Christian Tonnesen

ID: 260847409

2: Visual Tracking

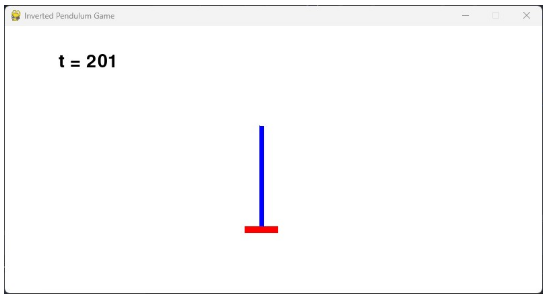
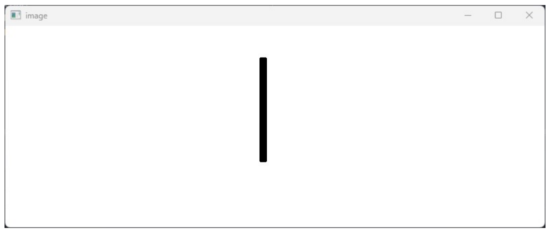
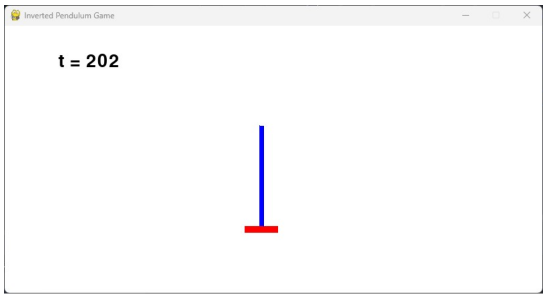
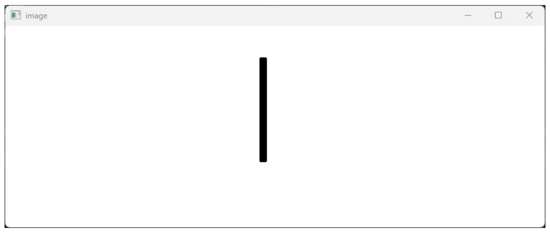
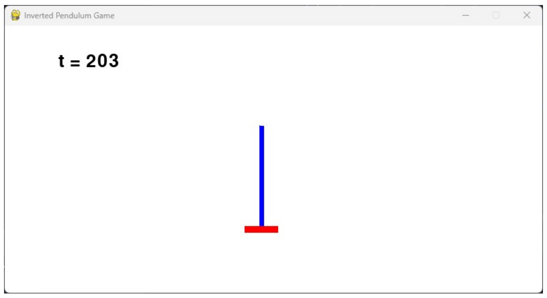
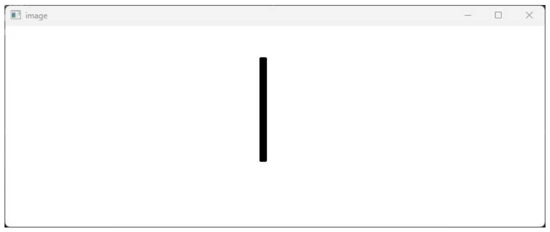
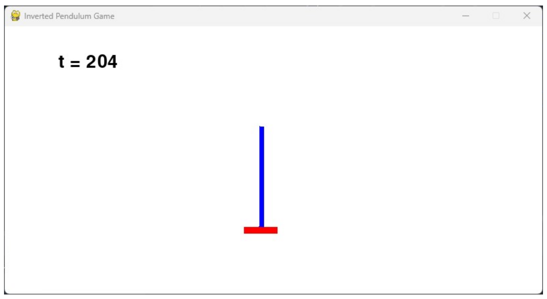
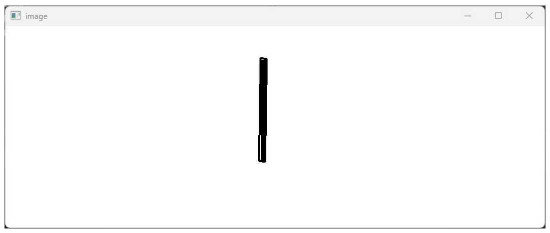
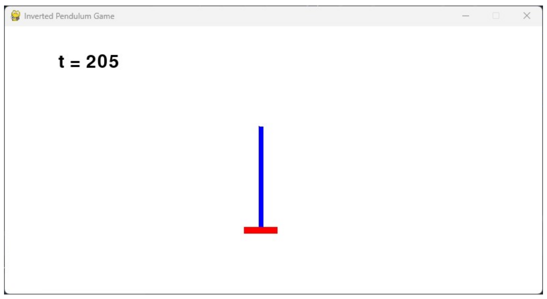
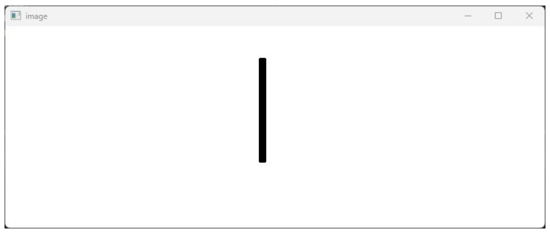
To perform the visual tracking, I leveraged the OpenCV software. At first glance, I could not find any direct way to measure the angle between the cart's top line and the actual pendulum, which would give us the `theta` value. I initially thought I could calculate the contours and box points of the cart, then use the top contour line of the cart and the closest line of the pendulum (if the pendulum was angled in the left, then use the left height contour line of the pendulum) to find an angle. However, I realized that this would quickly become a logistical nightmare to separate the contours of the cart from the pendulum's contours using just `x,y` values. I looked further into the documentation and discovered a function called `minAreaRect()` , which, when called on the contours of some object, would return the center, width/height, and angle of rotation for the smallest area rectangle the software can fit around the object. The key of this function was the angle of rotation, which essentially gave me the theta value, but only between `[-90,0]` degrees .

