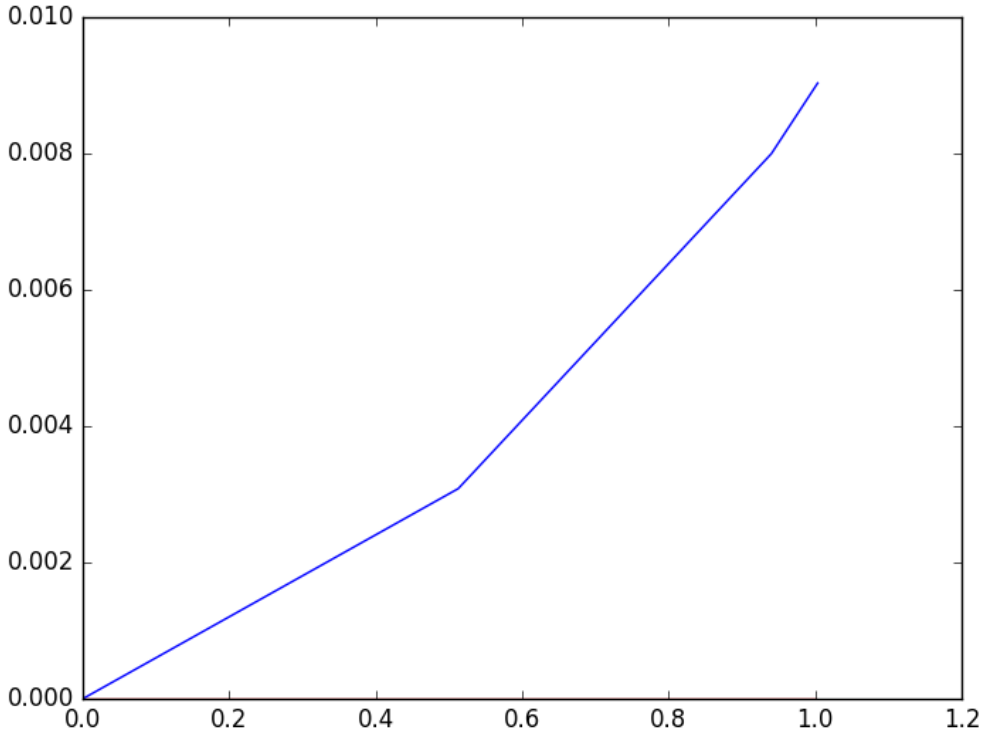

Figure 2: BangBang Controller translateX(1) Desired vs Real position

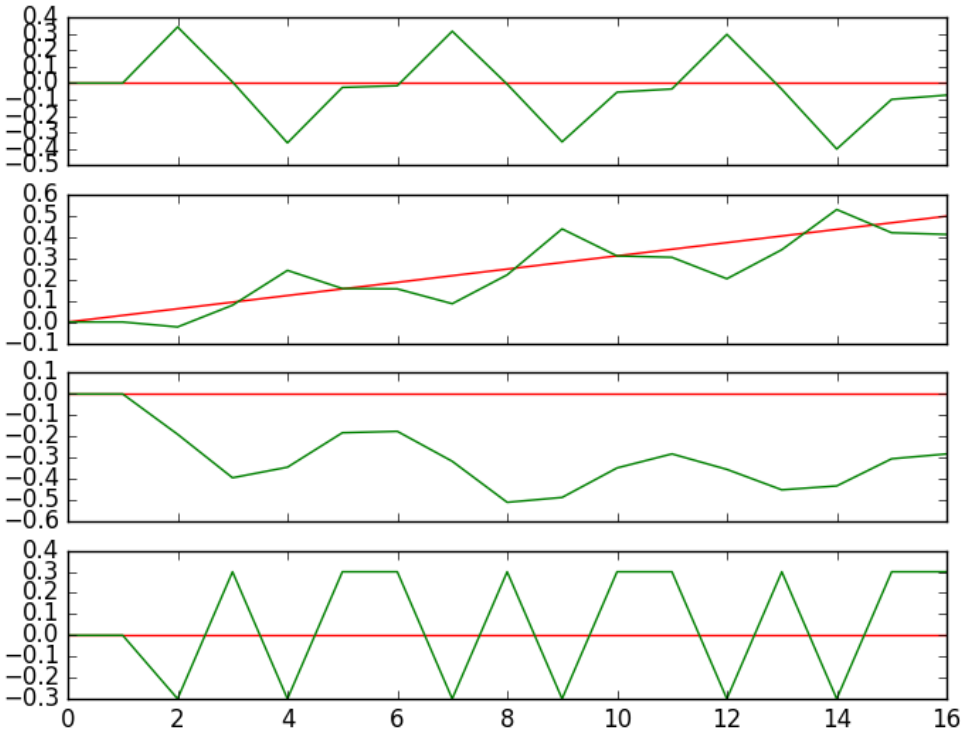

Figure 3: BangBang Controller translateY(0.5) Desired vs Real for each state over time over time

