Modelling 15%

* Modeling:

Our team modeled the bicycle system with the state having a length of five. Three states corresponding to the given x, y, and theta dynamics, and two additional states for the uncertain bicycle parameters, B, and r. The reasoning behind this state vector extension is discussed later. For filter implementation, we discretized the given continuous dynamics for x, y and theta using the forward Euler method. The B and r dynamics were modeled as constant, with the addition of additive noise for each timestep, ie. B(k) = B(k-1) + v(k-1).

* Measurement likelihood function

to obtain the measurement likelihood function for our particle filter, f(z|x) we used a change of variables over w(k) using the nonlinear measurement function for z(k). Where w(k) was modeled as additive noise. After solving the change of variables it was found that f(z|x) = f(w(k)) = f(h(z,x)) corresponding to w(k) being modeled as normally distributed (see appendix for handwritten work). From this we were able to implement our measurement likelihood function in Python utilizing the matrix form of the gaussian pdf equation given in chapter one of our class notes.

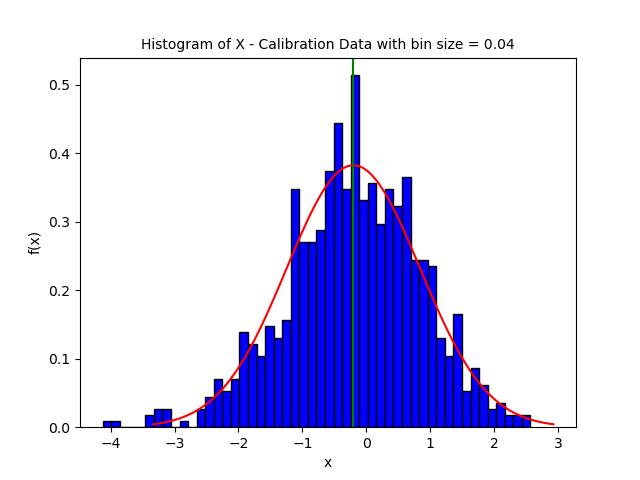
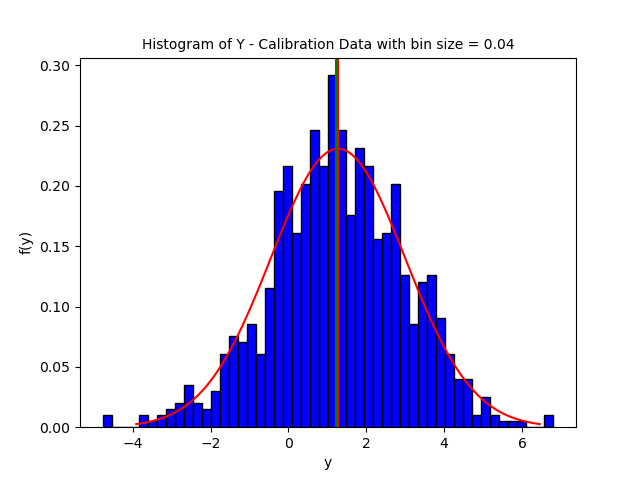
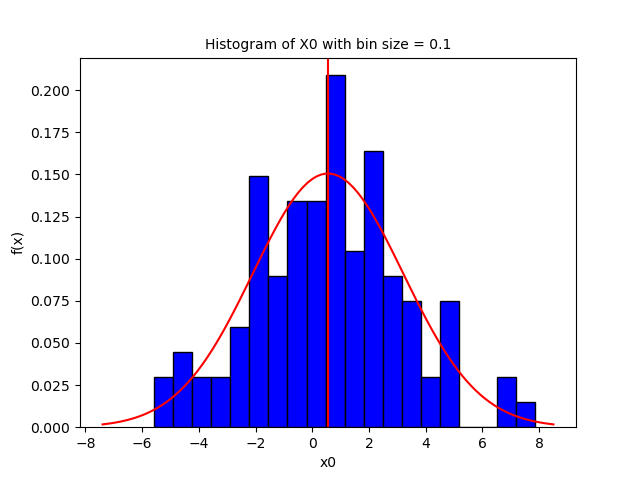
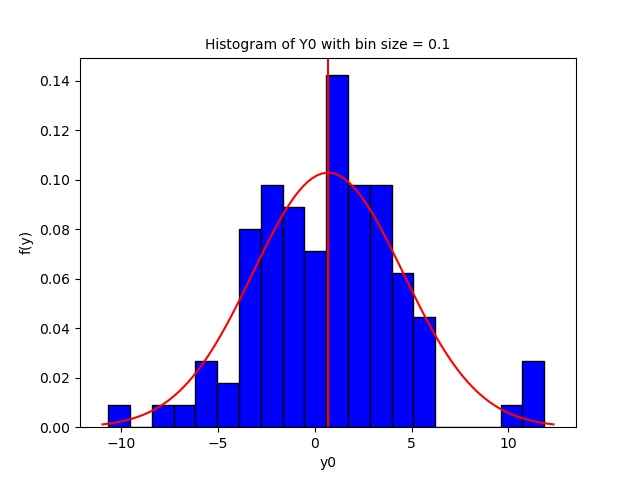
Design decisions and justification 15%