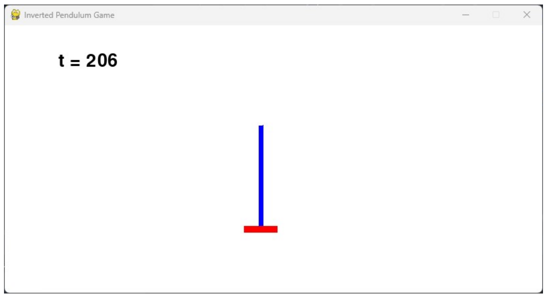
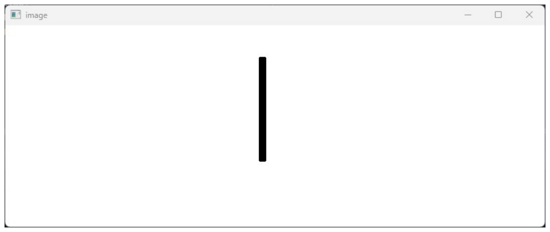
I first tried to implement this with both the cart and pendulum, but noticed I didn't actually need the cart aspect at all, just the pendulum. Given that the cart and the pendulum could be set to different colors, I set the cart to be brighter than the pendulum, grayscaled the current state image, then binary-thresholded the image so that everything was white except for the pendulum's pixels. I could then call the `findCountours()` function to get the outline of the pendulum, then feeding the contours to the `minAreaRect()` to find the angle of rotation. As mentioned earlier, though, the theta angle being returned was always between `[-90,0]` , which would not match the true `theta` . As such, I had to implement some code that would use the width and height of the rectangle, and then add/subtract the appropriate amount of degrees to bound the measurement between `[0,180]` . From here, it was a simple exercise of recentering the measurements around 90 degrees so that the `theta` values were measuring the difference from `pi/2` radians.

I did run into two issues though, first being that for timesteps before `timestep=2` did not have an updated `InvertedPendulumGame.surface_array` , which is responsible for storing a representation of the pygame display. Before `timestep=2` , the `self.surface_array` contained the title screen, which resulted in the `minAreaRect()` angle field returning `180` degrees (after conversion), since the shape it was using in place of the pendulum was the bottom leg of `Inverted Pendulum` in the title screen. To fix this, I had to reorder the rendering in `inverted_pendulum.py` to render the screen at the start of the first timestep and update the `self.surface_array` (see lines 292 to 305 of `inverted_pendulum.py`). If anything, this would lag the display by 1 timestep, but I did not see any noticeable difference. The second issue arose from the fact that the images of the current state of the pendulum also included the timestep in the top left corner, which would also be preserved after thresholding since it used black text. This meant that sometimes, the `findCountours()` function would target the timestep text and not the pendulum, messing up the `theta` reading. This fix was much simpler in that I just cropped the state image to remove the timestep text but preserve the pendulum, ensuring it was the only shape visible in the image.

As for the actual results, we can see from the below graph that the difference between the true and observed, called `TrueTheta` and `ObsTheta`, respectively, are quite close, within `+/- 0.001 radians` of each other.