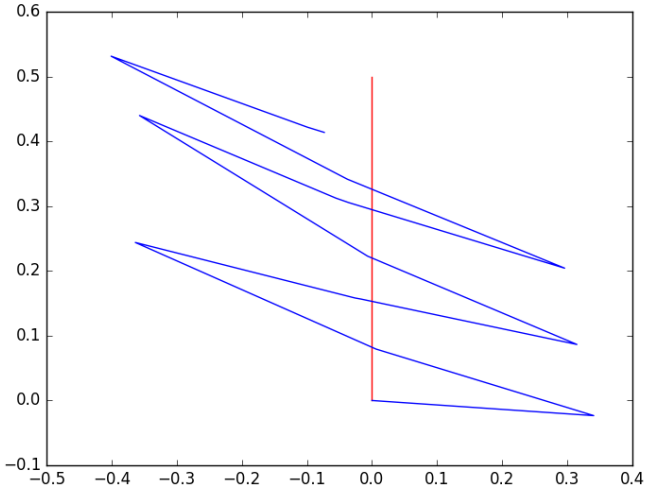

Figure 4: BangBang Controller translateY(0.5) Desired vs Real position

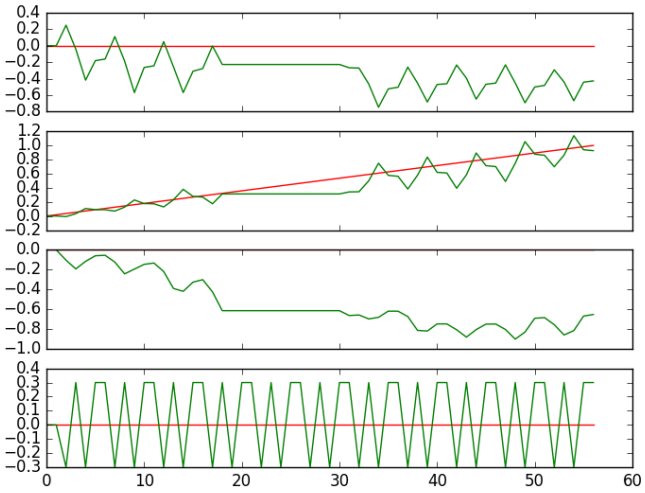

Figure 5: BangBang Controller translateY(1) Desired vs Real for each state over time over time

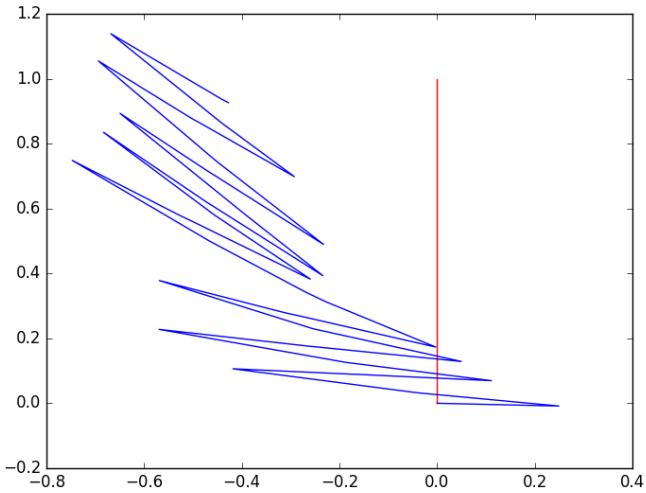

Figure 6: BangBang Controller translateY(1) Desired vs Real position

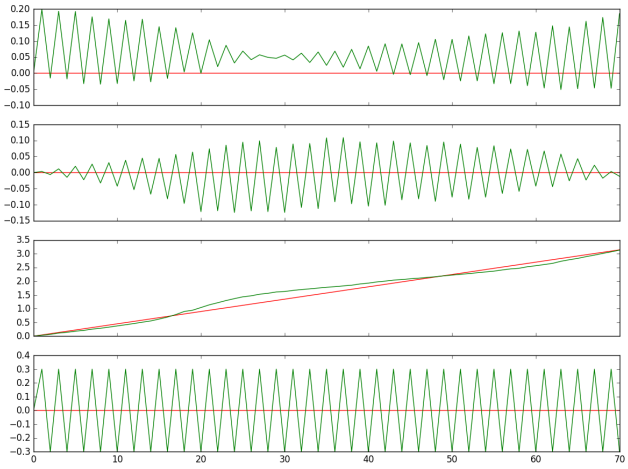

Figure 7: BangBang Controller rotateYaw(π) Desired vs Real for each state over time over time

