Georgia Tech CS 8803-O01 Final Project

Artificial Intelligence for Robotics

Team 19: Phil Strawser, Sean Christe, Paul Dinh, Daniel Bansch
August 1, 2016

I. Introduction

The goal of this project is to predict the trajectory of a HEXBUG Nano micro robot in a wooden box with a cylindrical candle in its center. It travels around in this environment for 60 seconds while a camera positioned above the box records the robot. Noisy data is provided, corresponding to the position of the center of the robot in terms of x and y coordinates, which indicates the number of pixels from the left and top edges of the video, respectively. The robots trajectory for the next two seconds, in which no data or video is provided, must be estimated. In order to estimate the missing two seconds, a robot model, environment model, and a variety of filters were designed and tested.

II. Software Components

i. Models

The coordinates used to model the position and size of the candle are obtained using a python file called candle_position.py. This file uses OpenCV [4] to position a Hough Circle onto the candle using the provided video files. A screenshot of the video and GIMP [6] are used to manually estimate the position of each corner of the box. This data is combined in a file called box_model.py to create a 2D array of 1s and 0s that indicate whether or not the robot can travel to any given x, y position. Because this 2D array is static for all runs, it is serialized into a data file called grid.bin and can be reloaded quickly when the model for the box is initialized. A file called robot_model.py is used to model the state of the robot with a class called RobotModel, which includes all the localization and propagation algorithms, the estimated length and width of the robot, and the 2D array of the box and candle created by the box_model.py file.

ii. Moving Average Filter (MAF)

One of the primary algorithms used in this project for state estimation is a simple moving average filter. This algorithm essentially acts as a low-pass filter. A history of observations are passed into this filter, and a state vector is returned. This serves as an estimate of the robots final state corresponding to the most up-to-date sample in the history. Starting with the first observation, the algorithm iterates through the input set and calculates the heading and velocity between all the points. The velocity, in this case, is simply the distance between two consecutive points. The vector between two consecutive points is used to calculate the heading. Next, the turning rate and the acceleration are calculated by taking the difference between consecutive heading and velocity calculations, respectively. Finally, the estimated velocity is calculated by taking the average of all the calculated velocities. The average heading is calculated by normalizing the heading vector to a length of one, taking the average of the x and y components, and taking the atan2 of these averages.

iii. Unscented Kalman Filter (UKF)

Due to the nonlinear nature of the motion patterns of the robot, the standard Kalman Filter algorithm is not applicable. Instead, the Extended Kalman Filter (EKF), and the Unscented Kalman Filter (UKF) are considered. Determining the linearization of the dynamics to design an EKF is difficult, as creating the Jacobian for an EKF can be quite a challenge. A benefit of the UKF is that it uses a dynamics model that can also used by the other algorithms. Previous studies have shown, that for almost all systems, the UKF will perform at least as good as the EKF [5]. Because of the reusability of the dynamics, the lack of need for a Jacobian, and the parity of the two algorithms, the UKF is selected for this project.

To develop the UKF, a transition function is developed to described how the robot moves from some state at time k to some state at time $k+1$.

$$x_{k+1} = x_k + v_k * cos(\theta) \tag{1}$$
$$y_{k+1} = y_k + v_k * sin(\theta) \tag{2}$$
$$v_{k+1} = v_k + a_k \tag{3}$$
$$a_{k+1} = a_k \tag{4}$$
$$\theta_{k+1} = \theta_k + \omega_k \tag{5}$$
$$\omega_{k+1} = \omega_k \tag{6}$$

where:

x_k = position in x at time k
y_k = position in y at time k
v_k = linear velocity at time k
a_k = linear acceleration at time k
θ_k = heading at time k
ω_k = turning rate at time k

The pykalman Python library [3] is used to facilitate quick implementation and testing of the UKF. The AdditiveUnscentedKalmanFilter was chosen over the more generic UnscentedKalmanFilter due to its lower computational cost. The AdditiveUnscentedKalmanFilter class requires a transition function and an observation function. The transition function for the filter is created using the equations listed in Equations 1-6. The observation function needs to map from state space, with dimensions $M \times 1$ to observation space, with

dimensions $N \times 1$. A simple matrix multiplication operation is applied due to the linear relationship between state-space and observation-space.