In addition, below are 5 consecutive tracking frames from the OpenCV software tracking. The `minAreaRect()` box can be seen around

TimeStep	Regular Image	OpenCV Tracking
201	 t = 201	
202	 t = 202	
203	 t = 203	
204	 t = 204	
205	 t = 205	

TimeStep	Regular Image	OpenCV Tracking
206		

3: Kalman Filter

Implementing Kalman Equations

Implementing the Kalman filter was fairly straightforward. All the necessary equations were located in the slides and with some massaging of NumPy matrix multiplication, could be translated into Python quite easily. Our state transition function and covariance matrices looked like the following:

```
self.measure_var = R
self.process_covar = np.array([[Q, 0], [0, Q]])
self.F = np.array([[1, time_change], [0, 1]])
```

where `R` and `Q` were two floats provided as args to the `inverted_pendulum.py` script, representing the measurement and process covariance, respectively. `time_change` was the amount of time between state updates, which was pulled from the `InvertedPendulumGame.pendulum.dt` variable and used to calculate the change in `theta` from `theta + theta_dot * time_change` , performed with the code `self.state_matrix = self.F.dot(self.state_matrix)` .

Order of Operations and Kalman Inputs

The order of events was also pretty straightforward:

```
1. KalmanPredict()
2. GetObservation()
3. KalmanUpdate(observations)
```

```
4. PID.GetAction(KalmanThe
```

In my first iteration of the function loop, the final calls looked like:

```
self.save_current_state_as_image("")
self.kalmanfilter.predict()
obs_theta = self.get_observation()
obs_theta_dot = obs_theta - self.prev_theta
kalman_theta, kalman_theta_dot, p_matrix = self.kalmanfilter.update(obs_theta, obs_theta_dot)
self.update_lists(obs_theta, obs_theta_dot, kalman_theta, kalman_theta_dot, p_matrix)
self.prev_theta = kalman_theta
state = [self.timestep, kalman_theta, kalman_theta_dot]
action = self.PID_controller.get_action(state, self.surface_array,
            self.controller_mode,
            [self.kp_heuristic, self.ki_heuristic, self.kd_heuristic],
            self.generate_disturbance,
            disturbance_1_time,
            disturbance_2_time,
            random_controller=self.random_controller)
```

In my first implementation, I was sending both `ObsTheta` and `ObsThetaDot` to the `kalmanfilter.update()` function, where `ObsThetaDot = ObsTheta - self.previous_theta` (`self.previous_theta` being the result of the previous `kalmanfilter.update` iteration). In my mind, this meant that the Kalman filter could see the variance in its prediction for both parts of the state `(theta; theta_dot)` . However, after meeting with the TA, I was told this was not necessary, and more importantly, inaccurate as we only had one sensor and thus could not accurately measure `theta_dot` . As such, we would allow the Kalman Filter to do `theta_dot` estimations for us. I then amended the `kalmandilter.update()` to be:

```
kalman_theta, kalman_theta_dot, p_matrix = self.kalmanfilter.update(obs_theta, obs_theta_dot)
```

With this done, we had a working Kalman Filter that would run the Kalman predictions, take the `theta` observations, update the Kalman filter, then pass the recalculated `theta` to the `PID` controller (the Kalman-calculated `theta_dot` was not relevant for the `PID` controller I implemented).

Tuning Q, R, and Timesteps

After having implemented the Kalman Filter, values (`Q and R`) still had to be tuned in order to figure out the most stable. It was also during this period I also started to see strange results. For `theta` values, I'd see a high positive correlation (roughly indicating low numerical difference) between the `TrueTheta` and calculated `KalmanTheta` values, somewhere around `~0.7 to ~0.8` r-value. This was great, as it meant my observation model was accurately grabbing a `ObsTheta` and the Kalman Filter was effectively filtering noise to produce a more accurate `KalmanTheta`. My `TrueThetaDot` and calculated `KalmanThetaDot` values would also turn out to be highly correlated, but in the negative direction instead. This indicated my `KalmanThetaDot` values were similar to the `TrueThetaDot` in absolute value, but were of opposite signs. Further inspection of the raw values confirmed this, where every positive `TrueThetaDot` had a similar corresponding negative `KalmanThetaDot` value and vice-versa. I spoke with Dr. Lofti about this, who was equally perplexed and suggested I try playing around with both the `time_change` value, and the `dt` value.

I first started with changing the `Q` and `R` values, since those were the intended parameters for tuning, hoping to solve the correlation/sign issue with `ThetaDot` without having to affect any time values. To do this, I reused some of my old cold used for testing the PID controller in Assignment2, which simulated multiples runs of the simulation and then wrote averaged results to an accompanying CSV. The basic premise was that I would run inverted pendulum game for 3 rounds with a unique combo of `Q` and `R`, then get a "representative" value from the mean of the overall `TrueTheta` values to demonstrate the success of the `Q/R` combo, i.e. the closer the average `TrueTheta` was to 0, the better the model was at keeping then pendulum straight, indicating a better parameter combo. For the PID controller values of the model, I reused the ones found during Assignment 2, as I found those to be the most stable for the original pendulum, with `Kp=1.6, Ki=0.001, and Kd=49`. I tested every combination of `Q` and `R` from `-10` to `10` for each, as I figured this range would provide the full spectrum of `Q:R` ratios and performance. I was able to collect the averages in `Finals/QvsRThetaof-10to10` and found the best value (lowest) to be when `Q=10, R=4`, which, on average, produced a `TrueTheta` of `0.00215076106921965 radians`.