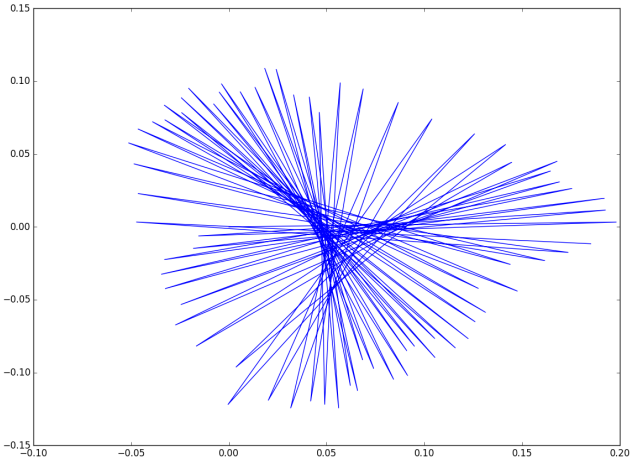

Figure 8: BangBang Controller rotateYaw(π) Desired vs Real position

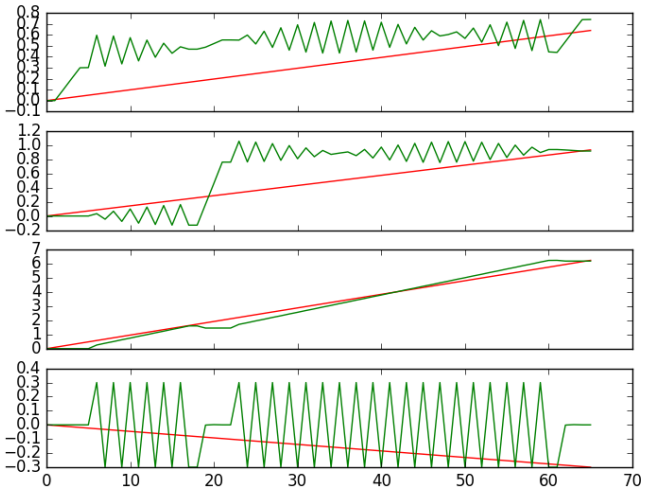

Figure 9: BangBang Controller sTurn() Desired vs Real for each state over time over time

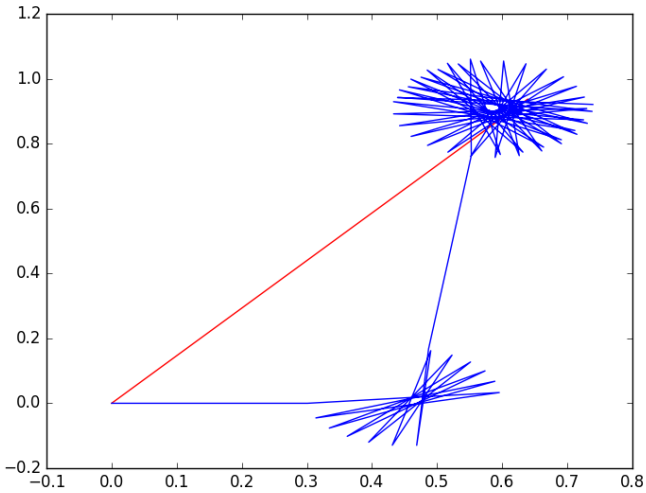

Figure 10: BangBang Controller sTurn() Desired vs Real position

6.2 Sinusoid Controller Graphs

Note that the graphs with 4 subplots are $[x, y, \theta, \phi]$ vs time. Where the red line is desired and the green is the actual state. The graphs with 4 subplots are $[x, y]$ plots, where the red line is desired and the blue line is the actual state.