The filter also needs a transition covariance matrix Q (transition noise), an observation covariance matrix R (measurement noise), the initial state mean, and an initial state covariance P. For the initial state estimate, it is common to average the first few points due to the sensitivity of the initial state covariance in a Kalman Filter [7]. A subset of the input observations were first passed through the MAF to get a reasonable initial state estimate, and then fed through the UKF to generate the state estimate.

With the current state estimated, the future predicted points are created by recursively passing the estimated state vector through the transition function. In order to tune the filter, a visual simulator is used to show how closely the filter is tracking the actual path of the robot. Tuning begins with the initial state covariance. This matrix has 6 different parameters that need to be tuned, which lie along the diagonal. These six values correspond to position in x, position in y, heading, turning rate, velocity and acceleration. Each of these six values affect the gain during the update phase of the filter. A higher parameter value allows the filter to react quicker to the input each time-step, whereas a lower value smooths out that portion of the state vector, and gives more weight to the initial state estimation. Each value was modified in turn, by a power of ten until reasonable tracking is observed in the simulator. Next, the observation covariance was tuned. While this is a 2×2 matrix, only the diagonals are tuned due to the x and y being uncorrelated with respect to measurement noise. Literature suggests that decent performance can be observed by setting the transition covariance to zero [7]. Another key parameter to set is the filter depth, meaning the number of samples used for state estimation. If too few are used, the filter does not converge. If too many are used, the filter will overfit the data, and will be slow to react when the robot comes out of a corner. Experimentally, it was found that around 10 samples for the initial state estimate using the moving averager, followed by 10 samples for the actual filter struck the best balance between accurate prediction in non-collision sequences of points, and recovering quickly post-collision with the environment.

iv. Particle Filter (PF)

The particle filter developed for this project has a fairly large deviation from a typical particle filter. Namely, it does not include the resampling phase, which is generally considered a core component of a typical particle filter. Instead of using resampling to converge the particles, an initialization using the MAF mentioned above is used. When the filter is initialized, it is fed a short history of observations. These observations are fed through the MAF to get an initial state estimate. Each particle is then generated based off of the initial state estimate. A little bit of noise is added to each parameter to give some variability, which is crucial for this class of filter. This bootstrapping step greatly improves execution time, since the averager requires a lot fewer calculations compared to propagating and resampling even a modest sized particle filter. Listing 1 shows the Python loop in which the particles are created.

Listing 1: Particle Generation

```
for i in range(N):
   x = random.gauss(avg_state[0], sigma_x)
   y = random.gauss(avg_state[1], sigma_y)
   h = random.gauss(avg_state[2], sigma_h)
   t = random.gauss(0, sigma_t)
   v = random.gauss(avg_state[4]*2., sigma_v)
   a = random.gauss(avg_state[5], sigma_a)
   particles.append([x, y, h, t, v, a])
```

After the particles are created, they are propagated using the same transition function developed for the other filters. After every propagation, the average position of all the particles is calculated and used as the position estimate for that time step. This process is repeated through all the propagations.

At first, the output from the averager was used unaltered. However, through manual exploration, it was discovered that the filter more accurately predicts the positions when the turning estimate is set to zero, with a little bit of noise. This is due to the fact that even a little bit of error in the turning rate can create large errors in the later estimation points. Using a visualization tool, it is also apparent that the particles are not propagating as far away from the initial estimation point, as would be expected. This is attributed to the shotgun scatter of the particles. At any given point in time, most particles will propagate in a forward direction. However, due to the randomness in the heading, the overall averaging suppresses some of the forward motion of the estimate. Therefore, the initial velocity is multiplied by a constant to make up for the disparity. Because most of the particles still moved in a forward direction, the constant was not cancelled out by backward or slower-moving particles.