When I ran the model using this `Q` and `R` value, however, I found that I was again getting a good correlation for the `TrueTheta` and calculated `KalmanTheta`, but the same high negative correlation for `TrueThetaDot` and `KalmanThetaDot`. Thus, I decided I needed to try adjusting the time values. I devised two different testing schemes: one labeled `OverallTimeChange`, which would use the same value for both the `pendulum.dt` and Kalman Filter state transition function matrix `time_change` variable, and another named `KalmanTimeChange`, which would use two different values for `pendulum.dt` and the `time_change` state transition function matrix. The `KalmanTimeChange` scheme was theoretically incorrect as it assumed two different times between state updates depending on where it was used (contrary to reality), but I felt it was necessary to observe. I repeated the same methodology from the `Q` and `R` value testing, running one set of simulations where I set `pendulum.dt` and the `time_change` to every value between `-1` and `1` with a `0.005` step, and another set of simulations where I kept the `pendulum.dt` value at the default of `0.005` and adjusted the `time_change` between `-1` and `1` with a `0.005` step. For these simulations, I was instead looking to minimize the difference between `TrueThetaDot` and `KalmanThetaDot` (`TrueThetaDotKalmanThetaDotDifference`). After all the simulations concluded, I found the best set-up was the `OverallTimeChange` scheme of using the same value for both, where the time value was `0.03` and minimum theta_dot difference of `0.000108044393992631`. However, when running simulations with this set-up, we witnessed that the model had both a high `TrueTheta` value most of the time (not stabilized), as well as highly inaccurate `KalmanTheta` values. Thus, we had optimized the `TrueThetaDot` and `KalmanThetaDot` difference at the expense of our other performance variables.

It stood to reason then, we needed to concurrently look at the `TrueTheta` value, the difference between `TrueTheta` and `KalmanTheta` (`TrueThetaKalmanThetaDifference`), and the difference between the

`TrueThetaDot` and `KalmanThetaDot` in order to choose the best option for all three. I ran six different sets of simulations: three simulations sets grabbing the `TrueTheta` , `TrueThetaKalmanThetaDifference` , and `TrueThetaDotKalmanThetaDotDifference` using the `OverallTimeChange` scheme, and another three getting the same using the `KalmanTimeChange` scheme. The goal was to find the minimum sum of all the three representative averages across both testing schemes across all possible values, which would tell us the "optimal" way to run the model. For instance, say we used the `KalmanTimeChange` scheme with the default `pendulum.dt` value of `0.005` and `time_change` value of `0.2` . We would see that after three simulations, we would have the following averages:

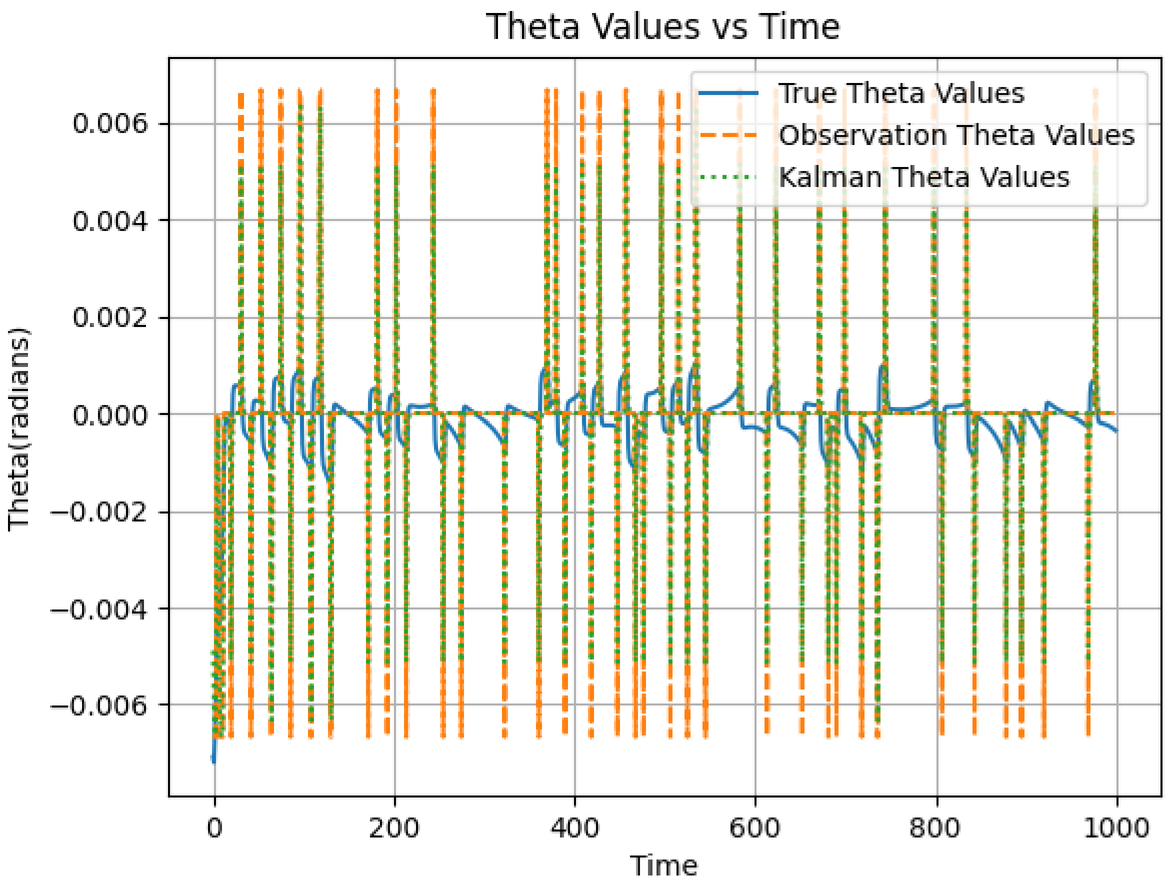
```
ThetaDifferenceFrom0: 0.00164592660690779
TrueTheta-KalmanTheta: 0.000192767498779813
TrueThetaDot-KalmanThetaDot: -0.00200313879350497

OverallMin = ABS(0.00164592660690779) + ABS(0.000192767498779813) + ABS(-0.00200313879350497) = 0.0038
```