* Why PF

The first step in this project was to get an estimation technique up and running before we started tuning anything else.

We first tried the EKF, and it did estimate satisfactorily. However, we found that the EKF did not have as many available parameters to tune and improvements to make compared to what is available for the PF. Namely, for the PF we are able to add roughening to increase performance with sample impoverishment, and secondly we have the option to tune how many particles we are using for the PF. Further, we acknowledged that the EKF is estimating from merely a linearization of the system, and for moderately nonlinear dynamics, the PF can sometimes perform better than the EKF. Since our dynamics included a sine, cosine, and tangent term, we concluded that the dynamics are nonlinear enough for a PF implementation to possibly outperform our EKF implementation. For these reasons, it made sense for us to implement a PF for comparison purposes.

After having a successful estimation implementation for both the EKF and PF, we found them to have similar tracking performance. For the reasoning presented above that the PF gave us more tuning/experimentation options, we committed to the PF and started tuning accordingly as elaborated below.

* How we choose noise values
  + Process Noise for x, y, and theta
    - As the cyclist’s wheels were known to slip slightly and the input steering angle and pedal speed information were known to not be perfect, the system dynamics were modeled with an additive gaussian random variable representing this process noise. The mean for this variable was set to zero to represent unbiased noise. The variance for theta was set to the same variance as used for the variance of Theta(0) the reasoning behind which is explained in the Initial Values For Theta0 section. The variance for X1 and Y1 was set to 1. Values other than 1 were tested; however, no noticeable difference in the estimator’s performance was observed and no good reasoning for setting this variance value elsewhere could be determined.
  + Measurement Noise
    - As given in the project instructions, the measurements were known to be corrupted by unbiased noise. This noise was assumed to be additive and was modeled as a gaussian random variable. As the noise was due to “electrical noise in the sensor, timing imprecision, and atmospheric disturbances that warp the path of the GPS signals” all of which are unrelated to each other and uncoupled to the state, we used the fuzzy central limit theorem to determine that the noise would be a gaussian random variable. To determine the variance of the noise, we utilized the calibration data provided. We first wrote a function in python to load the calibration data, remove all all time steps where the measurements were “NaN”, and used the measurement model to convert the measurements from the center of the bike to X1 and Y1 coordinates. Then we used Numpy’s mean and variance functions to calculate the mean and variance of this data. Next, we plotted a histogram of the data with a normal curve overlaid in red corresponding to the mean and variance previously mentioned. Finally, we overlaid on these graphs the true state as given at the end of the calibration data as a green vertical line. These plots can be seen below. The histogram and overlaid normal curve match up well justifying the mean and variance calculated. The vertical red line representing the means calculated for X and Y lie on top of the green line representing the true state. This gives confidence to the mean calculation as the measurements were known beforehand to be unbiased. Thus, we defined the measurement noise as a gaussian random variable with a mean of zero and a variance as described above.
  + 
  + 
  + Initial Values for x0, y0
    - As the cyclist was known to start near the origin, the values for X1(0) and Y1(0) were assumed to be independent, gaussian random variables with zero mean. To obtain the variance of these random variables, we wrote a python script to iterate through all of the experimental data runs provided, find the first time step of each run where a measurement was not “NaN”, and store that measurement in an array. This array represented the first measurement received for each run. While these measurements did not necessarily represent measurements for X1(0) and Y1(0), they were the closest measurements to X1(0) and Y1(0) we had for each run. Thus we decided to utilize the variance of this array, calculated using Numpy’s variance function, as the variance of the gaussian random variables representing X1(0) and Y1(0). As in section describing our calculation of the measurement noise, we plotted a histogram of this data and overlaid a normal curve with the calculation mean and variance. These plots can be seen below, and the histograms appear relatively normal.
  + 
  + Initial Values for Theta0
    - As the cyclist was known to be initially headed North-East, Theta(0) was modeled as a gaussian random variable with a mean of pi/4. To obtain the variance for this variable, we assumed that the majority of the data for this random variable should lie within 0 and pi/2 as if theta lies outside of this region, representing strict North and strict East, one would not say the cyclist is heading North-East. Using the empirical rule that almost all of the data for a gaussian random variable lies within three standard deviations, we calculated the standard deviation for Theta(0) by dividing the aforementioned region (pi/2 - 0) by three and subsequently the variance by taking the square of the standard deviation.
* Dealing with inconsistent measurements
  + As the GPS measurements of the center of the bicycle were known to be inconsistent, meaning the controller did not receive new measurements every time step, we implemented a simple check each time step to determine if a new measurement had been received by checking if all the measurement values were not “NaN”. If a new measurement had been received, the controller would proceed with the measurement update step of the particle filter using that measurement. If, on the other hand, a new measurement had not been received, the controller skipped the measurement update for that time step therefore basing the state estimate of that time step on the prior update step. This strategy was simple yet robust against the inconsistencies in the measurements
* Extracting an estimate from the PF pdf