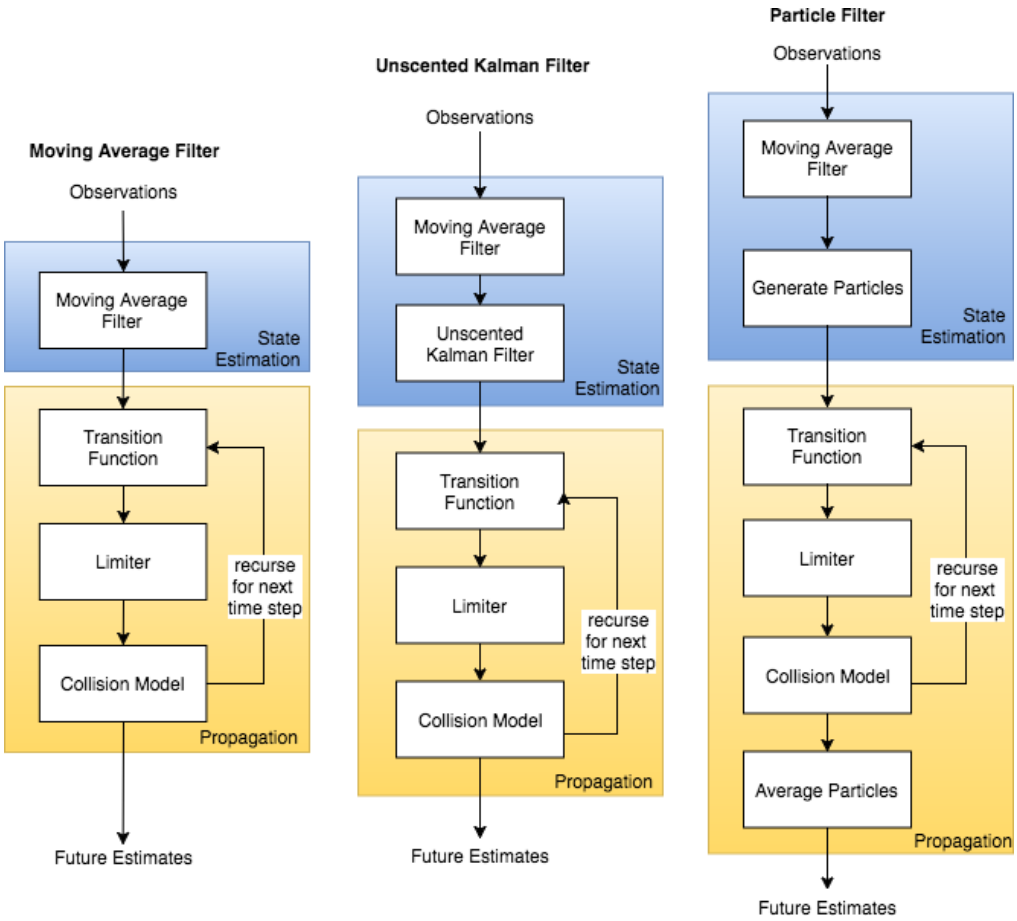


Figure 1: MAF, UKF, and PF Algorithms.

Figure 1 shows a flowchart for the three different filters utilized. In the first chart, observations are first passed into the MAF. The resulting state vector is input into the transition function to propagate the current position. This prediction is then fed into a limiter in order to ensure the predicted position is within the boundaries of the box. Finally, the collision model is used to appropriately reflect off any obstacles (box wall or candle) and the result is a final prediction for the next position of the HEXBUG. This transition $\rightarrow$ limit $\rightarrow$ collision propagation process is repeated to predict all 60 points.

v. Neural Network (NN)

The PyBrain [8] library is used to train neural networks for this approach. Our initial attempts at NNs used the x and y positions and combinations of motion parameters (velocity, acceleration, heading, turning angle) of the robot along with various motion histories and outputs (e.g. 10 frames input to 1 frame output, 60 frames input to 10 frames output). In the beginning, raw inputs were used without any type of feature scaling. Feature normalization was applied after the raw inputs resulted in poor performance. The NN saw low error in the cross-validation results using this approach, but was unreliable at predicting future positions in the test data. The problem was in using the actual position data as input, there was nothing but a few examples of that position for the NN to go off of when it saw it. In the cross-validation testing set, the positions and movements used were along the same path that the network was trained against, so it could make somewhat accurate predictions based solely on the fact that it had seen that path before. This algorithm needed a way to make the NN more motion-centric and less position oriented.