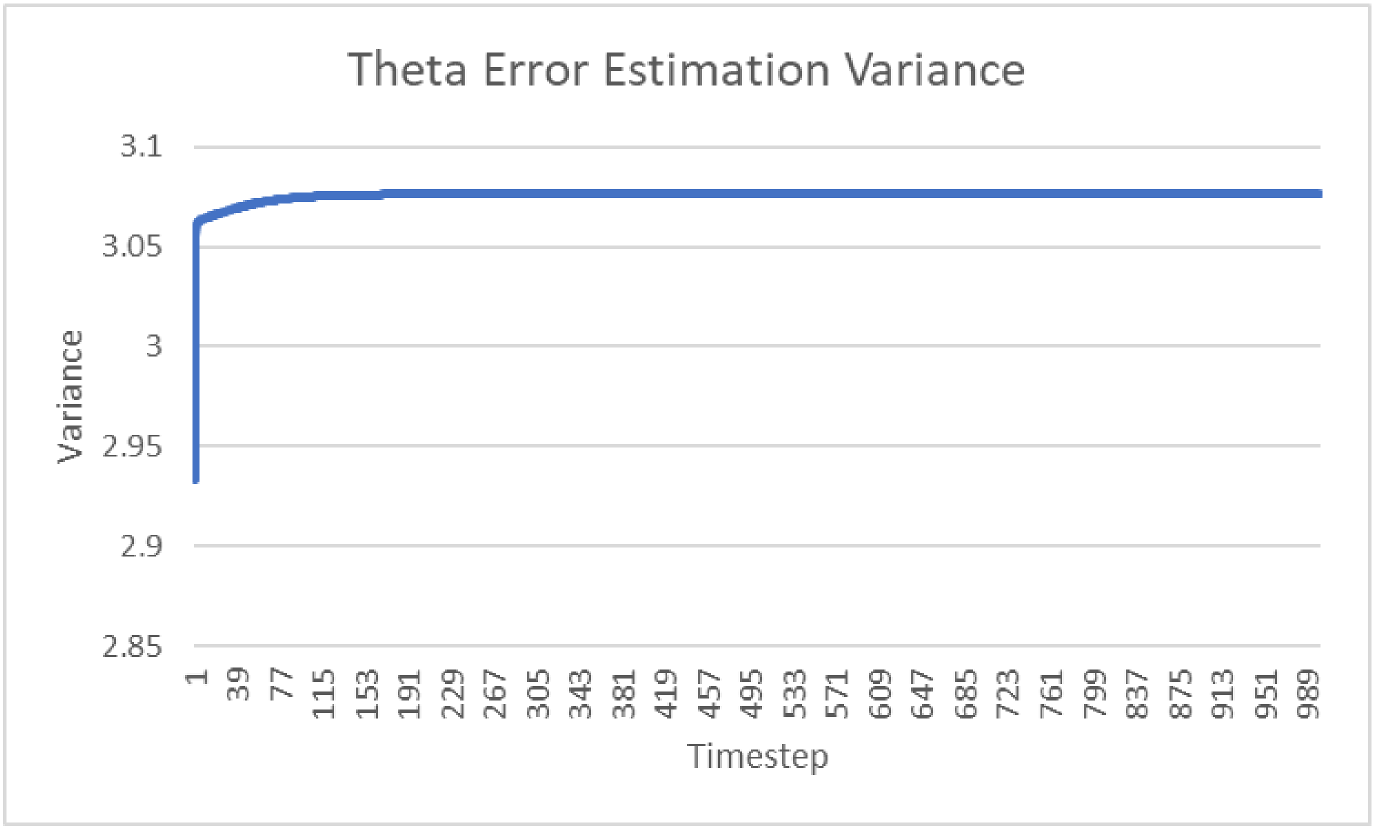
After running all timestep simulations (`400*400*3`), it was discovered that the best setup to minimize all performance variables was using the `OverallTimeChange` scheme with a `pendulum.dt` value and `time_change` value of `-0.015` (`OverallMin=0.00103218948719456`). When we ran the simulation with these parameters, we saw a correlation for `TrueTheta` and `KalmanTheta` of `0.497989364996573` and a correlation for `TrueThetaDot` and `KalmanThetaDot` of `0.378233278359509` . While `TrueTheta` and `KalmanTheta` is not as high as the original `~0.7 to ~0.8` , this is a direct consequence of balancing for the other performance variables, which can be seen in now having a positive correlation for `TrueThetaDot` and `KalmanThetaDot` . With these final parameters, the model also performs well in that it stays upright for all `1000` timesteps and shows minimal-to-moderate negative `x` drift for the cart position. All raw values for the six simulations can be found in `Finals/TimestepComparisonsQR.csv`