As stated in class, it is most common to extract the MMSE estimate (mean) from your approximated particle filter pdf at each timestep. For testing, we also attempted to extract the MMAE estimate (median) from our particle filter, but found that this made the performance of our PF significantly worse for multiple data sets.

* Roughening
  + As computational limitations set forth in the project’s instructions lead us to choose only 200 particles for our particle filter, we implemented a roughening step to help guard against sample impoverishment. This step was justified as we ran our estimator prior to implementing this step and saw the estimated values indeed clumped together. We implemented roughing using the equation given in the class notes:
  + 
  + Where K was our tuning parameter, Ei was the maximum inter-sample variability, d was the dimensionality of the state space, and N was the number of particles. A zero mean gaussian random variable with standard deviation sigma, as found by the equation above, was defined and N random values from this distribution were created and added to the particles after the measurement update step to roughen them. K was found heuristically, that is, we iteratively tuned K and ran the estimator on a number of data runs to analyze what value of K gave us the best performance. As the particle filter is a random estimator, judging the performance of runs against each other was difficult; nevertheless, we settled on using a value of K equal to 0.01. This small value of K slightly improved the particle filter’s performance by reducing the clumping we previously saw.
* Including B and r in the state vector

The baseline, B, and tire radius, r, of our bicycle was uncertain due to various issues such as flex due to the riders weight, inflation pressure of the tires, and other potentially non-constant uncertainties. This suggested to us that it would be beneficial to add the B and r parameters into our state vector to be estimated at each timestep.

We found it most sensible to initialize B and r’s approximated pdf at timestep zero to be centered at their nominal values, with their initial uncertainty equal to the process uncertainty used throughout the estimation problem.

At first it was unclear whether the uncertainty in the B and r parameters should be modeled as normally distributed noise, or uniformly distributed noise, so we tried both. We were given that the parameter B was uncertain to approximately +-10%, and r was uncertain to +- 5%. This meant that our modeling of these uncertainties for a uniform distribution simply corresponded to bounds that were +- 0.1 or +-0.05 times the mean value of each parameter respectively. The modeling of these uncertainties for a normal distribution corresponded to the standard deviations for B and r being one third times their manufacturer's suggested tolerance. We chose a standard deviation of one third the manufacturer's tolerance since 99.7% of a normal distribution lies within 3 standard deviations of its mean by the empirical rule.