The final NN used the x and y position deltas between each observation rather than the positions themselves as both the input and output. Since positions were removed from the input, a method was needed to teach the network why the position deltas would be getting smaller or larger (such as before or after a collision). The idea of an "environment code" is introduced to be used as a discrete value to identify whether the robot was near an obstacle or out in the open. The heading from the first point is added in the motion history to the current point, to give it an idea of which direction it was traveling. This feature vector gave the NN pretty much all of the information it needed to know. The final algorithm uses a motion history of 5 to calculate the next dx and dy. After training the network, Python's pickle module is used to serialize the NN out to a file, so that the training can be quickly reloaded when predictions are calculated.

The NN performed well in the simulator, but did not perform well for the full 60 frame motion prediction.

vi. META Averager

Extensive testing with these four approaches did not produce a clear winner. While the particle filter seems to be the best performing out of all the algorithms, there are certain test cases where one of the other approaches results in a better score. For this reason, experimentation with a simple approach that merged all of our approaches

together is conducted. Without any modification of each individual approach, this meta averager simply averages the output propagations of each as its solution. As a further improvement, weights are added to bias each point estimate towards the better performing algorithms. The weights are derived experimentally by running a large test suite. The output of the test suite shows relative performance between the algorithms in terms of wins. A win is awarded to an algorithm when it performs better than the other algorithms for a particular run. The percentage of wins an algorithm recieved was applied as the weight for that algorithm in the meta-averager.

vii. Future Point Estimation

When running the different filters in the simulator, it was common to see predicted trajectories that made very tight spiral patterns. This is due to errors in the state estimation that were propagated through the predictions. To help deal with this issue, a limiter was placed on the output of the transition function, but only during the propagation step of the estimator. When the limiter was enabled during the state estimation stage, the UKF would break down completely because some of the matrices would become non-positive definite. The best results are achieved when the turning rate is restricted to zero. Many predictions would show the robot simply sitting still for the duration of the prediction. Therefore, a minimum threshold on the velocity and the acceleration are added to keep the predictions moving in a forward direction. While there are edge cases where the HEXBUG can get stuck, or even flip over, the limits provide a consistent lower score over many samples.