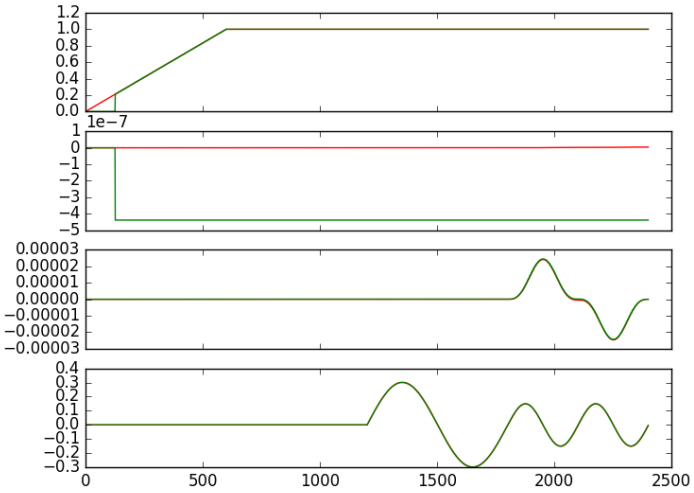

Figure 11: In Simulation Sinusoid Controller translateX(1) Desired vs Real position

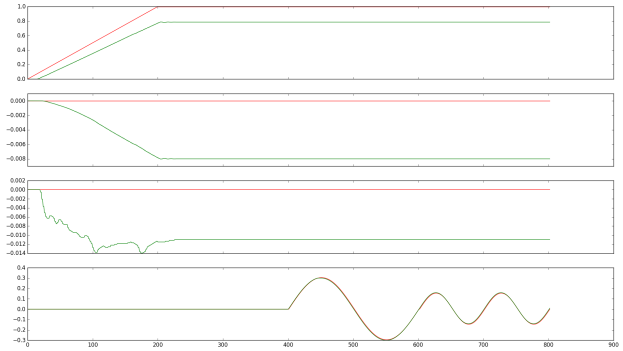

Figure 12: Sinusoid Controller translateX(1) Desired vs Real for each state over time over time

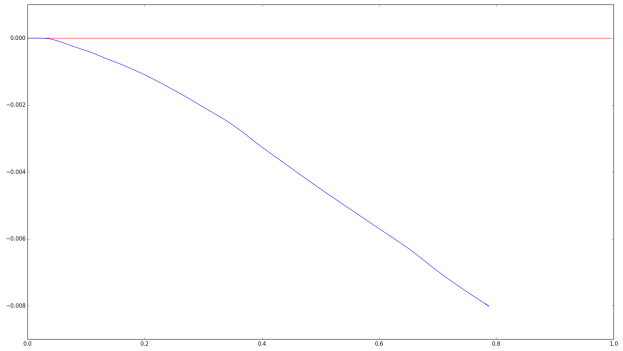

Figure 13: Sinusoid Controller translateX(1) Desired vs Real position

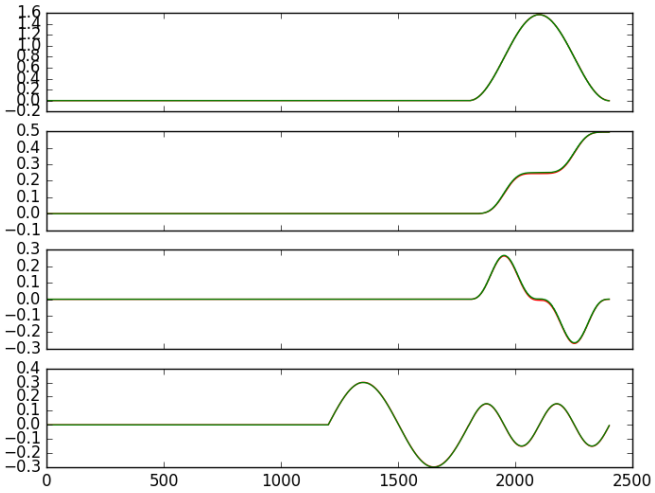

Figure 14: In Simulation Sinusoid Controller translateY(0.5) Desired vs Real position

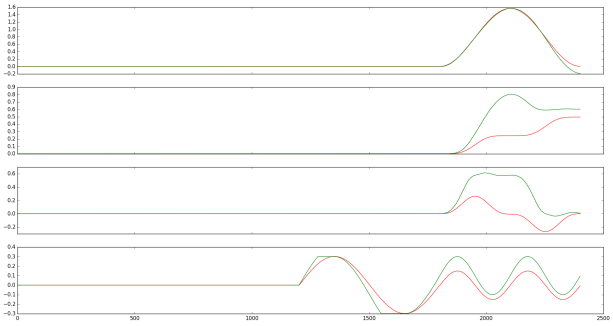

Figure 15: Sinusoid Controller translateY(0.5) Desired vs Real for each state over time over time