The performance of the PF with uncertain parameters B and r included in the estimation vector was tested over multiple data trials, comparing uniformly distributed noise as described above, normally distributed noise as described above, and our control case of no uncertainty at all (zero variance). After comparison, estimating B and r with normally distributed noise resulted in the best particle filter performance.

* Evaluation & discussion using evaluation data (run #1) 20%

After running evaluation data (run #1) with our particle filter we obtained final errors as follows:

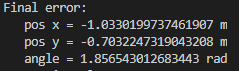
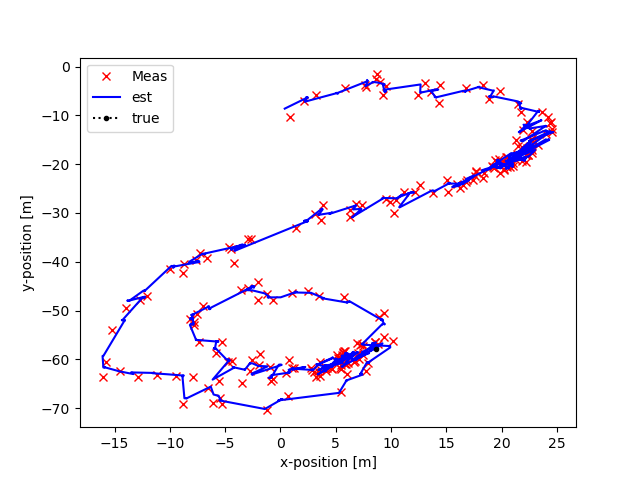
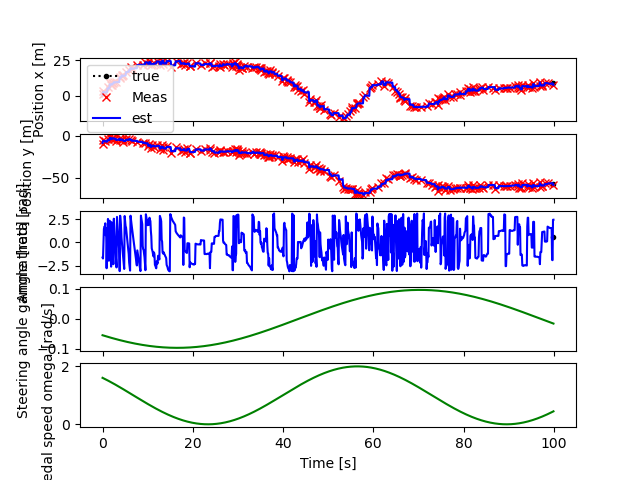
Pos x = -1.033 m

Pos y = -0.703 m

Angle = 1.857 rad

Moreover, the graphs of our estimator’s performance are included below. This data shows that our estimator’s performance overall is decent. The final error in X and Y is very good given that the error for both of them is around one meter and the only sensor used is a GPS sensor. In real life, position errors of one meter for a GPS sensor would constitute very good performance. On the other hand, the error for the angle, 1.857 rad, is bad. An angle of this magnitude means our estimator is estimating that the cyclist is oriented almost almost the exact opposite direction they are actually oriented.

Furthermore, after analyzing the x,y plot we found that the estimator seems to favor the measurement update as the estimate jerks toward new measurements. Also, after analyzing the plot of theta over time, we found the estimate of theta is quite jerky as well. One simple possible explanation of this behavior is that our particle filter needs more than the 200 particles we used to generate a good estimate. Given greater computational resource and/or computation time we could have initialized the particle filter with more particles thereby resolving this issue. A second possible explanation is that our noise model and/or values are wrong. For instance, the process noise may not be additive as part of the noise enters through the inputs and the variance of the process noise likely has a value other than the values we simply assumed for it. A final possible explanation is that the variance values for X1(0) and Y1(0) are wrong as the analysis done to calculate them utilized the first measurement of X1 and Y1 for each run which often was not at the first time step. Thus, our particle filter overall had decent performance which could be further improved with greater computational resources or better knowledge of noise variances.

* 
* 

Appendix

MODELING PDF HERE

MEAS LIKELIHOOD PDF HERE