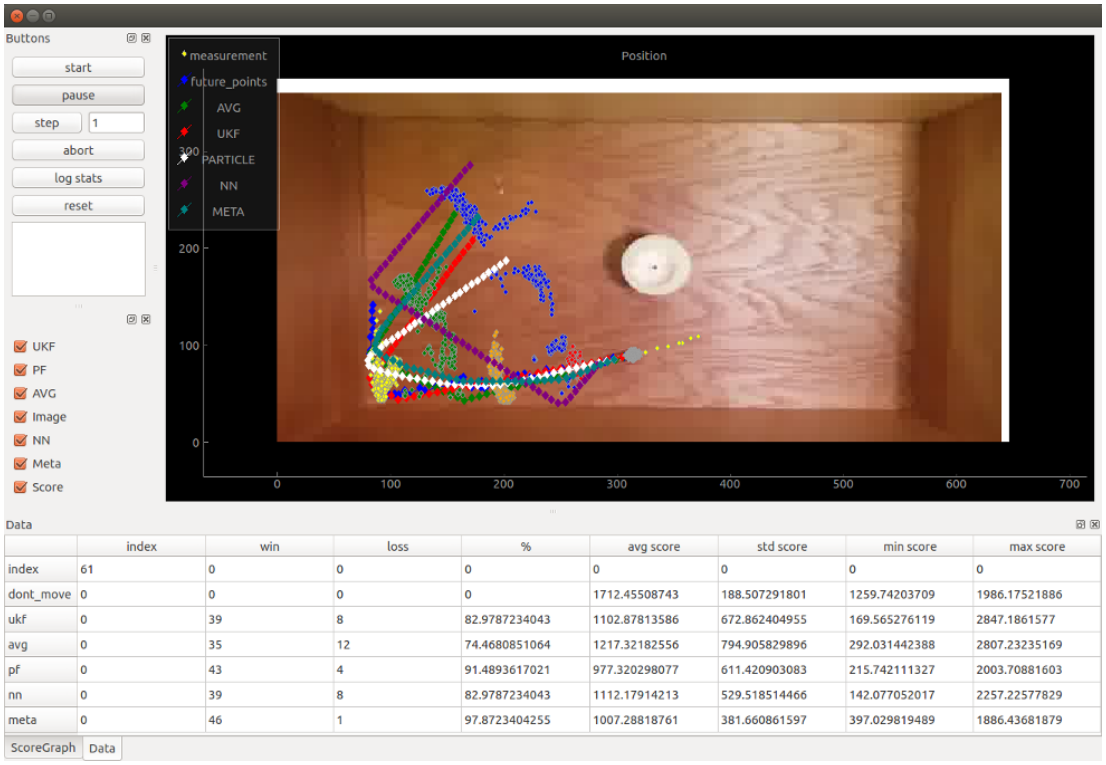
The filters have no way to model for the change of direction of the robot when it collides with the walls of the box or the candle. It is observed that when the front right part of the robot hits an obstacle the robot turns counter-clockwise by 90 degrees, and it turns clockwise by 90 degrees when the front left part hits. This is incorporated in the model by detecting which side of the robot collides, and adjusting the turning angle by 90 degrees.

III. Testing

i. Simulator

A graphical simulator was created using PyQt [2] and PyQtGraph [1] to facilitate the development and analysis of the various algorithms for the project.

Figure 2 shows the output of the simulator in the middle of processing one of the input files. In the main window, the current observation in addition to the 9 previous observations are displayed as yellow diamonds. The blue squares represent the next 60 observations from the current index in the data file. The remaining colored trajectories display the guesses for the next 60 propagations based on the current index. The boundaries and the candle are modeled using the box_model.py and candle_position.py mentioned earlier. To the left of the main window are the controls used to step through, pause, and log statistics at different points in time. Finally, the bottom of the simulator displays running statistics for the five different approaches. The ability to step through any of the given input files while visualizing the outputs proved invaluable for our final project. For an overview of the simulator running, a link to an unlisted video is provided: `https://www.youtube.com/watch?v=eZJg6P8Qe3M&feature=youtu.be`.



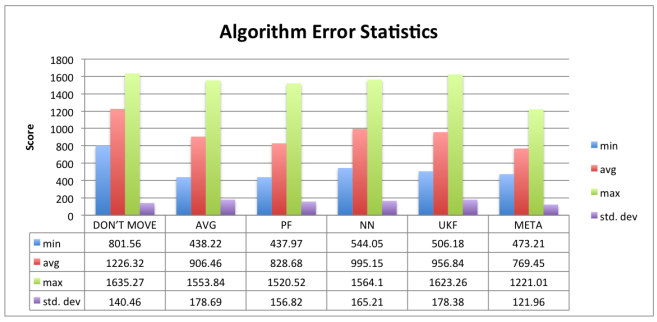
	index	win	loss	%	avg score	std score	min score	max score
index	61	0	0	0	0	0	0	0
dont_move	0	0	0	0	1712.45508743	188.507291801	1259.74203709	1986.17521886
ukf	0	39	8	82.9787234043	1102.87813586	672.862404955	169.565276119	2847.1861577
avg	0	35	12	74.4680851064	1217.32182556	794.905829896	292.031442388	2807.23235169
pf	0	43	4	91.4893617021	977.320298077	611.420903083	215.742111327	2003.70881603
nn	0	39	8	82.9787234043	1112.17914213	529.518514466	142.077052017	2257.22577829
meta	0	46	1	97.8723404255	1007.28818761	381.660861597	397.029819489	1886.43681879