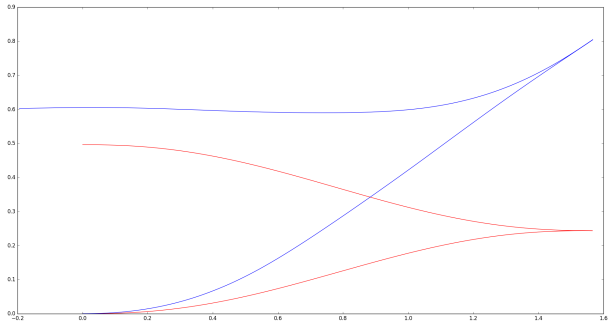

Figure 16: Sinusoid Controller translateY(0.5) Desired vs Real position

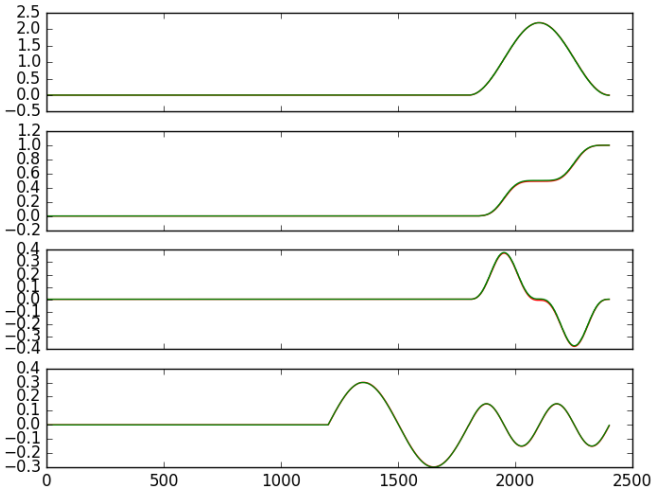

Figure 17: In Simulation Sinusoid Controller translateY(1) Desired vs Real position

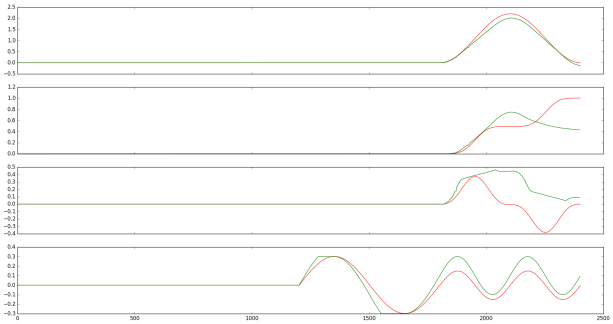

Figure 18: Sinusoid Controller translateY(1) Desired vs Real for each state over time over time

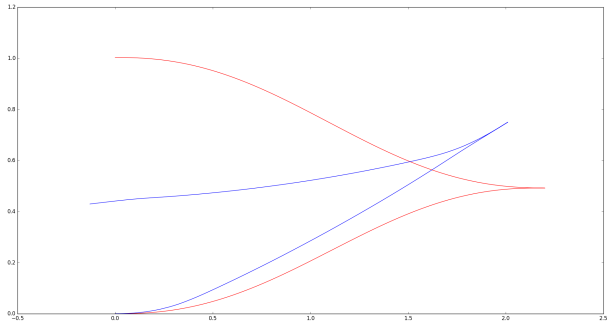

Figure 19: Sinusoid Controller translateY(1) Desired vs Real position

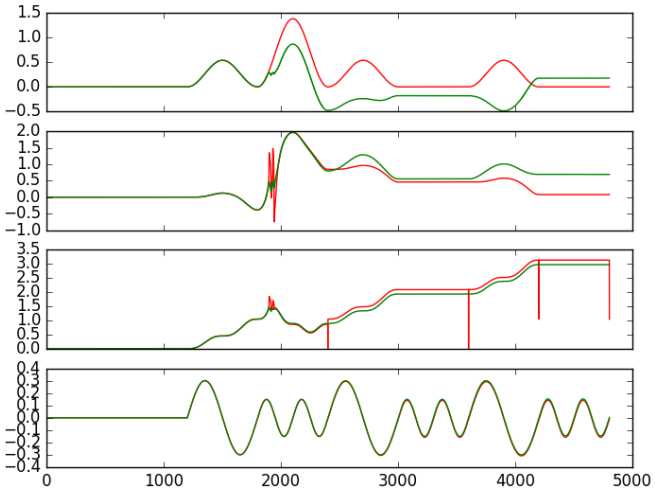

Figure 20: In Simulation Sinusoid Controller rotateYaw(π) Desired vs Real position