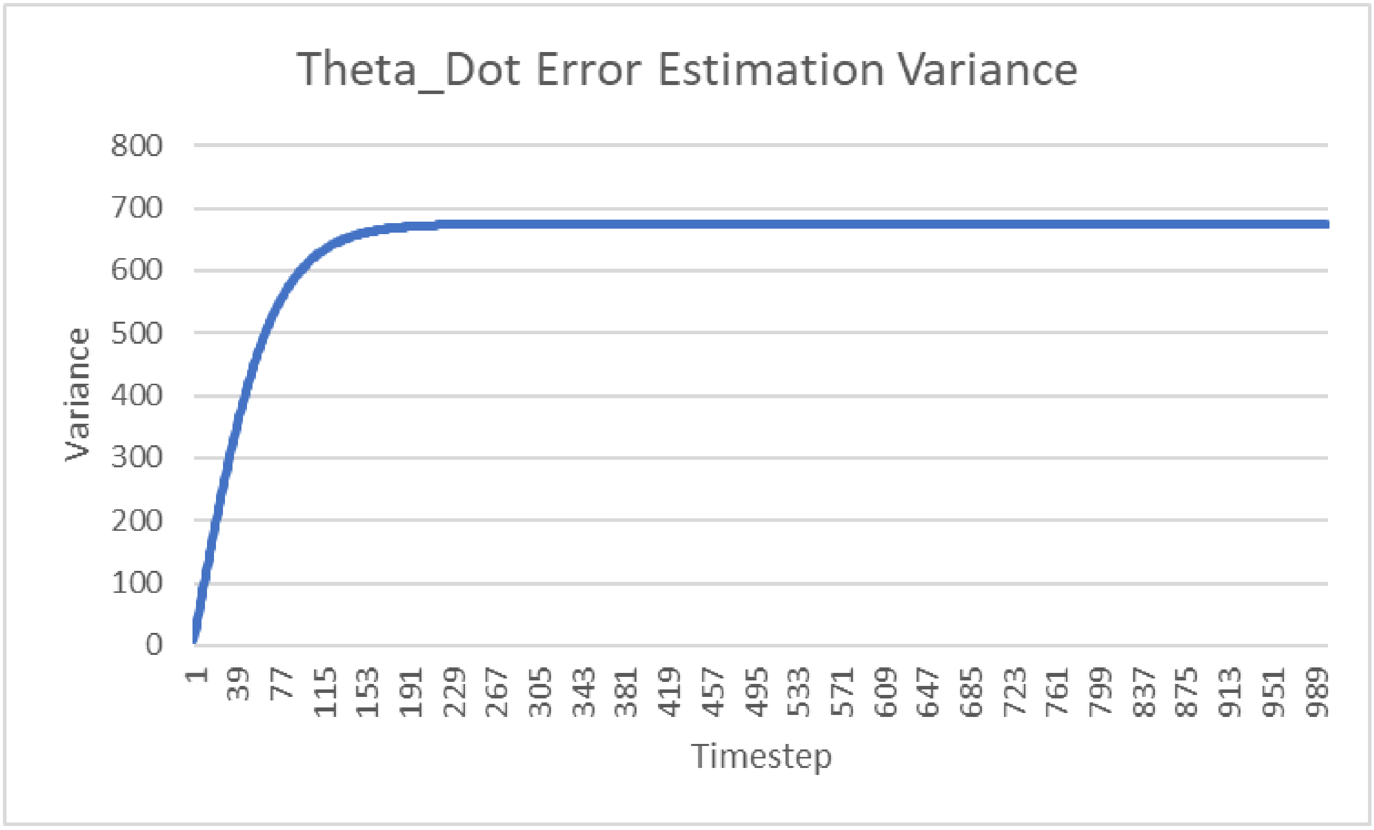
The following graphs display the performance of the model with the above parameters:

`True Theta vs. ObservationTheta vs. KalmanTheta values` . Note that the Kalman Filter (green) is functioning properly in decreasing the noise and producing results closer to the "true" value (blue) than the raw observations (orange).



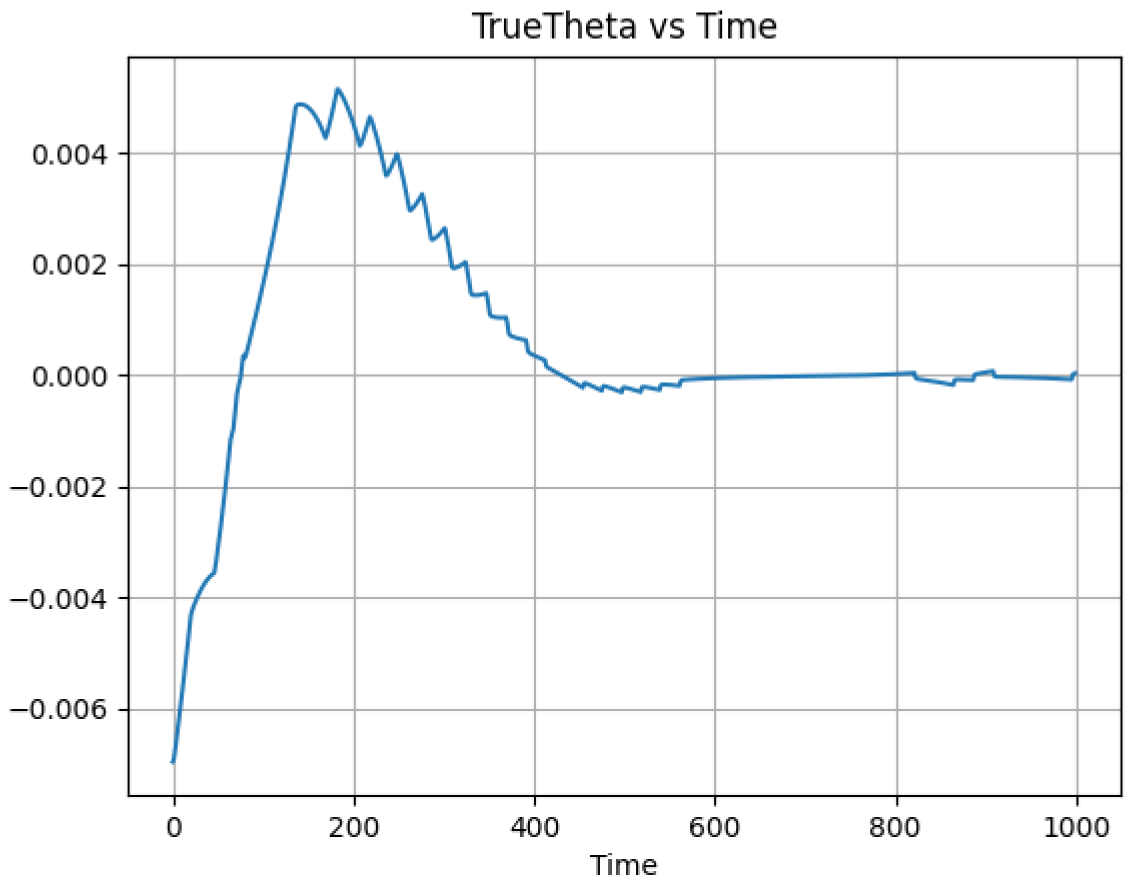
The estimation variance for both theta and theta_dot increases exponentially before plateauing:

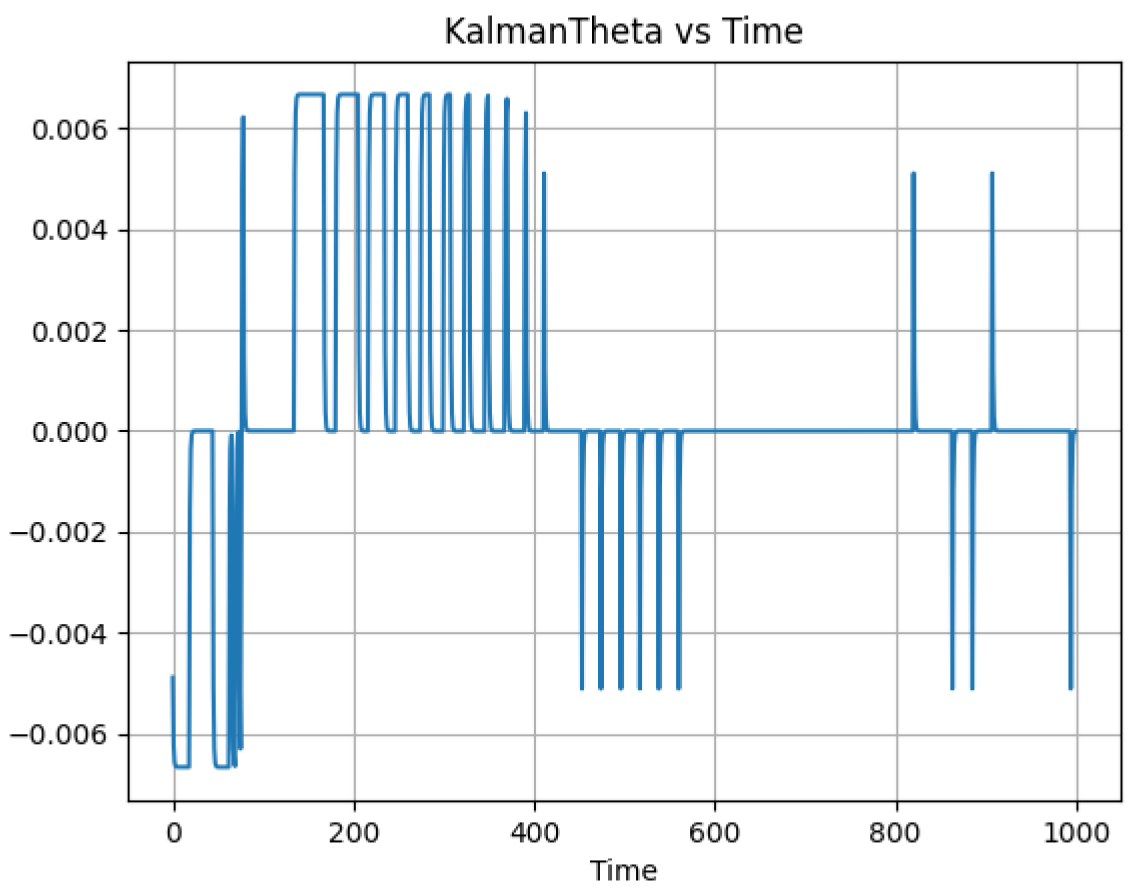

Theta Error Estimation Variance
3.1
3.05
3
2.95
2.9
2.85
Variance
1 39 77 115 153 191 229 267 305 343 381 419 457 495 533 571 609 647 685 723 761 799 837 875 913 951 989
Timestep


Theta_Dot Error Estimation Variance
800
700
600
500
400
300
200
100
0
Variance
1 39 77 115 153 191 229 267 305 343 381 419 457 495 533 571 609 647 685 723 761 799 837 875 913 951 989
Timestep

4: PID Controller

The controller was virtually the same used in Assignment2, with no tuning required, as the values chosen for `Kp, Ki, and Kd` produced stable results for all combinations of time schemes and `Q/R` values. When the final values for `Q/R` mentioned above were found, the controller performed as follows:





In the first graph, we can see that the model displays typical tuned `PID` controller behaviour, with high oscillations in the beginning before becoming dampened at around `timestep 500` with lower amplitude oscillations. This is because the addition of the derivative allows us to "slow-down" our action as our error grows smaller and smaller in approaching a theta difference of `0`. The integral term is subject to a "soft reset" when the value of the cumulative errors got too high in order to prevent a death spiral. I was initially concerned that this would replicate the issue of rendering any addition by the integral term to essentially nothing, similar to as when we made the `Ki` variable set very low, but I was intrigued to see that over the course of the `1000` timestep, the code only "reset" the integral about a 20th of the time, meaning it still made a useful contribution.

While more abstract and extreme in changes, we can see that the `KalmanTheta` graph is comparable to the `TrueTheta` graph, with many oscillations before `timestep 500` before settling out with minor deviations afterward for the remaining timesteps. Overall, the PID controller performs well when evaluating from either the "True" values and the Kalman values.