Figure 2: The PyQt simulator.

ii. Results

The simulator proved to be extremely useful, but was lacking in that it could only process one input file at a time. A testing suite was developed to get around this deficiency. This suite selected a random chunk of contiguous samples from each file, splitting each chunk into an input set and a ground truth. The input sets were then run through each algorithm and the outputs were scored against their respective ground truths. The score for a particular prediction is calculated by taking the root-mean-square error between the a set of predictions and their respective ground-truth. Therefore, a lower "score" equates to a better estimate. For each run, various statistics on the scores were performed as seen in Figure 3. Using these statistics, the developers were able to gauge the performance of each of the algorithms (e.g a low std. deviation is a good sign that an algorithm is fairly consistent).

As mentioned earlier in describing how weights for the meta-averager is selected, another important statistic is the number of wins, which is the number of times an algorithm performs better than the others. This single statistic gives a good indication of relative performance among the algorithms. The chart in Figure 4 shows the META averager as the clear winner 53% of the time.



	DON'T MOVE	AVG	PF	NN	UKF	META
min	801.56	438.22	437.97	544.05	506.18	473.21
avg	1226.32	906.46	828.68	995.15	956.84	769.45
max	1635.27	1553.84	1520.52	1564.1	1623.26	1221.01
std. dev	140.46	178.69	156.82	165.21	178.38	121.96

Figure 3: Aggregated error results of algorithms.

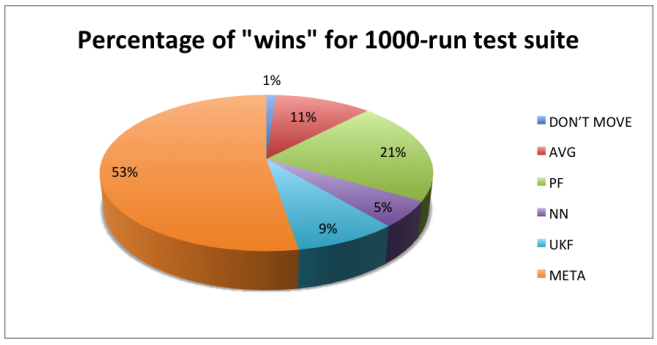


Figure 4: Aggregated win results of algorithms.

IV. Conclusion

A MAF, UKF, PF, and NN were developed and tested to estimate the trajectory of the HEXBUG robot as it moves through its environment. The PF performed the best overall, but proved to be lacking under certain conditions. To handle these cases, the META averager, a combination of the above approaches, was introduced. The META averager proved to have the lowest error statistics as well as the highest win ratio.

References

[1] Luke Campagnola. "PyQtGraph. Scientific Graphics and GUI Library for Python". In: *Poslední aktualizace* (2012).

[2] River Bank Computing. *PyQt*. URL: `https://riverbankcomputing.com/software/pyqt/intro`.

[3] Daniel Duckworth. *pykalman*. URL: `https://pykalman.github.io/`.

[4] Itseez. *OpenCV*. URL: `http://opencv.org/`.

[5] Joseph J Laviola. "A comparison of unscented and extended Kalman filtering for estimating quaternion motion". In: *American Control Conference, 2003. Proceedings of the 2003*. Vol. 3. IEEE. 2003, pp. 2435–2440.

[6] Peter Mattis, Spencer Kimball, Manish Singh, et al. "Gnu image manipulation program". In: *URL: http://www. gimp. org* (1995).

[7] Naren Naik, RMO Gemson, MR Ananthasayanam, et al. "Introduction to the Kalman Filter and Tuning its Statistics for Near Optimal Estimates and Cramer Rao Bound". In: *arXiv preprint arXiv:1503.04313* (2015).

[8] Tom Schaul et al. "PyBrain". In: *Journal of Machine Learning Research* 11.Feb (2010), pp. 743–746.