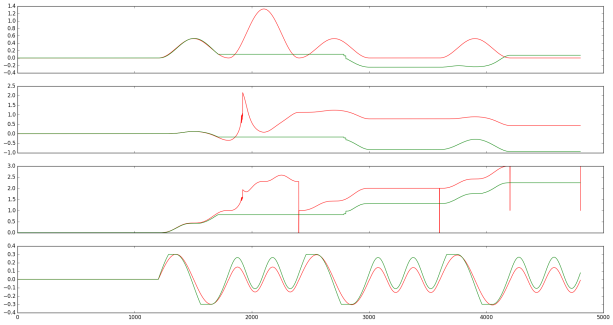

Figure 21: Sinusoid Controller rotateYaw(π) Desired vs Real for each state over time over time

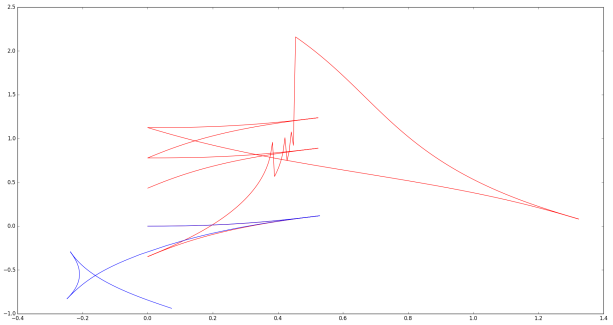

Figure 22: Sinusoid Controller rotateYaw(π) Desired vs Real position

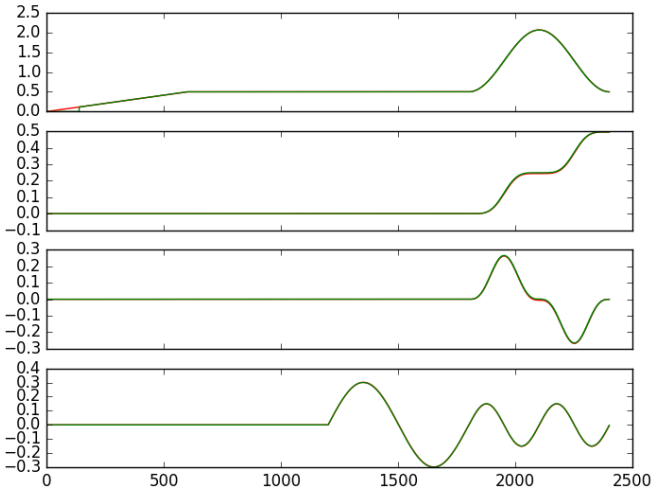

Figure 23: In Simulation Sinusoid ControllersTurn() Desired vs Real position

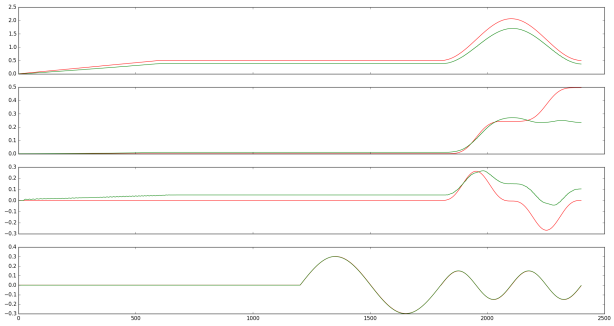

Figure 24: Sinusoid Controller sTurn() Desired vs Real for each state over time over time

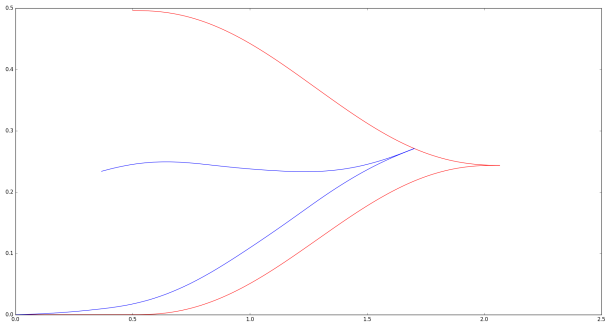

Figure 25: Sinusoid Controller sTurn() Desired vs Real position